Auton Agent Multi-Agent Syst
DOI 10.1007/s10458-009-9101-1

# Agent-based architectural framework enhancing configurability, autonomy and scalability of context-aware pervasive services

**Nikolaos Dimakis · John Soldatos · Lazaros Polymenakos · Axel Bürkle · Uwe Pfirrmann · Gerhard Sutschet**



**Abstract** Multi-agent software architectures have gained in popularity due to their beneficial behavior in designing and implementing sophisticated applications. However, current approaches in implementing such architectures have led to application-specific, non-scalable implementations which limit the reusablity and improvement of the whole architecture. Moreover, these attempts lack features to enhance the user experience, thus slowing the adoption of the resulting services. In this paper we describe a fully-fledged multi-agent architecture covering a large variety of preferred features including capabilities of 'plugging' ubiquitous services, servicing mobile users, interconnecting remote similar architectures and interfacing with advanced software components such as knowledge bases. This framework exploits a wide-range of context-aware components making it essentially context-aware, allowing for the development of ubiquitous context-aware human-centric services, which are the focus of our research. To illustrate the flexibility of this architectural framework, we present four services which were built using this architectural paradigm by different development teams and elaborate on their overall behavior.

N. Dimakis (✉)
Intelligent Systems and Networks Group, Electrical and Electronic Engineering Department, Imperial College, London SW7 2BT, UK
e-mail: nikolaos.dimakis07@imperial.ac.uk

J. Soldatos · L. Polymenakos
Athens Information Technology, 0.8Km Markopoulou Avenue, 19002 Peania, Greece
e-mail: jsol@ait.edu.gr

L. Polymenakos
e-mail: lcp@ait.edu.gr

A. Bürkle · U. Pfirrmann · G. Sutschet
Fraunhofer Institute for Information and Data Processing, Fraunhoferstraße 1, 76131 Karlsruhe, Germany
e-mail: axel.buerkle@iitb.fraunhofer.de

U. Pfirrmann
e-mail: uwe.pfirrmann@iitb.fraunhofer.de

G. Sutschet
e-mail: gerhard.sutschet@iitb.fraunhofer.de

Published online: 01 August 2009

**Keywords** Multi agent systems · Pluggable architecture · Pervasive computing · Smart spaces · Context-awareness

## 1 Introduction

We are experiencing a rapid paradigm shift in modern computing systems [25]. The emerging pervasive and ubiquitous computing paradigm aims at transforming physical environments into intelligent active spaces. Within these environments, the end-users enjoy a large variety of non-obtrusive computing services which operate regardless of the users' time and location [56]. This new wave of computing services is based on the existence of a large sensor and actuator infrastructure which, in addition to specialized software components and middleware interfaces, forms a highly heterogeneous and complicated network structure. This structure serves the application of the ubiquitous services as it moves place the sensors and other devices into the background, making the overall human-human and human-computer interaction as non-obtrusive as possible by avoiding user distraction [19]. Furthermore, providing services to nomadic users needs to take into account user mobility and the changing context of roaming users [20]. It is therefore evident that context-awareness is a key characteristic of pervasive and ubiquitous computing.

This promising new wave of computing services does not come without a number of challenging issues. The first of these challenges derives from the nature of these systems. Mechanisms are needed to deal with the large number of the underlying sensor and actuator devices, including special modalities for mobile users. The solution comes by introducing mechanisms to orchestrate the whole range of sensors, actuators and users, as well as the interactions amongst them. In addition to that, the services should also come up with sensor data collection mechanisms, which can deal with various heterogeneous formats, while at the same time ensuring scalability and minimization of measurement errors [12,13]. Having reliable sensor inputs at hand, pervasive applications need to process them to extract context information. This processing is likely to entail sophisticated algorithms e.g. identifying people and objects and tracking their locations.

One additional issue deals with the overall context modeling for identifying composite contextual states to model human activity and interaction. These states are formulated by aggregating elementary context cues originating from low-level context acquisition components which base their operation on the underlying sensor set. These components process the sensor output and generate simple context, meaning that they are dedicated to providing their own perception of the current activities occurring in the smart space. This context needs to be available to all ubiquitous applications in a scalable fashion, which in turn apply the service logic based on the final composite context. Service logic execution involves the invocation of actuating services that increase natural interactivity with the end-users. Apart from the wide range of technology issues, pervasive computing services have to take into account individual users' concerns such as user preferences and privacy. Tackling with these issues requires methods for securing access to secret and private data.

Overall, the pervasive and ubiquitous systems are highly distributed and heterogeneous. Thus, the software architecture needs to be enforced with new middleware interfaces which can facilitate the development, interconnection and deployment of these services. Ultimately, they should assist in the overall service management and configuration which could introduce additional tedious tasks which would otherwise have to be addressed by the users. Moreover, these interfaces need to be equipped with additional features which enhance the overall

application, namely autonomicity, scalability and security. Finally the middleware layer should appear as a transparent layer to all modalities using it.

From a development point of view, the pervasive computing application comprises a large number of distributed objects. A promising paradigm facilitating the development of highly distributed infrastructures is the multi-agent software approach. Multi-agent systems leverage distributed object technology and are commonly used in this area of computing applications. Agents are software entities which function in an autonomous fashion in highly dynamic environments as they can be accompanied by sophisticated logic. As a result, agent technology provides capabilities that are perfectly in-line with the above characteristics of middleware systems for pervasive computing [48]. This is reinforced by the fact that major pervasive computing projects rely on agent-based middleware infrastructures.

Pervasive computing systems feature invisibility, heterogeneity, proactivity, mobility, intelligence and security. Deploying systems with all these features is technically challenging and an active area of research. Agent systems have been broadly deployed in the scope of pervasive services, towards addressing one or more of the above characteristics. These features, however, cannot be directly used to classify agent based systems for pervasive and ubiquitous computing. Most agent systems concentrate on more than one of the above aspects.

On the one hand, agents facilitate transparent and robust distributed communications and on the other, they introduce negligible overhead in the service development process. Therefore, early applications of software agents in pervasive computing focus on distributed communication, interoperability and integration of components. Moreover, software agents have been broadly exploited to facilitate interactions and coordination between humans, sensors, applications and devices.

The application of multi-agent systems in pervasive computing services, however, is not a trivial task for many reasons. First, agents should be autonomous enough to handle errors that might occur during the excution of a service without introducing distractions to the end user. Furthermore, the service should consider mobile users as well, and each service should not be considered solely as an isolated application but also be able to "interconnect" and combine its functionality with other services that are available. Additionally, the multi-agent framework should be designed in such a way that it does not introduce numerous processing and communication layers which would not favour the development of additional services. For instance, in the communication aspect, this could be achieved by adopting existing and widely accepted standards. As the pervasive services rely on a large heterogeneous network of sensors and actuators, the multi-agent framework should facilitate the integration of additional sensors and actuators, as well as combining the context that is produced by sophisticated context acquisition components. Finally, the multi-agent framework should be accompanied by an intelligent repository which is able to reply to intelligent queries that might arise during the durations of the considered events.

In this paper we present a multi-agent framework for designing, developing, deploying and managing sophisticated ubiquitous and pervasive services exploiting a wide range of sensors and actuators while maintaining a high contextual level. We illustrate the overall multi-agent framework and elaborate on the specialized individual goals of the individual agent-members and how they facilitate the ubiquitous application.

The motivation for implementing systems based on this architecture was our participation in the Computers in the Human Interaction Loop (CHIL) project [5], an Integrated Project (IP 506909) under the European Commission's Sixth Framework Programme, which is one of the most prominent European research initiatives in the areas of pervasive computing and multimodal interfaces. CHIL brought together several research labs, both in the EU and USA, and to build services that leverage numerous perceptual technology components. CHIL

perceptual technologies comprise a rich collection of 2D-visual components, 3D-visual perceptual components, acoustic components, audio-visual (i.e. multimodal) components, as well as output perceptual components like multimodal speech synthesis and targeted audio. As a result, CHIL services provide ground for demonstrating the benefits of the introduced software architecture. Note that CHIL services are built within prototype smart spaces which serve as test-beds for perceptual, multimodal and middleware development. The authors operate a prototype smart space comprising several sensors and actuating devices, where several pervasive applications have been built. Our experiences from building prototype pervasive services provide a first class manifestation of the middleware framework.

## 2 Related work

The area of multi-agent systems has been an active area of research, supported by a wide range of organizations (EU, DARPA, EPSRC etc.). Several frameworks have been introduced, many of which have managed to evolve and gain significant popularity.

A prominent example is the *Open Agent Architecture* [32] (the successor of the *Adaptive Agent Architecture* [7,30]), an agent-based system that supports task coordination and execution, as well as multimodal input integration. This architecture automatically transforms and/or interprets information exchanged between applications, humans and devices, allowing agents to be implemented in any language. Its core feature is a facilitator agent, which receives "advertisements" from other agents and coordinates the communication amongst them, making it a distributed computing environment rather than a multi-agent system.

A multi-agent platform that emphasizes sensor, device and application interoperability and coordination is *Hive* [33], which has been primarily used to connect devices together in support of some greater application. *Hive* agents provide information about humans, sensors and devices enabling assembly of applications that leverage information from all these sources. *Hive* agents enable distributed transparent autonomic communication that features fault tolerance and high availability. Similar to *Hive*, *MetaGlue* is another software agents' platform enabling natural interaction between humans and the environment [6]. *MetaGlue* agents represent humans, objects, devices and enable their autonomous transparent interaction. MetaGlue has been used to support projects under the umbrella of the Oxygen [11,43] program, which is among the most important initiatives for pervasive, autonomic and human-centric computing.

Moreover, additional attempts have been made which place their focus in solving specialized problems, rather than designing a generic architecture which can be used by various designers of different servies. For example in [21], the *Cougaar* architecture is presented which targets solving logistics planning problems (specified by the UtraLog project [55]), making the architecture difficult to adapt to other uses as it requires code changes and modification of the structure of plug-ins and agents. One additional drawback is the custom messaging protocol for the agent interaction, despite the fact that many standards have been proposed (such as FIPA, KQML etc.). Similarly, the *Soft Real-Time Agent Control Architecture* [23] focuses on addressing the needs which arise during the real-time management of sensors in real world deployments, which require special treatment by the software architecture. The *Smart Classroom Project* [45], introduces a multi-agent software architecture which deals with enhancing the educational experience. *DimaX* [17], deals with failure management, scalability and adaptability in multi-agent systems, by introducing reusable fault-tolerant techniques for service developers.

*RETSINA* (Reusable Environment for Task-Structured Intelligent Network Agents) [51] is a well-known open MAS infrastructure that supports communities of heterogeneous agents. It provides two types of communication mechanisms; one for message transfer between peers, the other for multicast that is used for a discovery process to let the agents find infrastructural components. *RETSINA* provides matchmaker agents which are used to receive advertisement of services by service providers, and get information of relevant providers. The *RETSINA* agents communicate using KQML-compliant messages.

The *A-globe* agent platform [46] was designed to provide fast prototyping and application development, by supporting the communication in cases when agents become inaccessible from the rest of the agent community, agent migration and deployment on remote containers. These features make it a well suited platform for simulation and implementation of physically distributed agent systems with applications ranging from mobile robotics to environmental surveillance by sensor networks. *A-globe* was designed as streamlined lightweight platform which will operate on normal PC as well as mobile devices (PDA). It is able to cover a wide range of scenarios; however, its communication protocol inside the inter-platform domain is custom and does not follow any standard (such as FIPA).

*JADE* [26] is a very popular software framework for developing agent applications in compliance with the FIPA [18] specifications for interoperable intelligent multi-agent systems. The goal is to simplify development while ensuring standard compliance through a comprehensive set of system services and agents. *JADE* can be considered as an agent middleware that implements an agent platform and a development framework, as it deals with all aspects that are not peculiar to the agent internals and that are independent of applications.

In addition to middleware frameworks focusing on the development of multi-agent systems, a number of agent specific frameworks have been introduced which have gained significant popularity. *2APL* [10] and *Jadex* [40,27] are Belief-Desire-Intention (BDI) programming frameworks, which allow using higher-level agent concepts for implementing agents and facilitate the design and implementation of intelligent agents. *2APL* agents can generate plans by reasoning about their goals and beliefs, which are implemented in a declarative way. Plans can consist of actions of different types. Like most BDI-based programming languages, *2APL* provides different types of actions such as belief and goal update actions, belief and goal test actions, external actions (to be performed in the agents shared environment), and communication actions. Similarly, *Jadex* introduces a BDI layer on top of *JADE*. A *Jadex* agent has two basic parts; an Agent Definition File (ADF) written in XML and a set of Java classes, which specialize *Jadex* built-in classes, to specify how plans (intentions) are constructed out of beliefs and goals (desires). It supports two kinds of plans, service plans, which execute continuously, and passive plans, which execute only when triggered. These plans are implemented in Java, and their triggering conditions are specified in the agent's ADF.

*Jason* [28,4] falls in the same category as *2APL* and *Jadex*; however, it follows a different approach. It implements the operational semantics of AgentSpeak(L) [42], a BDI logic programming language. The type of agents specified with AgentSpeak(L) are sometimes referred to as reactive planning systems. A *Jason* multi-agent system can be distributed over a network effortlessly. *Jason* is equipped with a number of preferred features such as strong negation, handling of plan failures, the possibility to run a multi-agent system distributed over a network, fully customizable selection functions, trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting).

Also, the work on Electric Elves (E-Elves) [41], focuses on Human-Agent teamwork, and demonstrates how agents can improve the efficiency of a team of users. Every user on the team has an agent representative that knows the user's location and his agenda. If an

appointment needs to be changed, the agents can cooperate to reschedule, taking into account the schedule of their user as well as other constraints like room availability. An important feature of the system is that it allows for mixed initiative interaction, i.e. both the user and the agents can take the initiative to make or change appointments. The agents have adjustable autonomy regarding their initiative. For example a user can choose to let his agent reschedule his appointments autonomously or to be informed when a rescheduling is needed and then reschedule himself.

## 3 The CHIL agent framework

To clarify the scope of this paper we briefly describe the complete system architecture in the next section. Our overall architectural framework has been described in detail in [47], including sensor control and context acquisition interfaces for context-awareness. The rest of this paper will focus on the service framework of the architecture, which is implemented as a society of software agents. We will discuss the benefits of the agent society in the scope of designing and implementing human-centric ubiquitous services.

The implementation that we present in this paper is based on a different approach in modeling the activity of agents than BDI. Instead of following the BDI paradigm, each agent (or service) in the CHIL Framework can use its own "situation model", which describes the transition from one state to the next [8] according to environment conditions or state of other agents. Starting at a given state, and given a set of contextual information that captures the state of the environment, the agent or service can trigger appropriate actions depending on whether such a transition is possible. Situation models can be described in XML format and parsed at the initialization phase of each agent [16], which allows for increased flexibility when multiple development teams need to collaborate on the same service. In general, the CHIL Framework has been designed so that each individual development team has discrete roles and should not focus on specific details on how to interface with the overall framework. To the best of our knowledge the CHIL architecture, and the CHIL Agent Framework in particular, provide an excellent set of tools and libraries which facilitate the integration of context-aware services to support human interaction, in smart spaces.

In essence the requirements for the success of an agent framework to support such services are: (1) to introduce a low processing overhead as each agent should be able to run on light-weight devices such as PDAs, (2) to be able to handle failures that might arise during an interaction event, without the user being aware of this, (3) to follow standard interaction protocols with the use of ontologies so that services could interact during an event with common vocabulary, (4) to separate the application logic from the communication channels, (5) to allow interfacing with a global, consistent information repository, that is able to answer intelligent queries, and (6) to provide automated mechanisms to service the end user. We claim that the CHIL Agent Framework provides a well-rounded "marriage" of all these requirements.

The CHIL Framework is designed to tackle several issues which arise when a number of different services are to be interfaced, which may require the integration of heterogeneous components, but not necessarily agents[16]. We place our focus on human-centric services which interface with humans in a non-intrusive fashion. Our framework allows for a number of different services to co-exist and interface with each other, and to take advantage of a wide range of sensing and actuating equipment. In subsequent sections of this paper, we elaborate on the features of our platform which briefly are:

- To tolerate both service and hardware failures,
- To provide a mechanism for enabling the implementation of service specific code in pluggable handlers by keeping the agent service-independent,
- To allow for a centralized information repository, maintaining all the required information about the components/agents/devices that are available in the smart space,
- To interface with different devices, such as smartphones and PDAs,
- To allow the interconnection of similar "light-weight" smart spaces to interface with the main one,
- To facilitate the integration of different ubiquitous services, and
- To allow the services to interface with each other and exploit the joint functionality (e.g. an intelligent recording service combined with a meeting support service).

These features have been optimized to meet the requirements of a family of autonomous human-centric services. For instance, failure handling has been addressed in a number of existing frameworks in the literature; however, the nature of our scenarios prohibits the application of existing techniques due to the fact that our scenarios require the coordination of a large number of different components which affect each other (e.g. a video recording component requiring a camera that is, given a set of characteristics, retrieved from a knowledge base, or a personal agent that is activated as soon as a person enters the room, etc.). As an additional factor, we point out that the user should not be aware of any recoverable error that might arise, and that the service should tackle each case in a transparent fashion, delivering an uninterrupted service to the end users. The final aspect is of major significance to the CHIL Framework, and our focus was to design an architecture that, given the constraints that arise from the nature of such services, can support the users during their interaction, and to provide a rich set of features.

The JADE (Java Agent Development Environment) platform [26] was selected to be the backbone of our system as it is equipped with many features, which make it a highly flexible and secure platform. As seen in [1], JADE performs adequately in many of the evaluation criteria which include agent mobility capabilities, administration, network traffic issues, stability, security, debugging capabilities etc. Finally, JADE is compliant to the FIPA (Foundation for Intelligent Physical Agents) [18] standard for agent communication, a standard which is being used by a plethora of novel multi-agent systems. All agents in the system are based on the JADE Agent that provides basic agent functionalities like sending and receiving messages, life cycle management etc. JADE is the most mature implementation of the FIPA specifications. It comes with many tools to administer and perform tests with the agent platforms: a GUI for inspecting the state of running agents, launching new agents and sending messages to agents; a sniffer agent and more elaborate examples.

The CHIL Agent Framework tries to place a "shell" around each agent or service, enabling it to interact with the smart space, and to provide an unobtrusive human-centric service. This "shell" is shown in Fig. 1. Our framework shares some common characteristics with the frameworks presented in Sect. 2, such as the BDI framework, and that is on the issue of decision making depending on the environment conditions. In contrast to the BDI paradigm, the CHIL agents allow the integration of "situation models" [8] which model the human interaction in the smart space by identifying milestones that would appear in a human-interaction event. These models can be designed by a member of the service design team and do not need to be formulated in a proprietary fashion, a key parameter when numerous developing teams are collaborating. For instance, the situation model of the Memory Jog service described in Sect. 4.1 is described in an XML format [16]. Furthermore, the CHIL Agent Framework places significant effort in using standard interaction protocols (FIPA), which are


**Fig. 1** How the CHIL Agent encapsulates the JADE Agent with the pluggable behaviors, autonomy, device independence, interfacing with the knowledge base, and the situation modeling

implemented by behaviors loaded on run time (pluggable behaviors), and using a common intelligent information repository that can be accessed by any component of the CHIL architecture. Such features have made the CHIL Agent Framework a very strong backbone of the CHIL services. The prototype applications that we present in Sect. 4, have been developed by different teams of the CHIL Consortium using the same agent framework.

The CHIL Agent Framework has been meticulously designed to meet the requirements that appear in pervasive human-centric context-aware services. It targets primarily environments which are equipped with a large collection of sensors, actuators, and perceptual components and tries to orchestrate their interactions in delivering useful services to the end users. Compared to other frameworks such as A-globe or RETSINA the CHIL Agent Framework provides a fully FIPA-compliant communication scheme, as not only a number of different services and components need to be able to interact, but also all developing teams should have a common vocabulary and interaction protocols. Furthermore, the CHIL Agent Framework has specialized agents which are either pending on requests by other agents, or continuously "scout" the network for critical conditions, such as the failure of an agent, and act accordingly. Moreover, the CHIL Framework has three types of management services, not only on the sensor and agent level (using each middleware's management tools), but also using the Smart Space Resource Manager (SSRM) [49], which monitors and controls the whole infrastructure such as the perceptual components, sensors, and actuating services available. One additional usage of the SSRM is to facilitate the deployment of the services by providing an overview of the current perceptive capabilities of the smart space in terms of sensing and actuating equipment as well as the presence of context-acquisition components. Finally, the SSRM is accessible by any component in the CHIL Framework using webservices.

### 3.1 The CHIL reference architecture

The CHIL Reference Architecture (also called the Ice Cube) provides a collection of structuring principles, specifications and Application Programming Interfaces (APIs) that govern the assemblage of CHIL components into highly distributed and heterogeneous interoperable systems. The CHIL Ice Cube is designed as a layered architecture model (Fig. 2). Each level derives from the abstraction of the information processed and data flow characteristics such as latency and bandwidth, based on the functional requirements from the CHIL system design phase.

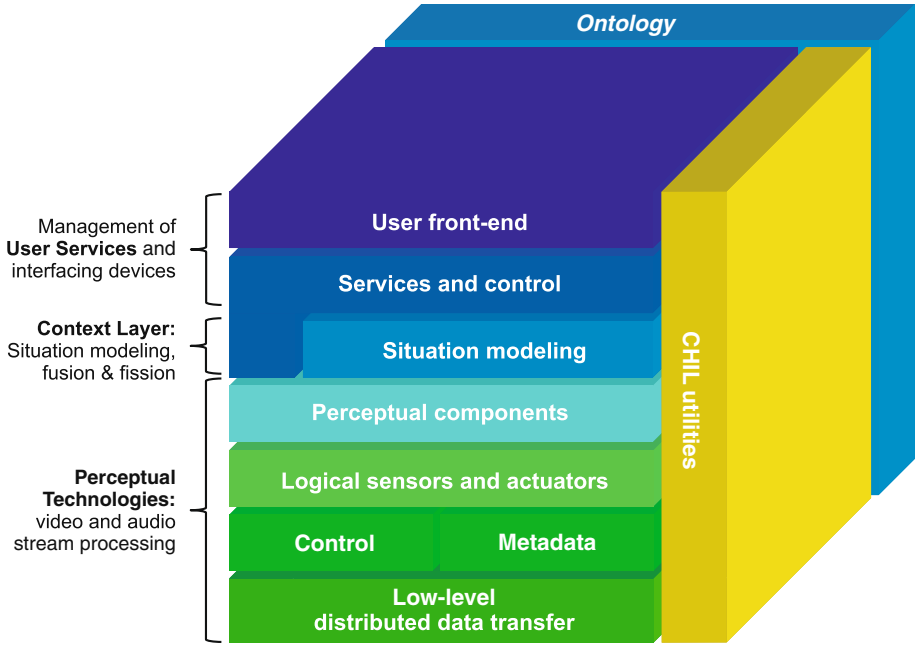



**Fig. 2** The layer model of the CHIL Reference Architecture (Ice Cube)

The upper layers—User Services—manage interactions with humans by means of user services hosted on various interaction devices. The components at this level are responsible for communication with the user and for presenting the appropriate information at appropriate time-spatial interaction zones. The User Services utilize the contextual information available from the Situation modeling layer. In projects like CHIL, where a number of service developers concentrate on radically different services, it is of high value that a framework ensures reusability of services. To this end, we have devised a multi-agent framework that:

- Facilitates integration of diverse context-aware services developed by different service providers. Facilitates services in leveraging basic services (e.g. sensor and actuator control) available within the smart rooms.
- Allows augmentation and evolution of the underlying infrastructure independent of the services installed in the room.
- Controls user access to services and supports service personalization through maintaining appropriate profiles.
- Enables discovery, involvement and collaboration of services.

The middle layer—Situation modeling—is the place where the situation context received from audio and video sensors is processed and modeled. The context information acquired by the components at this layer helps CHIL services to respond better to varying user activities and environment changes. For example, the Situation modeling layer answers questions such as: Is there a meeting going on in the smart room? Who is the person speaking at the whiteboard? Has this person been in the room before? Specifically, this layer is also a collection of abstractions representing the environment context in which the user interacts with the application. It thus maintains an up-to-date state of objects (people, artifacts, situations) and their relationships. The situation model acts as an inference engine, which watches for

the occurrence of certain situations in the environment and triggers respective events and actions.

The lower layers—Perceptual Technologies—host perceptual components that extract meaningful events from continuous streams of video and audio signals. All kinds and variations of perceptual components reside at this layer to perceive user actions in a smart environment. In CHIL we actually counted a total of 64 such components provided by 8 different suppliers. These components process the sensor signals from one modality (body trackers, face recognizers) or modality combinations (audio-visual speech-recognition) and deliver the events to upper layers. The CHIL Reference Architecture defines the API contract (Access, Subscriber, Control, Introspection, and Admin APIs) as well as the lifecycle (Unregistered, Registered, Launched, Running) to which the perceptual components must adhere to be considered CHIL-compliant.

The vertical columns—CHIL utilities and Ontology—provide support across layers. The CHIL utilities provide global timing and other basic services that are relevant to all layers. The Ontology provides a definition of CHIL concepts and a directory service that provides access to information at any layer. It is worth emphasizing that the layers are not strictly isolated. For example, a multi-modal service residing at the User front-end can render a video stream originating from the Logical sensors and actuators at the bottom of the Ice Cube.

In the following we describe the features of the agent community that implements the User Services layers of the Ice Cube.

### 3.2 Intelligent messaging

As proposed in the FIPA Abstract Architecture Specification [18], the information exchange between agents is based upon a well-defined communication ontology, in order to ensure that the semantic content of tokens is preserved across agents. The importance of such a common semantic concept is increased by the distributed development of the various services and the necessity of the service developers to understand each other correctly.

The CHIL Communication Ontology is part of the overall CHIL domain ontology and, like this, completely defined using the Web Ontology Language OWL [36]. OWL combines well proven Web technologies with state-of-the-art Description Logics and is available in three flavors. In CHIL, OWL DL (OWL with decidable Description Logics) is deployed for modeling the CHIL domain of discourse. The CHIL domain ontology is split into separate Web resources and can be composed on demand from smaller units using OWL's import mechanism. It comprises several modules that are physically represented by Web resources with distinct URLs. The idea of modularization is that software developers only need to reference those parts of the ontology that are relevant for them. Additionally, modularization increases performance and has been beneficial when deploying the agent communication subset of the ontology. The CHIL ontology consists of the following modules:

- *chil.rdf* is the main file of the CHIL Ontology. It imports all the domain specific modules and thus can be considered as a kind of main setup file.
- *chil-core.rdf* contains the axiomatic knowledge shared by many architectural layers of CHIL that are common to all smart room installations. It can be considered as the core of the CHIL ontology.
- *chil-pc.rdf* contains the descriptions of all perceptual components in CHIL. It consists of a generic perceptual component model, a set of standard categories of perceptual components, and finally descriptions of all vendor-specific components.
- *chil-ca.rdf* contains the agent communication part of the ontology.

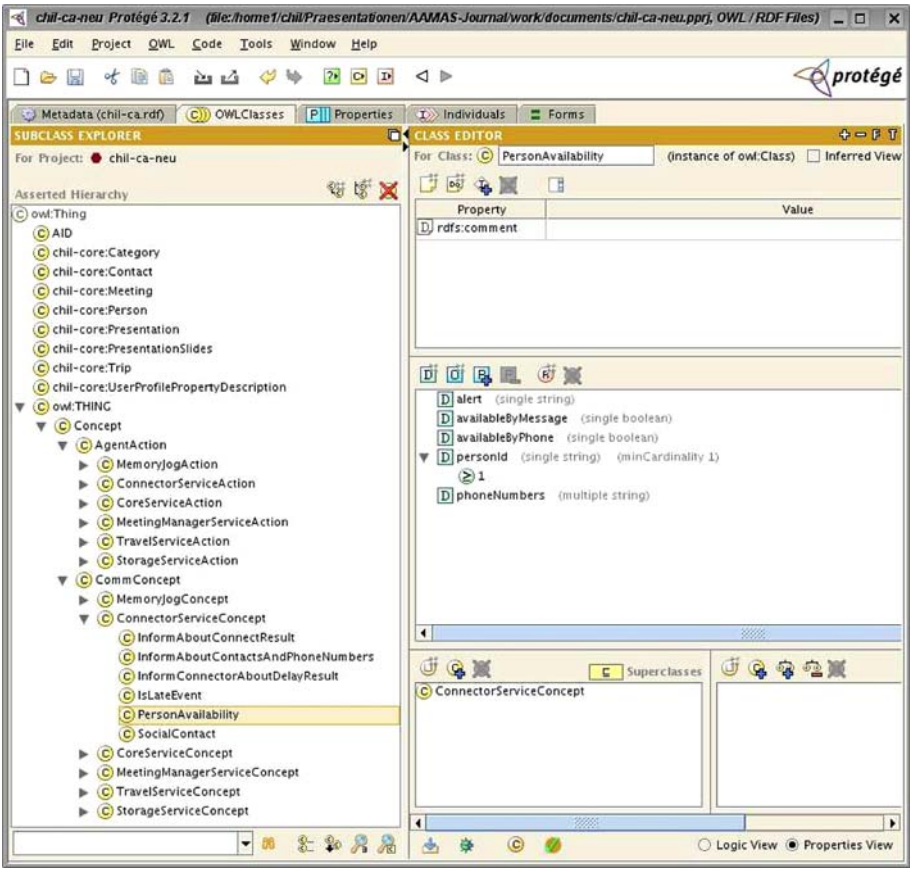



**Fig. 3** The Protégé tool for managing and visualizing ontologies, showing a section of the OWL-based CHIL ontology structure

– *chil-isl.rdf* finally gives an example of the factual knowledge of an actual CHIL installation.

As part of this modular concept, the CHIL communication ontology is fully integrated in the CHIL domain ontology. On the one hand, it uses concepts of the core ontology whenever artifacts that are part of the common domain are required in answers to requests. On the other hand, it extends the core ontology by tokens, which are specific to agent communication and not defined in the core ontology, particularly agent actions for requesting services and specialized result classes. Simultaneously, the communication ontology is based upon the *Simple JADE Abstract Ontology*, an elementary ontology provided by JADE as a basis for all ontological message exchange. New communication concepts therefore are subclasses of the JADE entries *Concept* and *AgentAction*.

This conceptual design can be seen in Fig. 3, which shows a part of the CHIL communication ontology visualized by the Protégé tool. The OWL class tree first lists the concepts, which are part of the core ontology and which are used as content of result messages in the agent communication, e.g. *Meeting*, *Person* and *Contact*. The remaining part of the class tree illustrates that the CHIL communication ontology can be seen as the entirety of all service

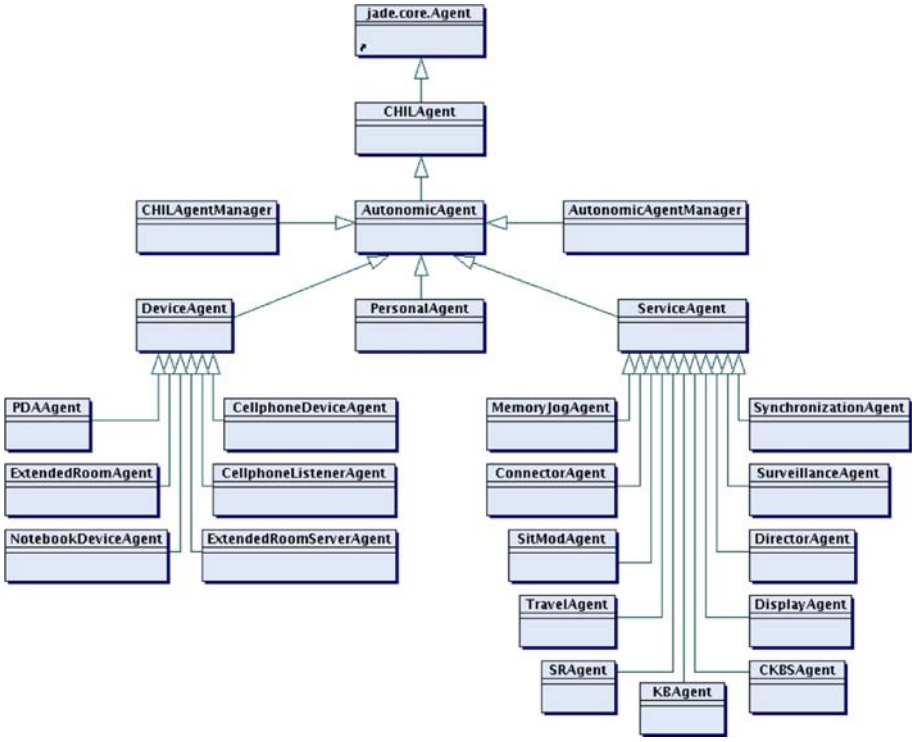



**Fig. 4** The agent structure of the CHIL agent tier

specific ontologies. Each service may define its own ontology for the intra- and inter-service communication, which is plugged in the appropriate agents at startup (see Sect. 3.5). The service ontologies consist of agent actions used as requests to agents and common concepts used as result notifications to the original requester. Figure 3 represents this structure by grouping the ontology entries by the appropriate services and the kind of entry.

The representation also points out one particular example of the Connector Service, a context-aware service that handles connection requests between people (cf. Sect. 4.3), that is representative for all entries: *PersonAvailability* is the answer to an agent action, which serves as a user request to the CHIL system in order to receive up-to-date availability information about a participant. As result, the user receives an instance of *PersonAvailability* indicating how the requested person will be alerted (quiet, mute, normal, loud), whether he is available by message and/or phone, the person identificator and his phone numbers. All received information depends on the current location and status of the requested participant (in meeting, in smartroom, inside building, outside, in car, unknown) and the social relationship between the two parties (VIP, business, personal, unknown) and, of course, subject to personal preferences and privacy issues (cf. Sect. 3.9).

Based on the CHIL communication ontology, the message handling of the agent framework consists of several parts. The *CHILAgent*, a basic abstract class for all agents (cf. Fig. 4 which illustrates our implementation structure) provides methods for creating, sending, receiving and decoding messages, which are strictly based on the semantic entries of the communication ontology. Furthermore, these methods, together with additional initiator and

responder classes for submitting and receiving messages, ensure that the agent communication is strictly compliant to the FIPA interaction protocols and communicative acts. Thus, the FIPA compliant messaging implementation of JADE has been elevated on a semantic level by enhancing and replacing the JADE methods with both FIPA compliant and ontology-based methods.

The message handling is completed by the *CHILAgentManager*. This agent is a central instance encapsulating and adding functionality to the JADE Directory Facilitator (DF), acting as a directory service, and coordinating the agent communication. Each agent behavior informs, on startup, the Agent Manager about the type of messages it can accept, and registers the agent's services, including required resources for carrying out that service. The *CHILAgentManager* acts, except for a few special cases, as a matchmaker, providing a requesting agent with handles to appropriate service agents capable of satisfying the request. Moreover, it ensures in cooperation with the *CHILAgent* that registration and deregistration of agents, modifications of agent and service descriptions, and the search for services is executed based on the communication ontology and fully conformant to the FIPA specifications.

### 3.3 Autonomy

Fault-tolerance has been discussed extensively in the literature, as in [17,31,57], where the authors focus on mechanisms for failure handling to ensure robust multi-agent systems. Also in Teamcore [52], every agent in an organization has an associated Teamcore proxy that records its membership in various teams and keeps track of active commitments made to these teams. The Teamcore proxies communicate with their corresponding agents to monitor the agents' ability to fulfill commitments and to inform the agents of changes to those commitments, but also amongst themselves so as to ensure coherent execution of team plans and the robust achievement of joint goals. Given this teamwork knowledge, the Teamwork proxies are able to detect anomalies in the execution path of a plan. In addition to failure handling, E-Elves [41,44] deal with the autonomic concept of agent systems, and in particular the issue of *adjustable autonomy*, where each agent is able to determine the degree to which it acts autonomously, so as to improve its performance by not being dependent on the user or too intrusive in its human interaction, while learning the user's preferences.

Our approach in providing autonomous behavior of both the context-acquisition layer and the end-service delivery has been presented in [15]. In this section we will outline the structural components which provide these features, and will focus mostly on the flexibility and performance of the introduced mechanism.

The challenges that arise in context-aware human-centric services pertain to the user experience; essentially the fault tolerant scheme should introduce minimal distraction to the users and at the same time it should be autonomous enough to investigate other alternatives. For instance, as far as hardware failures are concerned, the context-acquisition components are able to detect that a sensor has failed by the data feed being idle and divert their input from another sensor using the knowledge base. On the other hand, a software agent failing should cause no problems during an interacting event between users and, if needed, to maintain its latest state at all times. Such issues make the adoption of such services a very challenging task; however, the CHIL architecture has managed to deliver a high level of quality to the supported services.

For the purpose of achieving these goals, we adopted two widely used libraries:

– Hibernate [22], which is an object/relational persistence and query language for Java, and
– NIST Smart Flow [50], which is a specialized middleware that generates stream flows that wrap around the raw sensor data, enabling many processing components to subscribe to a sensor's output.

Hibernate is a highly flexible library which does not introduce additional overhead in the design or implementation phase of the development cycle. It is fully customizable using external files and allows for any type of object to persist. Moreover, it allows for defining any constraints present in the relational model (i.e. uniqueness, null values etc.), entity integrity as well as referential integrity. Finally, the persistent classes are not required to implement a specific interface nor extend a persistent super class, making Hibernate's persistent classes reusable in different contexts and persistence approaches.

Hibernate was selected over a number of persistence frameworks such as Torque [54], iBATIS [24], or TopLink [53]. Our main interests were performance, portability across different relational databases and to provide a fully object-relational mapping (ORM), without requiring a significant adaptation of the existing code, for instance by extending different classes. Hibernate is one of the leading frameworks in this area providing a fully functional and high-performance object/relational persistence.

The NIST Smart Flow is a highly flexible middleware which undertakes the distributed high-performance transfer of a stream (e.g. video or audio) to any computer within the smart space, which facilitates decentralized processing of sensor streams by multiple consumers. Distributed processing is essential for two main reasons:

– To allow more than one algorithm to leverage a particular sensor stream. For instance, a video stream may be needed by a body tracking algorithm and a visual personal identification method.
– To distribute the processing load to more than one computer. This is particularly important in the case of computationally demanding real-time multimedia processing, as our scenarios.

To the best of our knowledge, Smart Flow is the most prominent middleware in this area, and has been used in a number of large projects with great success.

### *3.3.1 Service autonomy*

In our implementation some dedicated agents play a very specialized role. The self-healing control is managed by two of the core agents, the *AutonomicAgent* which is the base class for all autonomic agents in our framework, and the *AutonomicAgentManager*. Exploiting the benefits of the JADE Framework for inter-agent communication, the *AutonomicAgentManager* is at regular intervals querying the Agent Management System of JADE for the status of specific agents. The result of this request reflects the current status of all registered agents participating in the framework. In the case of a dead agent, the *AutonomicAgentManager* initiates the regeneration sequence. This sequence is dependent on the type of self-healing the agent has requested during registration. We consider two types of registration, which signify a different handling method: Stateless and stateful handling. The agents specify their registration type during their registration to the *AutonomicAgentManager* as soon as they boot up. In Sect. 5 and in Fig. 11, we illustrate the overhead which is introduced by our framework for both cases of autonomic handling (stateful and stateless).

In contrast to the Teamcore approach, no additional agents are needed to implement this fault tolerance scheme. All agents register to the *CHILAgentManager* and *AutonomicAgentManager* which is monitoring their status. However, the *AutonomicAgentManager* is only monitoring the agents' status and is not aware of the agents' capabilities to fulfill tasks, thus it cannot provide the features of the Teamcore agents.

### *3.3.2 Stateless handling*

Stateless handling implies that the agent does not maintain any state during its execution. The regeneration sequence for this type of agent is a straightforward procedure based on restarting the agent, if necessary on a different platform. The agent registers again and participates in the framework by processing incoming messages. We follow this approach in the cases when agents are assigned to control actuating devices, such as the Targeted Audio Device [34,35], or projectors.

The reason is that such agents pend on incoming requests and do not maintain a state during the execution, as they actually act as message receptors. To clarify this, when a request arrives, the agent attempts to control the actuating service that was requested. If this was successful, then it sends back an acknowledgment informing the sender about the outcome of its request. If this acknowledgment is not received after a period of time, due to the fact that the agent died, the agent that made the initial request, sends the same message again. While this approach, in the worst case of cascading crashes, allows for repeated requests to an actuator controlling agent, the situation can be overcome, if initiating agents give up after a number of failed attempts. Similar actions are taken when an agent sends a request to an actuating agent, which has either failed or is in the process of being restarted.

### *3.3.3 Stateful handling*

In the case when an agent needs a state to be maintained during its execution, the *AutonomicAgent* behaves differently. At regular intervals, each *AutonomicAgent* stores a serialized object in a database together with a timestamp. If the agent dies and the *AutonomicAgentManager* is aware of this fact, it attempts to restart the corresponding agent, which in turn reads the last entry of its serialized object from the database, using Hibernate. The agent retrieves this entry which reflects its last known state, and attempts to re-initialize its internal variables. As soon as the old state is reloaded, the agent is able to participate again in the framework. The process of registering to the *AutonomicAgentManager* is repeated as before.

### *3.3.4 Tolerating hardware failures*

To be able to tolerate hardware failures, we make use of the NIST Smart Flow middleware. The processing component is able to determine whether the input stream is arriving or not, as it constantly polls the Smart Flow interface for new data. If the output stream is not available for some time, then the processing remains idle, and if possible, the algorithm is diverted to another input. This method has been used in the case of the Intelligent Meeting Recording service [2], which selects the best video stream to capture the speaker. To illustrate this, we provide the following example. Every camera that exists in a smart space is controlled by a camera driver. Each camera driver, using the Smart Flow middleware transmits its captured streams to the services which have subscribed to receive this type of input. Should a camera fail, the camera driver has no new data to send, thus the flows that are generated by the middleware are not transmitted. The receiving service processes all incoming streams as they

arrive and since one of them is not generating anything, the service becomes aware of the failure and disregards it. If the camera is restarted, then the camera streams are captured and transmitted using the middleware, making it transparent to the processing components.

While this feature is not implemented as an agent, its configuration can be managed by a dedicated agent which controls and supervises the data flow. In collaboration with other middleware interfaces (such as the IBM CHiLiX library [29]), the agent can divert the input and output data to the appropriate receptors.

### 3.4 Pluggable behaviors

In order to allow the distributed development of services and to facilitate their integration and configuration, a plug-in mechanism has been designed that enables the implementation of service specific code in pluggable handlers by keeping the agent service-independent. In the view of this plug-in mechanism, an agent of the CHIL system consists of three main parts:

1. the agent itself,
2. the agent's behaviors,
3. and the XML based configuration files.

The agent itself should contain only the common methods and attributes all concerned partners have agreed on. This will result in a stable agent. All other, service specific code should be inside the behaviors, which have to implement certain interfaces to be pluggable into a CHIL agent. Finally, the XML configuration files should contain a specification of the required behaviors, including their type and class.

At start-up time the agent looks into its configuration files to determine which behaviors have to be instantiated and added to the agent's behavior queue. Since the necessary code for this mechanism is concentrated in the basic *CHILAgent* and its helper classes *ConfigurationSupport* and *PluginSupport*, the source of the configuration data can be switched very easily; e.g. it is possible to get this data from a knowledge base instead of the XML file, by using the *CKBSAgent* which interfaces with the knowledge base. Furthermore, the plug-in mechanism is available to all agents in CHIL derived from *CHILAgent* without extra work for the agent developers.

Three types of pluggable handlers are considered to be necessary, namely

- *Responders* are added to the agent's behavior queue and react on incoming ACL (Agent Communication Language) messages sent by other agents.
- *EventHandlers* are put into a map along with the event types they are registered for. They are triggered by events from outside the agent's world, e.g. the user's GUI, a perceptual component, the situation model or a web service.
- *SetupBehaviors* are responsible for service specific initialization. They will be executed in the setup phase of an agent since they do not have to wait for a trigger to be activated.

Figure 5 illustrates the relationship between the concrete implementation of pluggable handlers and agents and their respective base classes.

Configuration of the agents' behaviors is done manually by the respective service developer or a CHIL system administrator. The described plug-in mechanism does not handle contradictions in the configuration files. An automated detection and resolution of conflicting behaviors would require a more formal description of the agents' tasks and further research.

A similar mechanism for pluggable behaviors can be addressed by BDI-based languages as in platforms we mentioned in our related work section, and can actually perform better in comparison with event handling mechanisms that we propose in this section when used in

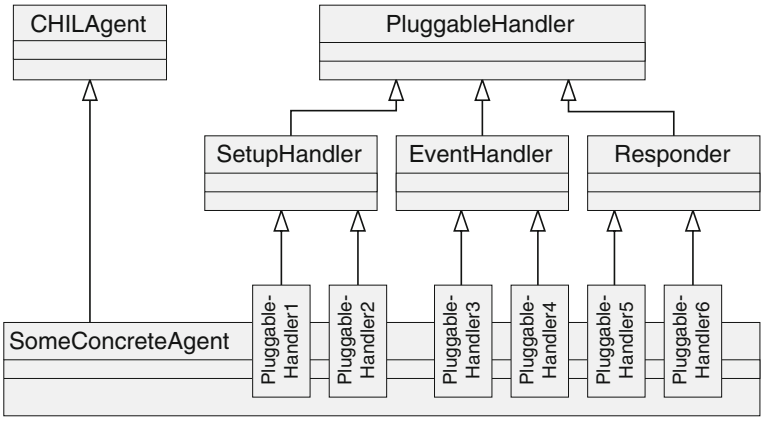



**Fig. 5** The use of pluggable handlers

interfacing with sensors or actuators. However, in the scenarios that we focus on, the actual sensor and actuator control is not guaranteed to be made by agents, or even to be implemented by a specific vendor. These agents are actually a "bridge" between the CHIL Agent Framework and the sensor/actuator layer of our architecture. Agents receive events from the agent framework, and interpret them in device controlling commands using IBM's CHiLiX library [29], a high performance library which has been developed in the scope of the CHIL project, and allows the controlling of either context acquisition components or device controllers.

### 3.5 Pluggable services

The Pluggable Behaviors mechanism allows service providers to create new service functionality and plug it into multiple agents without the need for recompilation. Since new functions inherently need new communication contents, a service must have the option to define, use and communicate new messages, which can be understood and compiled by all participating agents. Furthermore, to fully exploit the features of the CHIL services as a whole, a service must be able to communicate and cooperate not only with multiple agents, but also with multiple services. Using a global communication ontology for all services would foil every plug-in mechanism and distributed development; service providers would have to agree on a common ontology with all other parties before starting to realize new service functions.

As a consequence, the agent framework on the one hand had to provide a mechanism that allows service providers to define their own service specific communication ontology, and on the other hand enable agents to participate in multi-service communication, handle messages from various services and work with several ontologies. Hence, services and their communication ontologies had to be handled similar to Pluggable Behaviors: the framework should provide a mechanism to plug them in without the need for recompiling the agent's code. To this end, the Pluggable Behaviors mechanism has been extended to *Pluggable Services* by realizing a plug-in technique for services and service specific communication ontologies.

A service is integrated into the CHIL system by means of an (XML based) configuration file. This file specifies the service name, the participating agents and their pluggable handlers, and the service specific communication ontology (see Fig. 6). Each pluggable handler is defined by its type (setup, event or responder) and its class name. A priority value assigned to each handler can be used to determine the order of execution in case of competing behaviors.

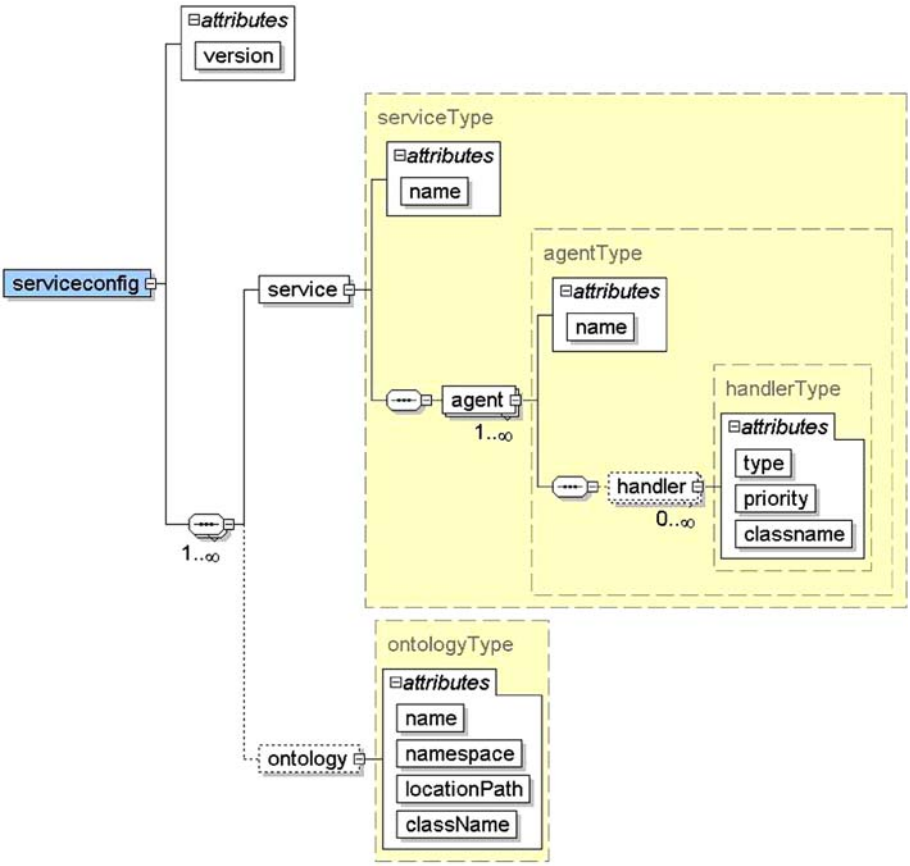



**Fig. 6** Service configuration XML schema

The service ontology is specified by its name, its namespace, and the location and name of the ontology class. During the startup phase, each agent parses the service configuration files and determines the pluggable handlers and the service ontologies, which are dedicated to it. The handlers are registered as described in Sect. 3.4, the service specific ontologies are integrated into the agent's ontology management by the ontology merging mechanism of JADE.

Furthermore, the configuration file provides an additional feature to system developers and administrators: it allows enabling/disabling of certain functionality by simply adding/removing the appropriate elements in the configuration file without having to recompile the source code. In a similar way, the services are declared to the CHIL system: A master configuration file (based on XML as well) lists all participating services and the names of their configuration files. Again, enabling and disabling complete services can easily be performed by adding or removing the appropriate configuration elements.

### 3.6 Device independence

A major challenge in human-centric ubiquitous services, is that the service should be available to the end-users regardless of their location and time. Servicing mobile users is a difficult

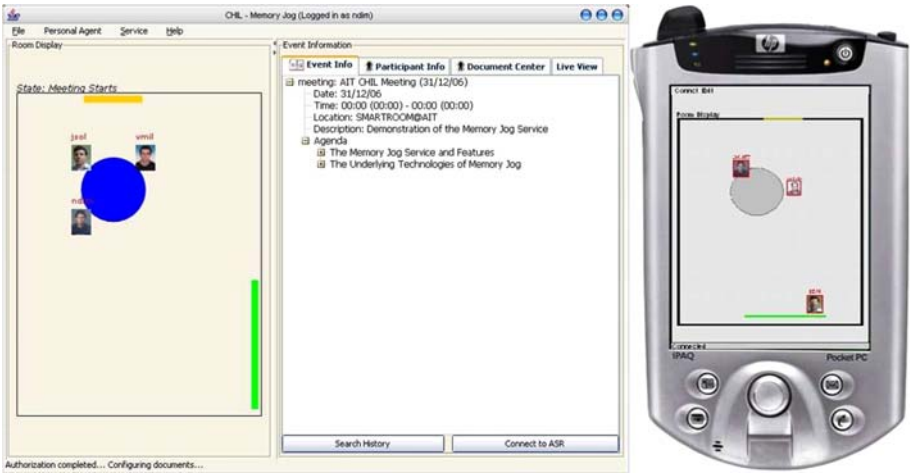



**Fig. 7** The Memory Jog Service Graphical User Interface in the cases of a desktop computer and a PDA device

task as they should at all times enjoy the same quality of services as those who are located within the smart space, equipped with high-end systems. In addition to that, device limitations should be taken under consideration, thus requiring some effort in adjusting the communication channels and visual information.

Our implementation caters for different types of devices. Each device is modeled by a *DeviceAgent* as seen in Fig. 4. The main *DeviceAgent* is the *NotebookDeviceAgent* which is used in the case of Laptops and Desktop computers. In addition to this *DeviceAgent*, others have been designed such as the *CellphoneDeviceAgent* and the *PDAAgent*, which handle requests for cellphone devices and PDAs, respectively.

In the case of the *NotebookDeviceAgent*, a *DeviceDesktop* has been designed which visualizes the information that the user initiates or that the service is providing. This graphical user interface is directly connected to the *NotebookDeviceAgent*, which also communicates with the users' *PersonalAgent* for personalization filtering, a feature presented in Sect. 3.9, and coordinates requests from the user. In the case of the *PDAAgent*, the *DeviceDesktop* is more compact and simple, constraints which arise due to computing power and screen issues. In Fig. 7, we show the graphical user interface of one of the services that has been developed with this framework, both in the case of a desktop computer and a PDA.

The *PDAAgent* interacts with the rest of the smart space using a serialized form of an ACLMessage, to avoid having to cope with different message formulation types. On the receiving end, an agent which receives the messages from the PDAs, and in general any wireless device, de-serializes the message and sends it to the appropriate agent, acting as a gateway between the devices and the actual smart space [14].

### 3.7 Directory service

Sophisticated context-aware services require the presence of a directory service mechanism for registration and later lookup. While standard approaches are able to manage large quantities of information, they come with limitations when handling components that can insert or retrieve information. Furthermore, in large-scale human-centric context-aware services with

numerous types of sensors, actuators and other components, information modeling becomes a challenge as, in addition to maintain information about each component individually, the directory service should be able to respond to all types of queries about the available components in the smart room; i.e. from determining which services are operating in the smart room to what type of sensorial sources each component has subscribed to. In our implementation we have developed a directory service leveraging a knowledge base which serves requests and handles registration of any component in the architectural framework [37].

The managing agents are the *CKBSAgent* and *KBAgent*, which wrap around the knowledge base and database, respectively. The two agents interpret agent requests into ontology and SQL queries and appropriately divert the request to the corresponding service. In the case of the database, the resulting reply is compiled to be a member of the communication ontology, described in Sect. 3.2. This enables the bundling of the information to be transmitted as a single ontology class, such as 'Camera', which contains information about the camera type, the location of the camera, etc. While these types of queries are more frequently used, our knowledge base mechanism is also able to handle queries that are vague. For instance, a service may be interested in getting the list of "all cameras that are facing the door", or "a panoramic camera", etc. The *CKBSAgent* interfaces with the knowledge base service, and the latter determines the answer depending on the information at hand.

While several approaches to directory services middleware exist, this approach has significant advantages over conventional technologies such as UPnP (Universal Plug 'n' Play), SLP (Service Location Protocol) and UDDI (Universal Description, Discovery and Integration), as these are not particularly tailored to the range of information and components needed in such a ubiquitous computing environment. For instance UDDI and SLP are service-oriented approaches, while UPnP is clearly device-oriented. Our approach introduces an intelligence flavour in this process, as the knowledge base service is able to infer information from existing sets of meta-data according to the current context. The knowledge base service is able to provide directory services, not only for the available devices, but also for available services, perceptual components and others.

The presence of a global intelligent database assists greatly in enabling the architecture to handle remote participants by not limiting them to use just portable devices such as a PDA, a feature described in the previous paragraph. In the following section, we illustrate a feature, which is based on this directory service and interconnects remote spaces.

### 3.8 Satellite spaces

One of the key problems that ubiquitous systems face is the issue of scalability in terms of expanding their reachability to more users. Users can be both mobile or not. While we addressed mobile users in a previous paragraph, we also place significant effort in servicing users who are not in the smart space but in a remote location, having a smaller set of sensing equipment, i.e. a webcam and a microphone. The issue in this case is to manage and deliver a high quality service to this kind of users, by making them feel as active participating members of the event currently taking place in the smart room. We consider that these users are located in light-weight smart spaces, which we call "satellite spaces". To deal with this problem we devised a mechanism which, by interfacing with the directory service, is able to "glue" two or more smart spaces together enabling the remote participants to interact with the people currently present in the main smart space.

The basic functionality lies in the mapping between communication ontology members to XML messages, which are encoded on the "server" side and decoded on the "client" side. All

messages are encoded and transmitted to the remote user. This process involves querying the database and the knowledge base, which reply by providing the active "smart space servers" and enable the communication between the personal agents of both the local and remote participants.

The satellite spaces heavily use the fault-tolerant Intelligent Meeting Recording service [2]. The reason is that any user that is located in a remote space other than the main smart space, requires a video stream to be sent to his location. In general, a smart space consists of a large number of camera sensors operating at the same time, each of them covering only a part of the room itself. The user thus would have to manually decide which stream to view at all times, which would require not only his constant intervention, but also to receive large amounts of data coming from all the cameras. Our Intelligent Meeting Recording is able to act as an "intelligent director" by selecting the optimal stream and transmitting it to the remote users, reducing the amount of unnecessary information that is transmitted. The Intelligent Meeting Recording selects at all times the video stream with the most frontal view of the speaker and considers this as the primary video stream for storing and transmitting it to remote users.

### 3.9 Personalization

In a CHIL environment computer assistants 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. Each CHIL user is described by a user profile, which contains a set of personal attributes including administrative data relevant to the CHIL system (e.g. access rights, user capabilities and characteristics) and individual personality information like professional and personal interests, contacts and the social relationships between the contacts and the user (e.g. VIP, business or personal) as well as interaction, device and notification preferences such as notebook, PDA, cell phone call, SMS, MMS, targeted audio, etc. The user profile also contains information like how a user wants to be notified about a phone call depending on his current activity and his relationship to the caller. These personal settings allow the CHIL system and its context-aware services to interact with and to assist its users taking into account everyone's individual needs and preferences.

The administrative part of the user profile is maintained by the system administrator; personal data can be added and modified by the user exclusively by means of a GUI. Access to and control of the user profile is managed by the user's *PersonalAgent*. Thus, the *PersonalAgent* not only operates as a personal assistant, but also as a privacy guard to both sensitive and public user data. Since the *PersonalAgent* is the only instance having access to the user's profile, it ensures user data privacy.

The *PersonalAgent* is assigned to a user during the login procedure. It controls, via a dedicated *DeviceAgent*, the complete interaction between its user and the CHIL system: it knows what front-end devices its master has access to, how it can best receive or send information to and from its master and what input and notification types its master prefers. Furthermore, the *PersonalAgent* has access to the Situation Model through the *SituationWatchingAgent*. Supporting both requests and subscriptions the *SituationWatchingAgent* can permanently update a *PersonalAgent* about its master's current context (location, activity, state of the environment) and the availability of the various devices in a dynamically changing situation. Based on the static data of the user profile and the dynamic context information, the *PersonalAgent*

handles user input and connection and notification requests to its master in the best way and with the most appropriate media possible.

The Connector Service described in Sect. 4.3, for instance, adaptively handles incoming calls on a cell phone based on the callee's user profile and current situation. In his profile a user can specify if a call or notification should be put through or be intercepted by the Connector Service depending on both the social relationship between caller and callee and different groups of activities. For example, a person in the CHIL environment might opt for accepting calls from a VIP (e.g. his boss) but declining calls from a friend while in a meeting-like situation.

### 3.10 Qualitative advantages of the CHIL agent framework

The benefits of the CHIL Agent Framework are manifold. On the one hand, the architecture undertakes a wide range of tedious tasks, easing the deployment of new services, and on the other hand it provides a transparent layer to the developer in terms of information retrieval. It offers high flexibility, scalability and reusability and it facilitates the integration of components at different levels, like

- Services,
- Perceptual Components,
- Sensors and Actuators, and
- User Interfaces.

Particularly the plug-in mechanism for agent behaviors, described in Sect. 3.4, constitutes a powerful technique for the development, test, integration, configuration and deployment of new services and components. Developers may create new agents and behaviors and use this mechanism for easy behavior integration and agent configuration, thus facilitating and accelerating the process of development and testing. They may benefit from the reusability feature of the agent framework by including own behaviors in existing agents in order to use the functionality of these agents. And they may profit from the flexible configuration facility, allocate behaviors to different agents, turn behaviors on and off and even turn complete services on and off. This mechanism not only provides a facility for the implementation of services in handlers by keeping the agents service-independent, but also facilitates the integration of heterogenous components and ubiquitous services, and establishes a basis for a generic architecture which can be used across developers of different services, all of them major goals which have been addressed in the introduction of Sect. 3. Additionally, detailed guidelines for service integrators are available, which help service developers to integrate their services into the CHIL architecture framework.

Another important quality factor is the use of an ontology based agent communication, described in Sect. 3.2. Elevating the collaboration of components to a semantic level not only augments the robustness of the system in terms of mutual understanding of internal components, but also reduces the error-proneness of integrating new components and enhances the interoperability with external systems in a significant manner. This approach both augments the integration of new services and devices and meets the goal of co-existing services interfacing with each other, as addressed in Sect. 3.

One of the major quality objectives was to provide human-centric services interfacing with humans in an unobtrusive way. This goal has been achieved by the concept of personal agents acting as personal butlers, which not only react on explicit user requests, but also observe human needs and proactively provide appropriate services, based on user

profiles which contain administrative and personal data. This has been described in Sect. 3.9. Furthermore, suitability and user friendliness of provided services are enhanced by the facility of using different devices like notebooks, PDAs and smartphones, realized by special device agents which closely communicate with the user's personal agent, as described in Sect. 3.6.

Tolerating both service and hardware failures, as indicated in the introduction of Sect. 3, is performed by stateless and stateful agent handling based on the object-relational persistence service *Hibernate* and the *NIST Smart Flow* middleware, described in Sect. 3.3. The quality of the CHIL Agent Framework is also improved by a centralized information repository and the interconnection of light-weight smart spaces to the main smart space, both issues adressed in the introduction of Sect. 3. The former one is realized by a knowledge base, which is managed by two specialized agents. It services requests and handles registration of any component as described in Sect. 3.7. The latter one has been implemented as smart satellite spaces and has been described in Sect. 3.8.

As a manifestation of the fact that this framework facilitates the development and deployment of services, we present in the following section four user-centric services, all of which have been developed using this framework.

## 4 Services

The architectural framework has been utilized for implementing several non-intrusive applications. In the following, four example services are introduced, namely the Memory Jog Service, the Surveillance Service, the Connector Service and the Travel Service. Those prototype services were developed by individual teams at different locations with diverse system set-ups.

### 4.1 The Memory Jog

The Memory Jog Service is a pervasive application which aims at providing non-obtrusive assistance to users in smart spaces, during events such as meetings, lectures or presentations, by supporting participant-participant interaction [39], while observing their functional roles [3]. It exploits the whole range of the underlying sensor, actuator and context-acquisition infrastructure to provide the required assistance to the participants. It provides preferred features for events, such as event tracking, intelligent meeting recording [2], private messaging using text or audio means, person and speaker tracking, intelligent display of information as well as providing a summary of events during a person's absence. All these features are adequately 'packaged' in a flexible and well-designed Graphical User Interface (Fig. 7), which can be adapted to the user's terminal equipment, e.g. a desktop computer or a PDA device. Moreover, these services are transparent, without intruding and interrupting the event and creating distractions. The Memory Jog Service is designed as an agent-member of the agent tier exploiting the benefits of our multi-agent structure. Its features, which are either initiated by the user, or are triggered non-intrusively and manifested during the evolution of the event are:

– **Agenda tracking:** The Memory Jog follows the meeting agenda during the evolution of the event, provides information regarding the presenter and the subject under discussion, and enables the user to search for past similar discussions.

- **Intelligent meeting recording:** The Memory Jog can determine the optimal view of a speaker and store this video stream instead of storing the whole amount of video streams from all cameras. Moreover, this stream is forwarded to the external participants which can view a live view of the actual event.
- **Remote participation:** The Memory Jog fully exploits the capabilities of the software architecture for remote and mobile participants. Users can interact and participate in the event by logging in either using their PDAs, Smartphones or by their Laptops using a simple webcam and a microphone.
- **Database/knowledge base search:** By using the Memory Jog, the user can query the databases and knowledge bases which accompany the Memory Jog Service and get information regarding a user's biography, the agenda of previous meetings, etc.
- **Private messaging (audio, text):** The graphical user interface can significantly assist the user in the cases when he/she wants to send a private message to individuals in the room. Moreover, external participants can use the Targeted Audio device [35] which can deliver highly targeted beams of sound to any location of the smart space.
- **Summary of missed events:** The graphical user interface provides the user in a non-obtrusive manner with a summary of contextual events which he may have missed during a period of absence. This information is visualized to the user as a timeline of events which show the alteration of events during this period. The user can later see the video of these events, as it will be stored using the Intelligent Meeting Recording feature of the Memory Jog, which was described previously.

### 4.2 The surveillance service

A smaller-scale ubiquitous service exploiting the benefits of this software architecture, is the Surveillance Service. As before, this service is managed by a dedicated agent-member of the agent society which is gathering all the context originating from the underlying set of context-acquisition feeds. The result is processed and is presented graphically in a well designed GUI.

The purpose of the Surveillance Service is to monitor in-door environments about human activity and trigger alarms based on a set of rules which signify a 'hot zone'. In our implementation, we assume that unidentified individuals are not allowed to be in a specific location. Tracking individuals is done by the body tracker component, whereas the person identification is done by the corresponding component. Each action is logged to a database for later replaying but it can be used in real-time as well.

The Surveillance Service is a good example of exploiting the benefits of an autonomic system. The service is responsible for coordinating a number of different camera sensors and, according to their status, to make intelligent decisions based on the events that take place in the smart space, e.g. to call security because there is a person near a hot spot. However, this is only one part of the service logic, as these decisions and events that have been identified, are sent to other agents as well, in our case to database controlling agents for logging. Should a camera fail, the service bases its decisions on the other cameras, disregarding the output of the failed one, which may introduce errors in judgment. As these decisions have been made, they are forwarded to the database agent which logs them to a relational database. Should this agent fail, the service will become aware of that as it did not receive an acknowledgment for the latest request. It stores all events and decisions and waits for the agent to regenerate. As soon as the agent is operating again, it informs the service about its status, and the process is repeated.

### 4.3 The connector service

The Connector Service addresses a particular type of social situation between two (or more) people by means of communication devices. Crucial issues in this context are the availability of the callee, and the appropriateness of the time and place for the call to be accepted. What if a rejected phone call was of unexpected importance? What if the call is closely related to the current task? At the same time, on the callers side, it can become quite annoying to lose time playing phone tag trying to reach others, asking third persons about the whereabouts of the original contactee, and not being available oneself for returning calls [38].

The Connector Service is a context-aware application that intelligently connects people. It manages communication links (connections and notifications) and handles connection and notification requests, either simple requests (e.g. a meeting participant wants to talk to another) or more complex requests (e.g. a meeting participant wants to notify all the other participants). It maintains an awareness of its users' activities, preoccupations and social relationships and mediates a proper connection at the right time between them. It adapts the behavior of the contactee's device automatically in order to avoid inappropriate interruptions. Moreover, the Connector Service can use any available output device to deliver information to users in the most unobtrusive way possible.

The corresponding *ConnectorAgent* mediates between various *PersonalAgents* and handles communications affecting two or more *PersonalAgents* (*PersonalAgents* may communicate with each other directly if there is only a direct communication between two *PersonalAgents*). This includes that the Connector Service stores all pending connections and finds a suitable point of time for these connections to take place.

In order to describe the complex behavior of the Connector Service, we consider a sample scenario described in detail in [9]. In this scenario a meeting is scheduled in a CHIL smart room, where most of the meeting participants are already present. Another participant (Jeff) realizes that he will be late for the meeting and wants to inform the other participants of his delay. With the help of the Connector Service Jeff is able to reach all persons affected by his delay with only one virtual call.

Jeff uses his smartphone (Fig. 8, left) to trigger his *PersonalAgent* (via the smartphone's *DeviceAgent* representing his phone) which informs the *AgentManager* of the delay. The *AgentManager* asks the *MeetingAgent* for details about the meeting, e.g. the room where the meeting is held and a list of all scheduled meeting participants. With this information the *AgentManager* requests the *ConnectorAgent* to deliver Jeff's message to the attendees. Instead of contacting all attendees individually, the *ConnectorAgent* retrieves a list of participants already present in the meeting room in order to notify them all at once. Information about the current attendance is kept in the situation model and is accessible through the *SituationWatchingAgent*. Notification of the present attendees is done via the *SmartRoomAgent*, which is aware of the output devices available in the meeting room, e.g. central video projector, loud speakers and targeted audio. Since the meeting has not started yet, the *SmartRoomAgent* chooses to display Jeff's delay message via the central video projector. For those participants who have not yet arrived in the meeting room, the *ConnectorAgent* informs their *PersonalAgents* which in turn may choose an appropriate output medium for their masters (e.g. notebook, PDA, cell phone, etc.) according to their current situation and user profile. Finally, Jeff is informed by his *PersonalAgent* that his message was delivered to all meeting participants.

### 4.4 The travel service

The Travel Service is another example of a ubiquitous service realized by a member of the CHIL agent society. It assists travelers with planning and re-arranging their itineraries. This service can either be evoked directly by the user through one of his personal devices (cf. Fig. 8, right) or act proactively. An example scenario has been implemented which demonstrates close cooperation between the Connector Service and the Travel Service.

This scenario follows up the Connector Service scenario where one participant (Jeff) is late for a meeting. His presence is considered to be crucial for the outcome of the meeting. Hence, the beginning of the meeting is delayed until Jeff arrives. All meeting participants are known to the CHIL system and have CHIL-enabled personal devices, i.e. notebooks, PDAs or smart phones with a CHIL software client. Most of the participants have tight itineraries with flights or trains leaving shortly after the scheduled end of the meeting. Their planned itineraries are known to their respective *PersonalAgents*.

Triggered by Jeff's delay message each *PersonalAgent* determines whether the delay is likely to let its master miss his return connection. If this is the case the *PersonalAgent* providently initiates a search for alternative connections. It provides the *TravelAgent* with the necessary information including user preferences, e.g. if its master prefers to fly or take a train. The *TravelAgent* processes the request by retrieving information from semantic web services of railway operators, airlines and travel agencies. The current implementation uses


**Fig. 8** Smartphone client to access multiple CHIL services

simulated semantic web services due to a lack of suitable services. However, once such services become available they can be integrated without difficulty.

Eventually, the Travel Service sends a list of possible connections to the *PersonalAgent* which notifies its user. As with all CHIL services, notifications are done as unobtrusively as possible. The *PersonalAgent* chooses the most appropriate way to inform its master taking into account his current environmental situation (e.g. "in meeting"), the currently available output devices (i.e. personal devices like smart phones, PDAs, notebooks and output devices of the smart room, e.g. targeted audio or steerable video projector) and his preferred way of notification (e.g. pop-up box or voice message). The user can then decide whether he wants to change his itinerary.

A possible outcome of the search could also be that the *PersonalAgent* informs its master that he should leave the meeting as planned, since there was no suitable alternative itinerary. In case the CHIL user is not satisfied with any of the proposed itineraries and wants to look up travel connections himself, he can use his CHIL-enabled personal device to do so. Fig. 8, right, shows the query mask on a smart phone. An equivalent user front end is available for notebooks.

### 4.5 Summary

On the one hand, the described prototype services indicate the variety of possible applications supported by the CHIL Framework. It is not only capable of hosting individual services but also facilitates the collaboration of services on a semantic level as shown using the example of the Connector/Travel Service.

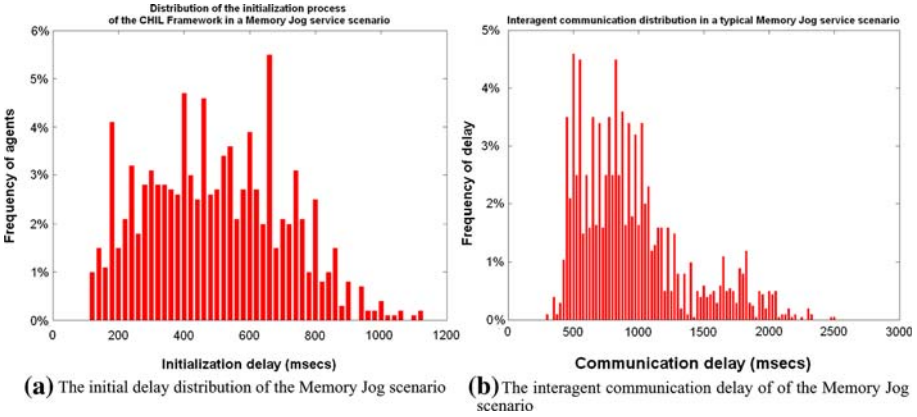
On the other hand, by implementing those services we have demonstrated the suitability of the agent framework for developing context-aware cooperating applications. Specifically, implementations have shown that it is capable of interfacing with the situation modeling logic, as well as invoking basic services. The full value of the framework is demonstrated by plugging multiple services into the same instance of the framework based on the same underlying infrastructure for sensing and context-awareness.

## 5 Performance evaluation

In this section we discuss the performance of the CHIL Agent Framework in terms of communication, processing overhead, and agent autonomy performance. These results were obtained during a series of executions of the Memory Jog scenario, which includes a wide range of agents with different capabilities and tasks, such as device controllers, database managers, sensor coordination agents etc. The setup of our smart space can be seen in [47], where we also show the configuration of our sensing and actuating equipment. Our network consisted of seven computers,[1] six cameras, four inverse-T shaped microphone arrays, one MarkIII linear microphone array, one projector and one targeted audio device, as well as one database and one knowledge base for information storing. The agents controlling each sensor and actuator were executed on the host to which the device was allocated, whereas the rest of the agents were spread across all available computers. Finally, the knowledge base repository was instantiated on a different computer than the one with the database.

Our initial investigation lies in the performance of this framework when it comes to providing context-aware services in smart spaces; the results are shown in Fig. 9. In Fig. 9a,

[1] All computers are identical dual Xeon at 2.8GHz, with 2.0 GB RAM.

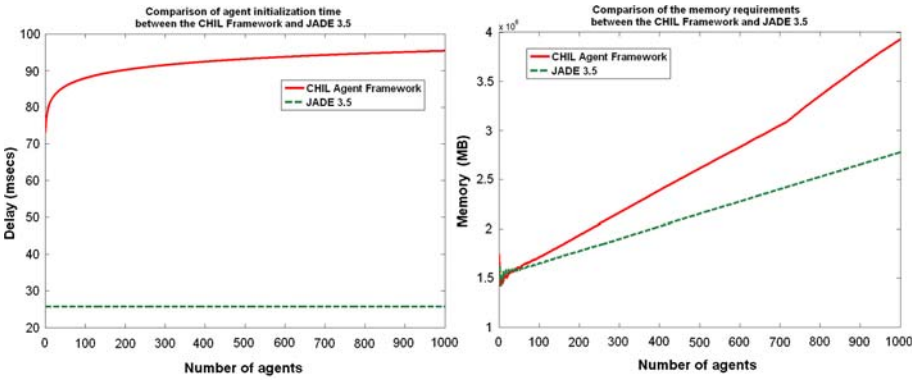



**(a)** The initial delay distribution of the Memory Jog scenario **(b)** The interagent communication delay of of the Memory Jog scenario

**Fig. 9** The agent booting delay, and the interagent communication delay in the CHIL Framework in the case of the Memory Jog scenario

we display the amount of time needed until an agent member of the CHIL society joins the framework. This initial delay is caused by the registration procedure to the *CHILAgentManager*, which acts as the main directory lookup service of the CHIL Framework, the possible queries that the agent asks for existing agent-members, and other initial steps which are specified by the service logic. These results were obtained by logging the amount of time needed by 721 agents (of different nature such as situation modeling agents, database agents etc.), throughout the scenarios covered by the Memory Jog service on various occasions. So these results reflect a very typical scenario of services that have been designed and implemented using this framework. In general, these results are satisfactory if we consider the fact that each agent in the Memory Jog scenario performs a large number of tasks. For instance this applies to agents which control cameras and set up each sensor prior to the commencement of the event, or interfacing with the knowledge base and retrieving information about the current smart space which is a highly time consuming process, etc. In these scenarios which involve human-centric services, such a performance has been experienced to be acceptable, in comparison with scenarios which would include very frequent agent interaction as for example in cases of simulations.

The communication overhead introduced by the CHIL Framework, as illustrated in Fig. 9b, is calculated from the time of the message transmission to the time that the receptor has identified the type of the message and responded with an acknowledgement. As in Fig. 9a, these results are also based on a typical run of the Memory Jog, a scenario which includes a wide variety of message content (simple text, ontology entries, lists of ontology entries etc.), and we consider it as a representative scenario to base our conclusions. The mean overhead delay stays below 1,000 ms; however, rare cases of delays which are even more than 2,000 ms have been recorded. Upon reception of a message, the agent decodes the message based on a list of message types that it accepts. This message, however, may contain additional data which needs to be decoded as well. These procedures do not require significant processing overhead. The results that are shown in Fig. 9a reflect the communication delay as experienced during the Memory Jog service, and should not be compared with the communication performance of JADE. For example, acknowledging a message could involve the triggering of a camera sensor to be activated, or the querying of a database for some information, etc, actions which are time-consuming. These results are to be considered as the communication delay



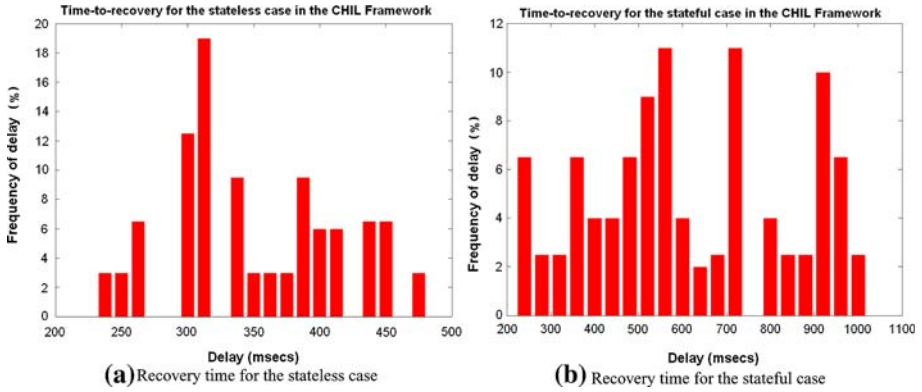

**(a)** Time comparison for the instantiation of 1000 JADE and CHIL Agents **(b)** Memory requirements comparison for the instantiation of 1000 JADE and CHIL Agents

**Fig. 10** Memory requirements and instantiation duration for the CHIL and JADE Frameworks

of the overall service, as they take into consideration all actions that the agents make upon reception of a message. However, the underlying communication mechanism is the same as in the JADE Framework, as we are not interfering with it. In the scope of the context-aware ubiquitous services that we explore, such a performance has proved to be quite effective in not being distractive to the users.

Despite the fact that the CHIL Agent Framework is significantly more complicated than its foundation framework, JADE, both the amount of time needed to instantiate 1,000 agents, and the required memory remains at a reasonable level, as seen in Fig. 10. To measure these, we developed a "dummy" agent, identical in both cases, extending the base class for each framework (i.e. the *CHILAgent* and the *Agent* for the CHIL and JADE Framework, respectively) and logged the amount of time until the registration was completed, and the memory requirements for each case.

The initialization time was measured from the moment that each agent requested the *CHILAgentManager* to register the agent, until the moment that the *CHILAgentManager* completed the registration. Regarding the memory consumption, this was measured using the `java.lang.Runtime.totalMemory()` and the `java.lang.Runtime.freeMemory()` methods and computing the difference between their results. The basic reason for the differences between the JADE and the CHIL agents is the fact that the basic JADE Agent performs no tasks during instantiation, but simply joins itself to the JADE Framework. In contrast to this approach, the CHIL agents perform a number of tasks which are both memory and time consuming. These tasks are finding the *CHILAgentManager*, initiating the registration sequence by communicating with the *CHILAgentManager*, setting up the individual logging preferences for the agent, specifying the CHIL specific ontology communication mechanisms and instantiate a behavior for handling CHILEvents (events which are posted by agents when an action is to be taken). The rate of the memory increase in the CHIL Framework case is considered to be associated with the large number of data structures that the *CHILAgentManager* maintains, which are not given a predefined size. As more agents are injected into the system, these data structures are adapted and take up more space in memory. This rate, though larger than the JADE case, still increases in a near linear fashion, at least this is seen until 1000 agents have been initialized. Finally, the *CHILAgentManager* agent on average, requires 600–700 ms to be instantiated, a time which is not too surprising as it




**(a)** Recovery time for the stateless case **(b)** Recovery time for the stateful case

**Fig. 11** Delay distribution for the recovery mechanisms as designed in the CHIL Agent Framework

undertakes additional tasks such as to take over the registration and deregistration process for each new agent, to assist in the searching for agents etc. This agent is only instantiated once.

Both memory and time requirements for the CHIL Agent Framework are higher than JADE's, something that is expected as the CHIL Agent is a significantly more complicated framework. In terms of the initialization process, this increases linearly in relation to the number of agents in the platform. Let us elaborate on these results, which were taken by an automated script that instantiated a JADE and CHIL agent every second. As the number of agents increases, the number of messages that are present in the same processor increase so the time to complete the registration is certainly affected. The memory requirements increase in the same manner up to a certain point (700–750 agents), and above that the rate is slightly increased. We claim that this is due to the recalibration of internal data structures such as hashmaps or arraylists, that have been assigned an initial value, but as the number of agents increases, these do increase as well but in a not standard fashion.

In the last figure (Fig. 11) we show the effect of agent restarting by the *AutonomicAgentManager* agent. This agent restarts a dead agent as soon as it realizes its death. The measured time is calculated from the moment of restarting until the moment of re-registering to the CHIL Framework as an active agent. This amount of time is highly related to the regeneration type that the corresponding agent has requested during registration, types which have been described in detail in Sect. 3.3. In the case of stateful handling, this is the amount of time until the last object is retrieved from the database.

## 6 Conclusions

In this paper we presented a multi-agent architecture for facilitating the design, development, management and maintenance of sophisticated context-aware ubiquitous computing services. We illustrated how this framework can coordinate a variety of sensor equipment and generate elementary context. These context cues can later be fused by complex situation modeling components, which are able to recognize complex situations that can be used to model human interaction.

The main benefits of the approach presented result from universal interfaces to different perceptual components (e.g. visual and audio trackers) and the scalability and reliability of

the architecture due to stable adding and removing of agents to or from the community of acting agents. The flood of data generated by a large amount of audio, video and other sensors (e. g. RFID) can be reduced to a manageable amount of situations and contexts to provide adequate input to different services.

We presented how different services are able to interoperate on a semantic level. The benefits of interoperation will become more visible as soon as a larger set of services is to be implemented. Useful services in this context could be an automatic minutes taking service (a rudimentary version has already been implemented), graphical seating arrangement display (dynamically changing during a meeting and seen from the perspective of each user in the scene) and coordinating services in a multi meeting scenario.

Our initial results are encouraging in terms of agent communication and failure handling, having an increased requirement in memory and initialization time as expected. However, these have been experienced in highly populated scenarios which are not under study. In addition to that, the CHIL Agent Framework is able to handle issues of hardware and software failures in a transparent fashion, introducing minimal distraction to the users during an interacting event. Furthermore, the option of plugging multiple services in the framework enables each service designer to focus on individual components rather than requiring knowledge of the whole software and hardware architecture.

In the future we will work on automatic code generation for behavior skeletons based on the agent actions and their associated concepts defined in the ontology. This will simplify the combination of different behaviors and will make it possible to check their consistency. Moreover, learning capabilities will be addressed to adapt the personal agent's reaction to individual needs and preferences for each user.

**Acknowledgments** This work is part of the FP6 CHIL project (FP6-506909), partially funded by the European Commission under the Information Society Technology (IST) program. The authors acknowledge valuable help and contributions from all partners of the project, especially of partners participating in WP2 which defined the software architecture of the project.

## References

1. Altmann, J., Gruber, F., Klug, L., Stockner, W., & Weippl, E. (2001). Using mobile agents in real world: A survey and evaluation of agent platforms. In *2nd international workshop on infrastructure for agents, MAS, and scalable MAS at the 5th international conference on autonomous agents* (pp. 33–39). New York: ACM Press.
2. Azodolmolky, S., Dimakis, N., Mylonakis, V., Souretis, G., Soldatos, J., Pnevmatikakis, A., et al. (2005). Middleware for in-door ambient intelligence: The PolyOmaton system. In *4th IFIP TC-6 networking conference, 2nd next generation networking middleware workshop (NGNM)* (pp. 33–39). Waterloo, Canada.
3. Benne, K., & Sheats, P. (1948). Functional roles of group members. *Journal of Social Issues, 4*, 41–49.
4. Bordini R. H., Hübner, J. F., & Vieira, R. (2005). Jason and the golden fleece of agent-oriented programming. In *Multi-agent programming* (pp. 3–37). US: Springer. doi:10.1007/0-387-26350-0_1.
5. CHIL: Computers in the Human Interaction Loop. At http://chil.server.de. FP6 IST-IP506909.
6. Coen M., Phillips B., Warshawsky N., Weisman L., Peters S., & Finin P. (1999). Meeting the computational needs of intelligent environments: The metaglue system. In *1st international workshop on managing interactions in smart environments* (pp. 201–212).
7. Cohen, P. R., Cheyer, A. J., Wang, M., & Baeg, S. C. (1994). An open agent architecture. In O. Etzioni (Ed.), *Proceedings of the AAAI spring symposium series on software agents* (pp. 1–8). Stanford, CA: American Association for Artificial Intelligence.
8. Crowley, J. L. (2003). Context driven observation of human activity. *Lecture Notes in Computer Science, 2875*, 101–118.
9. Danninger, M., Flaherty, G., Bernardin, K., Ekenel, H. K., Köhler, T., Malkin, R., et al. (2005). The connector: Facilitating context-aware communication. In *ICMI '05: Proceedings of the 7th international*

*conference on multimodal interfaces* (pp. 69–75). New York, NY: ACM Press. doi:10.1145/1088463.1088478.
10. Dastani, M. (2008). 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems, 16*(3), 214–248. doi:10.1007/s10458-008-9036-y.
11. Dertouzos, M. (1999). The future of computing. *Scientific American, 281*(2), 52–63.
12. Dey, A., Salber, D., & Abowd, G. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human Computer Interaction, 16*, 97–166.
13. Dey, A. K. (2001). Understanding and using context. *Personal Ubiquitous Compututing Journal, 5*(1), 4–7. doi:10.1007/s007790170019.
14. Dimakis, N., Mylonakis, V., Soldatos, J., & Polymenakos, L. (2006). Reaching outside the smart space: The memory jog gateway. In *15th international conference in computing (CIC'06)* (pp. 412–420). Mexico City: IEEE Computer Society. doi:10.1109/CIC.2006.57.
15. Dimakis, N., Soldatos, J., Polymenakos, L., Schenk, M., Pfirrmann, U., & Bürkle, A. (2006). Perceptive middleware and intelligent agents enhancing service autonomy in smart spaces. In *IEEE/WIC/ACM international conference on web intelligence and intelligent agent technology* (pp. 276–283). Los Alamitos, CA: IEEE Computer Society. doi:10.1109/IAT.2006.98.
16. Dimakis, N., Soldatos, J. K., Polymenakos, L., Fleury, P., Cuřín, J., & Kleindienst, J. (2008). Integrated development of context-aware applications in smart spaces. *IEEE Pervasive Computing, 7*(4), 71–79. doi:10.1109/MPRV.2008.75.
17. Faci N., Guessoum Z., & Marin O. (2006). Dimax: A fault-tolerant multi-agent platform. In *SELMAS '06: Proceedings of the 2006 international workshop on software engineering for large-scale multi-agent systems* (pp. 13–20). New York, NY: ACM Press. doi:10.1145/1138063.1138067.
18. FIPA. The Foundation for Intelligent Physical Agents. At http://www.fipa.org.
19. Garlan, D., Siewiorek, D., Smailagic, A., & Steenkiste, P. (2002). Project aura: Towards distraction-free pervasive computing. *IEEE Pervasive Computing, 21*(2), 22–31.
20. Hansmann, U., Merk, L., Nicklous, M. S., & Stober, T. (2003). *Pervasive computing: The mobile world (Springer professional computing)*. New York: Springer.
21. Helsinger, A., Thome, M., & Wright, T. (2004). Cougaar: A scalable, distributed multi-agent architecture. In *IEEE international conference on systems, man and cybernetics* (pp. 1910–1917).
22. Hibernate. Relational persistence for java. At http://www.hibernate.org.
23. Horling, B., Lesser, V., Vincent, R., & Wagner, T. (2006). The soft real-time agent control architecture. *Autonomous Agents and Multi-Agent Systems, 12*(1), 35–92.
24. iBATIS. IBATIS Data Mapping Framework. At http://ibatis.apache.org/.
25. Islam, N., & Fayad, M. (2003). Toward ubiquitous acceptance of ubiquitous computing. *Communications of the ACM, 46*(2), 89–92. doi:10.1145/606272.606302.
26. JADE. Java Agent Development Environment. At http://jade.tilab.com.
27. Jadex. BDI Agent System. At http://vsis-www.inormatik.unihamburg.de/projects/jadex.
28. Jason. Jason: A Java-based interpreter for an extended version of agent speak. At http://jason.sourceforge.net/.
29. Kleindienst, J., Curin, J., & Fleury, P. (2007). Reference architecture for multi-modal perceptual systems: Tooling for application development. In *Intelligent environments, 2007. IE 07. 3rd IET international conference on* (pp. 361–368). doi:10.1049/cp:20070393.
30. Kumar, S., Cohen, P., & Levesque, H. J. (2000). The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams. In *4th international conference on multi-agent systems (ICMAS 2000)* (pp. 159–166).
31. Kumar, S., & Cohen, P. R. (2000). Towards a fault-tolerant multi-agent system architecture. In *AGENTS '00: Proceedings of the fourth international conference on autonomous agents* (pp. 459–466). New York, NY: ACM Press. doi:10.1145/336595.337570.
32. Martin, D., Cheyer, A., & Moran, D. (1999). The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence, 13*(12), 91–128. doi:10.1080/088395199117504.
33. Minar, N., Gray, M., Roup, O., Raffi, K., & Maes, P. (2004). Hive: Distributed agents for networking things. *IEEE Concurrency, 8*, 24–33.
34. Olszewski, D., & Linhard, K. (2006). Highly directional multi-beam audio loudspeaker. In *Interspeech 2006*
35. Olszewski, D., Prasetyo, F., & Linhard, K. (2005). Steerable highly directional audio beam loudspeaker. In *Interspeech 2005* (pp. 137–140).
36. OWL. Web ontology language (owl). At http://www.w3.org/2004/OWL
37. Pandis, I., Soldatos, J., Paar, A., Reuter, J., Carras, M., & Polymenakos, L. (2005). An ontology-based framework for dynamic resource management in ubiquitous computing environments. In *2nd*

*international conference on embedded software and systems* (pp. 195–203). doi:10.1109/ICESS.2005.29.
38. Pianesi, F., & Terken, J. (2009). *Computers in the human interaction Loop, chap. User-centered design of CHIL services: Introduction* (pp. 179–186). Human-computer interaction series. London: Springer. doi:10.1007/978-1-84882-054-8_16.
39. Pianesi, F., Zancanaro, M., Falcon, V., & Not, E. (2006). Towards supporting group dynamics. In *Proceedings of the artificial intelligence applications and innovations (AIAI2006)* (pp. 302–311).
40. Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In J. D. R. Bordini, M. Dastani, & A. E. F. Seghrouchni (Eds.), *Multi-agent programming* (pp. 149–174). USA: Springer Science+Business Media Inc.
41. Pynadath, D., Tambe, M., Arens, Y., Chalupsky, H., Gil, Y., Knoblock, C., et al. (2000). Electric elves: Immersing an agent organization in a human organization. In *Proceedings of the AAAI fall symposium on socially intelligent agents—The human in the Loop*. Menlo Park, CA: The AAAI Press.
42. Rao, A.S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In R. van Hoe (Ed.), *Seventh European workshop on modelling autonomous agents in a multi-agent world*, Lecture notes in artificial intelligence (Vol. 1038, pp. 42–55). Berlin: Springer Verlag.
43. Saif, U., Pham, H., Paluska, J. M., Waterman, J., Terman, C., & Ward, S. (2003). A case for goal-oriented programming semantics. In *5th annual conference on ubiquitous computing (UbiComp '03)* (pp. 74–83).
44. Scerri, P., Pynadath, D., & Tambe, M. (2001). Adjustable autonomy in real-world multi-agent environments. In *AGENTS '01: Proceedings of the fifth international conference on autonomous agents* (pp. 300–307). New York, NY: ACM. doi:10.1145/375735.376314.
45. Shi, Y., Xie, W., Xu, G., Shi, R., Chen, E., & Mao, Y., et al. (2003). The smart classroom: Merging technologies for seamless tele-education. *IEEE Pervasive Computing, 2*(2), 47–55. doi:10.1109/MPRV.2003.1203753.
46. Šišlák, D., Rehák, M., Pěchouček, M., Rollo, M., & Pavlíček, D. (2005). A-globe: Agent development platform with inaccessibility and mobility support. In R. Unland, M. Klusch, & M. Calisti (Eds.), *Software agent-based applications, platforms and development kits* (pp. 21–46). Basel: Birkhauser Verlag.
47. Soldatos, J., Dimakis, N., Stamatis, K., & Polymenakos, L. (2007). A breadboard architecture for pervasive context-aware services in smart spaces: Middleware components and prototype applications. *Personal and Ubiquitous Computing Journal, 11*(2), 193–212. doi:10.1007/s00779-006-0102-7.
48. Soldatos, J., Pandis, I., Stamatis, K., Polymenakos, L., & Crowley, J. L. (2007). Agent based middleware infrastructure for autonomous context-aware computing services. *Computer Communications Magazine, Special Issue on Emerging Middleware for Next Generation Networks, 30*(3), 577–591. doi:10.1016/j.comcom.2005.11.018.
49. Soldatos, J., Stamatis, K., Azodolmolky, S., Pandis, I., & Polymenakos, L. (2007). Semantic web technologies for ubiquitous computing resource management in smart spaces. *International Journal of Web Engineering and Technology, 3*(4), 353–373. doi:10.1504/IJWET.2007.014438.
50. Stanford, V. (2002). Pervasive computing goes to work: Interfacing to the enterprise. *IEEE Pervasive Computing, 01*(3), 6–12. doi:10.1109/MPRV.2002.1037716.
51. Sycara, K., Paolucci, M., Velsen, M. V., & Giampapa, J. A. (2003). The RETSINA MAS infrastructure. *Autonomous Agents and Multi-Agent Systems, 7*(1/2), 29–48. doi:10.1023/A:1024172719965.
52. Tambe M., Min Shen W., Mataric M., Pynadath D. V., Goldberg D., Modi P. J., et al. (1999). Teamwork in cyberspace: Using TEAMCORE to make agents team-ready. In *Proceedings of the AAAI spring symposium on agents in cyberspace* (pp. 136–141). Menlo Park, CA: The AAAI Press.
53. TopLink. At http://www.oracle.com/technology/products/ias/toplink/index.html.
54. Torque. At http://db.apache.org/torque/.
55. UltraLog. The ultralog project. At http://ultralog.net.
56. Weiser, M. (1991). The computer for the 21st century. *Scientific American, 9*(9), 94–104.
57. Zorzo, A. F., & Meneguzzi, F. R. (2005). An agent model for fault-tolerant systems. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing* (pp. 60–65). New York, NY: ACM Press. doi:10.1145/1066677.1066696.