An Information-Driven Architecture for Cognitive Systems Research

Sebastian Wrede

Dipl.-Inform. Sebastian Wrede
AG Angewandte Informatik
Technische Fakultät
Universität Bielefeld
email: swrede@techfak.uni-bielefeld.de

Abdruck der genehmigten Dissertation zur Erlangung
des akademischen Grades Doktor-Ingenieur (Dr.-Ing.).
Der Technischen Fakultät der Universität Bielefeld
am 09.07.2008 vorgelegt von Sebastian Wrede,
am 27.11.2008 verteidigt und genehmigt.

Gutachter:

Prof. Dr. Gerhard Sagerer, Universität Bielefeld
Assoc. Prof. Bruce A. Draper, Colorado State University
Prof. Dr. Christian Bauckhage, Universität Bonn

Prüfungsausschuss:

Prof. Dr. Helge Ritter, Universität Bielefeld
Dr. Robert Haschke, Universität Bielefeld

Gedruckt auf alterungsbeständigem Papier nach ISO 9706

An Information-Driven Architecture

for Cognitive Systems Research

Der Technischen Fakultät der Universität Bielefeld

zur Erlangung des Grades

Doktor-Ingenieur

vorgelegt von

Sebastian Wrede

Bielefeld – Juli 2008

Acknowledgments

Conducting research on a computer science topic and writing a thesis in a collaborative research project where the thesis results are at the same time the basis for scientific innovation in the very same project is a truly challenging experience - both for the author as well as its colleagues. Even though, working in the VAMPIRE EU project was always fun. Consequently, I would first like to thank all the people involved in this particular project for their commitment and cooperation.

Among all the people supporting me over my PhD years, I need to single out two persons that were of particular importance. First of all, I would like to thank my advisor Christian Bauckhage for his inspiration and his ongoing support from the early days in the VAMPIRE project up to his final comments on this thesis manuscript. Also, I am very grateful to my colleague Marc Hanheide for frequent discussions about our ideas and approaches, for opening insights into image processing problems as well as his patience while serving as a beta tester of the developed software. Not to mention that working with Marc in different projects was always fun and at the same time productive.

The development of a fundamental infrastructure approach as described in this thesis is of course not possible in isolation. Hence, I would like to thank the members of the Applied Informatics at Bielefeld University, which are not only great colleagues but also exceptional collaborators. However, as there are too many, I cannot mention everyone. Nevertheless, I would still like to thank Jannik Fritsch for initially applying the concepts in our mobile robotics scenarios and disseminating the ideas to the robotics community as well as Ingo Lütkebohle and Jan Schaefer for their contributions and lots of fruitful discussions which inspired and initiated many extensions of the resulting software framework. Furthermore, I want to say “Dank je wel!” to Lisabeth van Iersel who always assisted in bureaucratic paperwork and was willing to listen to any issue that I came across in these years.

Last, but not least, I want to express my gratitude to Gerhard Sagerer and Franz Kummert as they provided me the opportunity to pursue a PhD in the Applied Informatics. In particular, I want to thank Gerhard Sagerer for his ongoing support, his encouraging but thought-provoking impulses and his friendly pressure. Of course, I also want to thank the additional members of the examination board, Helge Ritter, Robert Haschke and Bruce Draper for their time and willingness to review this thesis.

In addition, I will not forget to deeply appreciate the help and advice of Thorsten Gröger[1] when it came to the graphical illustration of complex circumstances – even when you were on holiday.

Finally, I want to express my gratitude to those who should in fact be mentioned first: my family. While my dad left far too early, I will always be grateful for his unquestionable trust in me. I thank my mum for sharing the excitement about my research work until now and crossing fingers whenever necessary as well as my sister for constantly motivating me even in doubtful situations. However, without the everyday assistance and patience of Christina, her negotiation skills with regard to Clara when I was staying up all night in my office, and the emotional support by both of you beloved ones, I would never have managed to come this far.

[1] http://www.stilwechsel.de

Contents

Abstract

With computer science more and more leaving the traits of solitary algorithms and distinct disciplines towards complex intelligent and integrated systems, challenging research questions are in reach to be explored in novel application scenarios. Under the term “cognitive systems” and its subfields of “cognitive robotics” and “cognitive vision”, research recently made a significant leap forward regarding these challenges. Experimental cognitive systems research is thus characterized by a flexible composition of different algorithms and the development of interdisciplinary models for artificial cognition.

Integrated cognitive systems allow us to address scientific questions that go far beyond what can be achieved with solitary algorithms. For example, such systems include personal robot “companions” or assistance systems that are embedded in the world and permit interaction with humans and their environment. Integrated cognitive systems allow us to test hypothetical models of cognition in the “real” world. Owing to the innate complexity of these systems, questions of software integration and software architecture have become research activities in their own right. Consequently, topics and methods known from software and systems engineering need to be adopted for research on experimental cognitive systems.

This thesis addresses the questions how the complexity in software architectures for cognitive systems can be reduced and how joint integration in large-scale research projects can be facilitated. It approaches these questions from three viewpoints: the functional, collaborative, and engineering viewpoint. Acknowledging their importance leads to the design of a coherent and comprehensive architectural concept that is introduced with this dissertation. This approach fuses paradigms of event-driven and service-oriented architectures with domain-specific support for cognitive systems, yielding a novel concept: information-driven integration. The resulting software architecture facilitates joint development and integration by providing on the one hand good support for the functional requirements of experimental cognitive systems and on the other hand by explicitly considering the peculiarities of research environments as integration contexts.

The application of the information-driven integration architecture in various cognitive systems projects is presented as strong evidence for the appropriateness of its design and implementation. This thesis bridges the gap between single algorithms and their respective component developers on the one side, and system integration and evaluation on the other by means of a novel integrating approach supporting the collaborative construction of experimental cognitive systems.

1. Software Integration in Cognitive Systems - A First Encounter

> In theory there is no difference between theory
> and practice, but in practice there is.
> – *Anonymous*

Cognitive Systems, Interaction, Robotics. The overarching vision of these three areas of current European research [Eur05] is to develop artificial systems that are able to act within the real world, either autonomously or in cooperation with humans. An exemplary class of applications within this paradigm are autonomous service robots, see Figure 1.1 for an ancient imagination. These kind of systems well-know from science-fiction literature are a dream of novelists and researchers since the earliest days of fiction and computer science until now. Although today in the automotive industry the number of deployed robot systems compared to employed human workers increased to the ratio of one to ten [Gat07] and robots like the semi-autonomous NASA mars rovers Spirit and Opportunity [MLB07] are capable of exploring the solar system, successful examples of cognitive systems like the envisioned service robot that are deployed in the real world and sharing their environment with humans are still rare.

Over the last two decades, some of the innate challenges in building autonomous systems stood the test of time. Robust, while at the same time fast enough visual perception and understanding of the robot's environment or speech recognition in noisy surroundings are exemplary problems, researchers are still faced with nowadays. The integration and coordination of the manifold behaviors of a complex robot, the task of action selection or the adaptation of system behavior to unknown situations while assuring high reactivity are equally important research topics which are not fully explored yet within the domains of cognitive systems research. To bridge the gap between the vision of cognitive systems and the existing instantiations of these concepts, scientists all around the world covering a very broad range of disciplines are currently concerned with the development of systems that need to combine a wide variety of new functionality into novel applications in an inherently interdisciplinary approach. Following the cognitive systems paradigms, recent research projects develop a large number of innovative applications like autonomous vehicles [TMD+06] or pro-active driver assistance systems [MGS+07], multimodal interaction systems that are capable of assisting people with dementia by observing and guiding them [HBPM07] as well as surveillance systems, e.g., capable of monitoring aircraft servicing operations at airports [TBF+06] - just to mention a few of them.

One of these recent endeavors to realize instances of such systems from the area of cognitive interaction technology and the inception for this thesis was the VAMPIRE project [VAM06], which has been an international collaborative research project funded by the European Union (EU) on cognitive computer vision. One of its primary aims was to construct an artificial cognitive system that is able to provide context aware assistance through a head mounted display to a human user by understanding and memorizing what the user sees and recognizing the actions he carries out in a natural environment.

While a large part of the research conducted in the VAMPIRE project has been conducted in augmented reality, context awareness, computer vision, pattern recognition, symbolic reasoning, and knowledge representation, an additional research question was addressed to provide an avenue for the further advancement of cognitive systems: it is the question of finding architectural principles, development methodologies, and software technologies for the integration and software development process within interdisciplinary cognitive systems projects that shall enable researchers of different background to jointly work on an large-scale software-intensive research systems.

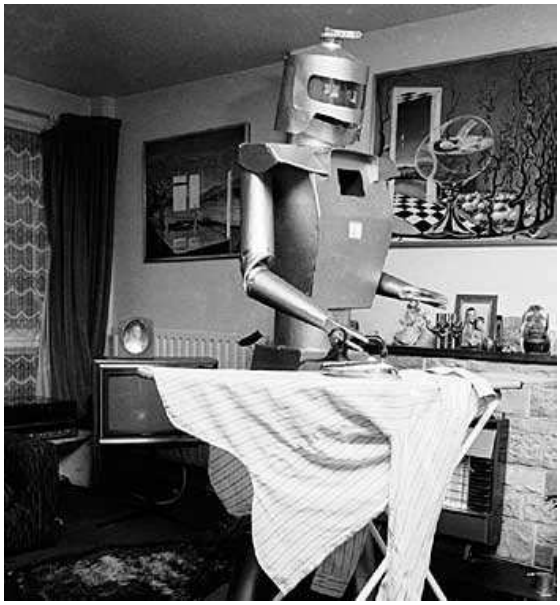


Figure 1.1.: *Early vision of a domestic service robot [Wal05], acting autonomously in a human-centered environment.*

The IEEE Standard Glossary of Software Engineering Terminology [IEE90] defines integration as "*the process of combining software components, hardware components, or both, into an overall system*". Although software integration has been an important topic for business software vendors and computer science research over the last decades and has been studied at different levels of abstraction, it is a rather new trend that software development and integration within the field of autonomous intelligent systems is explicitly focused by the scientific community, e.g., in robotics with the series of *Software Development and Integration* (SDIR) [Bru05, Bru07a, Bru07b, Bru08a] workshops. Based on research mainly conducted in the VAMPIRE project, this thesis takes up on this emerging topic, addressing the specific challenges of **Software Integration** in collaborative interdisciplinary research projects on cognitive systems and presenting a coherent approach to the software integration of functional modules in a consistent architectural approach for this domain. The implicit question that needs to be addressed is how to manage the ever growing *complexity* resulting from advanced application scenarios and yet more integrated functionality in the context of experimental research that is otherwise not explicitly considering software architecture and software integration. The intricate architectural and technological challenges in this task and their specifics with regard to collaboration in large-scale cognitive systems research projects such as VAMPIRE or COGNIRON [Cog06] will be analyzed from three distinct viewpoints: a *functional*, *collaborative* and *architectural* perspective.

The resulting **Information-Driven Integration** (IDI) architecture that is presented in this dissertation carefully considers the insights gained during the conducted requirement analysis. Conceptually, the IDI approach exploits the content of exchanged high-level information in the software architecture of cognitive systems for effecting coordinated interactions between functional modules. It thereby raises the level of abstraction for the design and development of complex distributed software architectures compared to generic middleware, and provides functional services for collaborative software development in joint research projects on experimental cognitive systems. Its design adopts principles of service-oriented and event-driven architectures adopting current enterprise integration technologies like group communication middleware, state-of-the-art database and XML technologies as well as formal methods for describing discrete event-based systems. The introduced architectural models allow for a modular development of loosely-coupled cognitive systems architectures.

Research on the IDI approach was conducted coevolutionary throughout the collaborative development process in the VAMPIRE project. A central quality of information-driven integration is that it facilitates an efficient software development process in interdisciplinary cognitive systems research, which permits researchers to develop complex distributed software architectures for real-world cognitive systems. The necessary requirements analysis, the developed integration models and the resulting software architecture are the main contributions of this thesis. While the proposed architecture has already been used successfully in several experimental research projects aiming at real-world prototype systems, an emphasis will be put on a description of the assistance system the author developed in close collaboration with the partners in the VAMPIRE project. This initial application served as an iterative testbed and evaluation scenario for the developed architectural approach.

Because the scope of cognitive systems research is extremely broad, the following section introduces the subset of cognitive systems, which are in the center of our current and our envisioned research. While still being a broad area for research, cognitive systems aiming at *Cognitive Interaction Technology* (CIT) define the actual application context that is addressed by the presented approach. Having introduced the integration domain, subsequently the three perspectives on integration that contribute to the innate complexity of software development and integration in cognitive systems research projects will be introduced.

In the remainder, the research questions this thesis addresses as well as its objectives will be summarize and a short overview of the followed technical approach will be given. To conclude this introduction, the structure of this dissertation together with its main contributions will be outlined.

1.1. Cognitive Systems for Human-Machine-Interaction

During the last decade, computer vision research has seen a change from brittle and narrow applications to more general and adaptive approaches. Starting out from early work of Christensen and Crowley [CC97], the term **Cognitive Vision** has been established [Ver04] subsequently to describe approaches that try to combine achievements from artificial intelligence, computer perception, machine learning and robotics with the aim to build more robust, resilient and adaptive computer vision systems.

Within this context, Christensen restricts the process of cognition to be a *generation of knowledge on the basis of perception, reasoning, learning and prior models*. As a consequence, a cognitive vision system needs to be embodied [Ver08] in order to actively sense its environment and to interact with its surrounding for knowledge acquisition [Chr03]. Embodiment provides the basis for situated cognition [HW05], which permits a system to learn and act in the context of its environment. For instance, a single cognitive activity may be embedded in a temporal stream of activities carried out or supported by additional tools, systems or humans outside of the considered system. Without the possibility for physical embodied exploration, the ability to perceive the dynamical changes in the environment and to communicate actively about events by interaction, learning can not take place in a cognitive system.

Rather recently, e.g., in the current EU research roadmaps [Eur05], cognitive vision has been embedded in the more general paradigm of **Cognitive Systems**. Cognitive systems conceptually extend to other perceptual modalities such as haptic or auditory senses. According to that scheme, cognitive vision systems can be seen as *visually-enabled cognitive systems*.

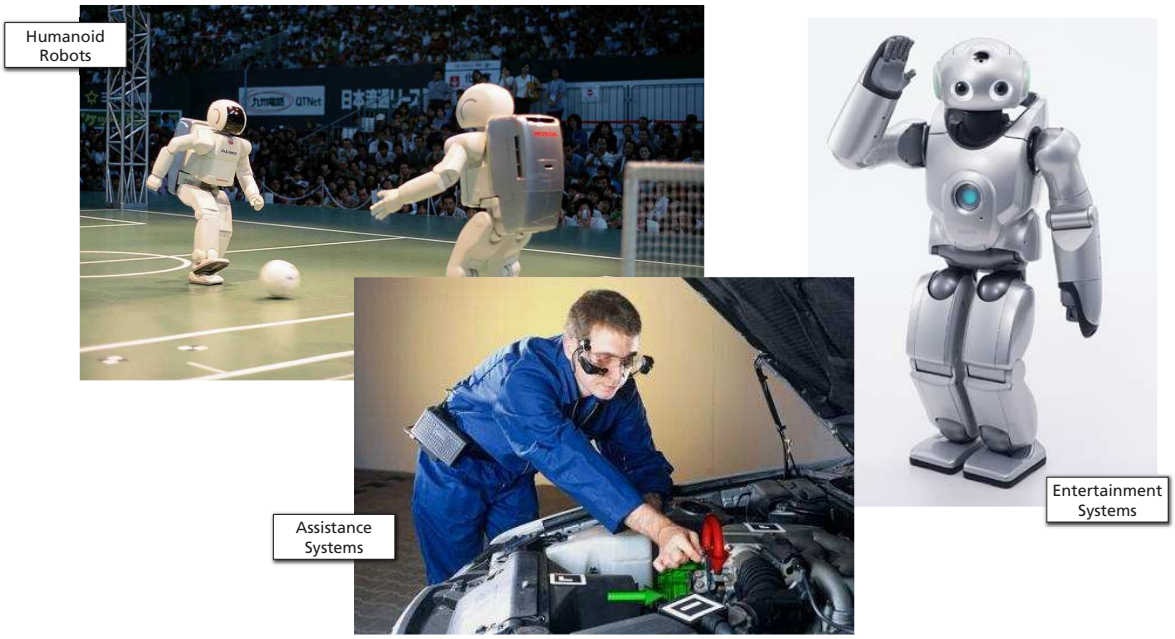


Figure 1.2.: *The Honda ASIMO humanoid robot [Yos04], the Sony Qrio [Gep04] entertainment robot and an augmented reality assistance system [CTS07] are exemplary applications of state-of-the-art cognitive systems technology operating in the real world.*

While the general research agenda of cognitive systems extends to cognitive neuroscience, epistemology, cybernetics and others, the work carried out by the author and his colleagues in the VAMPIRE cognitive vision project and other cognitive systems projects like COGNIRON [Cog06] is targeted at the construction of systems featuring advanced human-machine-interaction capabilities. The goal is to assemble systems that make use of a complete loop from perception via cognitive processing to re-action in the world rather than providing a formal theory about possible cognitive architectures [Cas03]. Thus, our efforts on building systems with cognitive abilities are to a large extent focused at the integration of an increased number of perceptual modalities and features to facilitate interaction with humans. Examples for the use of sensorial information in such cognitive systems are to exploit vision for human body tracking to allow interaction through gestures [SHH+08] or the analysis of human language for prosody information. On top of these cues, typically higher-level cognitive functions aggregating information are added, e.g., a dialog system based on speech and gesture recognition in order to communicate in a natural way with human interaction partners [LWS06, SHS07].

Cognitive interaction has various application domains ranging from assistance systems for engineers or monitoring systems for elderly people that provide home-help displaying useful prompts or calling care personnel [HBPM07] to interactive robots, that obviously need to perceive their environment and interact or even collaborate with humans. Some examples are shown in Figure 1.2. Due to the ever growing set of cognitive abilities combined in these applications, the question of an effective integration, which is the main topic of this thesis, is raised with increased priority. In order to find answers to this question, let us now look more closely at the reasons for the intricacies cognitive systems research may impose on system development.

1.2. Viewpoints on Software Architectures for Integrated Cognitive Systems

While the benefits of a research approach explicitly addressing embodied cognitive systems as an individual goal becomes nowadays more widely accepted and endorsed by current research programs [Eur05], the downside of aiming at these types of integrated systems is that they come with significant costs. Practical experience shows that if an implementation of such software-intensive systems is actually carried out, scientists within project teams quickly face problems of *programming-in-the-large* [DK76].

With the ever increasing size and complexity of experimental cognitive systems, the design, specification and evolution of the overall system structures become increasingly important. The need to handle the complexity introduced by demanding scenarios that call for richer functionality and more deeply integrated systems in fact urges developers and scientists to deal with these overall system structures. Hence, talking about the software integration of artificial cognitive systems, inevitably poses questions of software architecture. For this reason and because many considerations within this thesis are conducted from an architectural viewpoint, we need to define more clearly what architecture means in this context. A recent definition of this term, which was established within the ANSI/IEEE standard 2000-1471 that deals with the architectural description of software systems is as follows:

Definition 1.1 (Software Architecture) *Software architecture is defined [...] as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. [IEE00]*

Davide Brugali[1] raised at the workshop on SDIR-II at ICRA 2007 the metaphorical question why it generally is so hard to build well-architected systems in the domain of robotics resarch. Even if we restrict our view to the software aspects of integration in cognitive systems, in the perspective taken on in this thesis the answer is in the intricate interplay of domain-specific, project-specific and engineering aspects as depicted in Figure 1.3, which makes this topic a research activity in its own right. In order to give some first answers to the question of Davide Brugali, let us shortly introduce these three viewpoints on the different sources of complexity that contribute to the overall challenge of integration. Please note, that the analysis of the three perspectives will be confined in the first part of this thesis, identifying requirements from the subsequently introduced tasks.

Functional Viewpoint

The envisioned overall behavior of a cognitive system emerges from a rich set of cognitive capabilities, employing a variety of computational models, e.g., for perception, learning and classification or for feedback and interaction assembled in a specific functional architecture. Goerick and Ceravola describe this architectural level with the following definition we will adopt within this thesis:

Definition 1.2 (Functional Architecture) *A functional architecture represents the constraints of a hypothesis or model of the network of functional areas in the brain that makes different modules or components interact. [CG06]*

[1] Prof. Davide Brugali is the chair of the IEEE RAS Technical Committee on Software Engineering in Robotics.

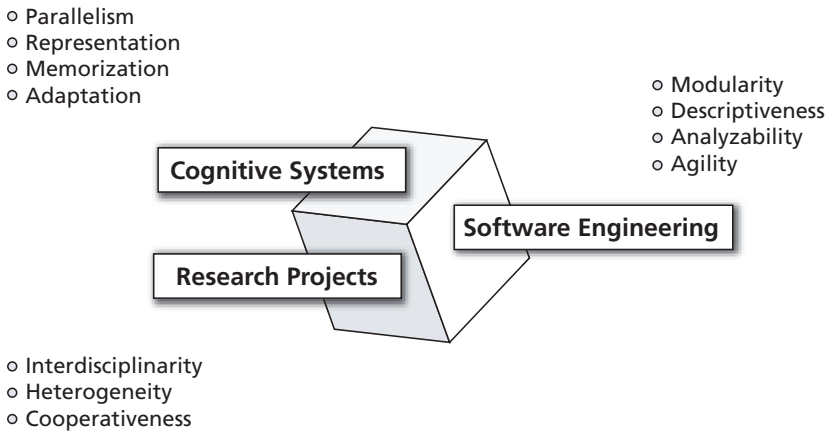


Figure 1.3.: *Following a holistic approach, the challenges of software integration in the domain of artificial cognitive systems research are analyzed from three distinct viewpoints.*

An initial challenge that generally arises in software architectures for embodied cognitive systems research already on the level of this functional architecture is to account for the inherent *parallelism* of such systems. Components must be able to process new information that is provided by attached sensors with low response time in order to quickly react to changes in the environment. The resulting system must be able to adapt its behavior at least in a time scale that is appropriate to ensure system safety and that is convenient for interaction partners. This concern and the processing power necessary for handling high-volume datasets as well as complex processing operations often increases complexity as it implies the distribution of processing load and the exchange of large amounts of data. This manifests itself in a system architecture that is usually build on top of middleware technologies for distributed systems. As these are in parts overly complex, their use sometimes imposes intricate usability problems. However, as parallel processing is a core necessity of nowadays cognitive systems, large parts of the work presented in this thesis will address this problem.

Looking from a rather technological perspective, the offered services on the level of the functional architecture not only differ in terms of their concrete function but are realized in component implementations of differing granularity and size. Component abstraction levels span from data processing modules performing image segmentation to high-level modules for semantic reasoning, e.g., for system self-awareness [Slo98]. Due to this variety an additional challenge at this level is the sheer amount of different *representations* that range for example from image data to acquired semantic knowledge implying inevitable challenges for handling interoperability issues.

Closely linked to the question of representation is the question of how to *memorize* which is a prerequisite for learning, adaptation and cognitive systems in general [Chr03]. Chapter 2.2.2 will elaborate more on this, as memory functionalities have been one of the fundamental functional requirements originating from the research paradigm followed in the VAMPIRE project.

In order to allow a cognitive systems functional architecture to *adapt*, e.g., to environmental changes, the processes in a cognitive systems architecture need to be flexibly coordinated and orchestrated as well as dynamically reconfigured, possibly across different abstraction layers. The coupling between reactive behaviors and more deliberative cognitive functions thereby still pose challenging questions of coordination and control as they were already introduced two decades ago by Brooks [Bro91].

Thinking about hybrid architectures in terms of the ongoing debate between cognitivist and emergent architectures [Ver08], it is still an open question how to combine cognitive functions following these different paradigms. While the scientific community does not provide any single answer to these problems, adaptation and coordination are central challenges in the software architectures of almost any experimental cognitive systems.

Collaborative Viewpoint

The functional capabilities as well as their technological consequences define the inevitable *physiological complexity* of the integration challenge in cognitive systems research. While this kind of complexity is regularly recognized and considered by domain-specific software architectures, this thesis additionally proposes to explicitly address the social complexity [Fia07] of this task, originating from the *integration context*.

This context is comprised of the project stakeholders, the environment in terms of resources and process models as well as the system developers themselves. For cognitive systems research, the context is defined by the structure and management processes prevalent in research projects and the involved people, e.g. scientists, students or even people from industry, their personal backgrounds and motivations. Although software engineering research has proven that social aspects are critical for the success [DL99] of software projects, the processes in research projects often simply neglect these insights, even despite the fact that the challenges imposed by the domain and the context are not easier at all than in regular business information systems.

Collaborative research projects on cognitive systems such as VAMPIRE or COGNIRON are inherently *interdisciplinary* and may involve a fairly large number of scientists from different domains. While this interdisciplinarity is attributed to be a catalyzer for progress in many areas of cognitive systems research, it poses additional questions for software development such as how to communicate about system-level structures between the involved scientists. If a common software-intensive demonstration system is desired as a concrete outcome of a project, the involved team members therefore need to develop a shared vocabulary and understanding of the problem domain, which can become much harder when people have very different scientific backgrounds.

From practical experience, this thesis assumes that the likeliness of performing a successful system integration in the course of an interdisciplinary project is dramatically increasing, when the integration model addresses this challenge explicitly, e.g., by introducing models, clear definitions and languages for system-level entities that are easy to understand, formalize and match well their later real-world implementation. As a concrete consequence, it must for instance be possible for all project members to easily engage in communication about the system-level integration within a project meeting, about representations and interactions of “their” modules within a system architecture.

This is especially important within this context as the situation in research projects concerning integration often exhibits similarities to the *anarchical* situation of enterprise software integration [Joh02], which is characterized by only a limited amount of *cooperativeness*. This may lead to less consultation and agreement between stakeholders and participants or at best as an oligarchical situation that is present when researchers either employ or develop a certain set of standards for a given problem domain. Large parts of this thesis address this aspect and propose a framework that can deal with only a limited amount of agreement between module developers and their corresponding software modules.

Another challenge that arises from the interdisciplinary constitution of project teams can be the varying level of expertise and background knowledge even when merely computer scientists are involved. We can safely assume that only a small number of the involved researchers will have the necessary background knowledge and motivation to face the intricacies of low-level integration technologies like sockets, marshaling or concurrent programming - just to name a few aspects. Therefore, the integration architecture needs to provide *suitable abstractions* on a level high enough to be usable for the - according to Bill Gates [Gat07] - "average" programmer. Microsoft nowadays tries to address this challenge with the recently released Microsoft Robotics Studio [Jac07] for the domain of robotics, explicitly targeting for example the problems of concurrent programming by employing a completely asynchronous process integration model.

Engineering Viewpoint

The engineering viewpoint subsumes a large number of functional and non-functional requirements important for software and system engineering of experimental cognitive systems. A primary challenge is how *modularity* can be achieved in cognitive system instances and how larger subsystems can be composed out of much simpler services.

This composability and further facts about modularity are critical for the overall utility of the approach as will be explained later on in Chapter 3 particular for supporting cognitive system research. While the functional architecture is aligned with the domain, this viewpoint additionally considers the challenges arising from the functional architecture's realization in hard- and software components. Thus, let us define the term *System Architecture* to distinguish this architectural level from the functional level:

Definition 1.3 (System Architecture) *A representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and human interaction with these components. [Car06]*

Within this definition, the focus is set on the specification of hardware and software deployment, e.g., specifying the distribution of software components and processes to hardware nodes. Additionally and particularly important for interactive cognitive systems is the description of use cases [Coc01] by which humans shall use a system instance or interact with it. As the functionality of systems is different, the system architectures are also typically rather special to each class or family of systems.

The resulting challenge is to raise the level of *descriptiveness* and to facilitate the modeling of different attributes of the system architecture like structural composition or dynamic execution in a distributed system. However, modularity and descriptiveness may reduce, but do not completely prevent failures in system design and execution. For these reasons, functionality for *analysis* of the static and runtime architecture of an IDI system must be provided. An exemplary test could be whether a promoted external interface of a component matches the referenced service interface type.

A different challenge is to facilitate an *agile* development process that, e.g, allows for easy testing and incremental development of cognitive systems architectures. Even if some parts of the overall architecture are still unfinished or need to be replaced, a corresponding simulation should in principle be applicable.

The Integrative Viewpoint

While the aforementioned challenges for software architectures supporting cognitive systems research provide already some guidance for closer consideration, there are many more requirements, e.g., originating from the scenarios and research paradigms in the VAMPIRE project, that will be discussed in the corresponding chapters for each of these perspectives. However, once acknowledging this triangular view on software integration, a new way of looking at software architecture is following.

This additional viewpoint on software architecture shall respect all three previously introduced viewpoints and propose a level of abstraction that is easily tractable for the developers and stakeholders in its specific integration context.

The introduction of explicit component *interaction patterns* may serve as an example for this. They reduce the complexity of the interplay between the individual building blocks of a cognitive system and thus increase the level of abstraction used in integration. Capturing typical interactions in a small number of reusable patterns is beneficial both for system as for module developers and can play a similar role for system-level integration as design patterns for developers of object-oriented software [GHJV95]. Therefore, the following definition of an **Integration Architecture** focuses on this level of abstraction and summarizes the main architectural aspects that need to be considered at this level of software architecture:

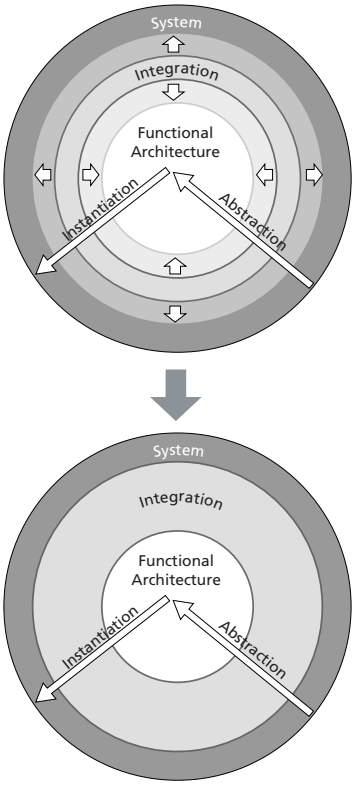


Figure 1.4.: *Evolution of the integration functionality.*

Definition 1.4 (Integration Architecture) *An integration architecture deals with the structural composition of software components into a system instance. It provides design elements which bind domain functionality provided by software components to artifacts of the integration architecture, thereby exposing their services to other components. It provides design patterns for the composition of design elements and establishes guidelines for the selection among design alternatives. It provides functionality for physical distribution, communication, synchronization and coordination between design elements and functionality for data access within an architecture.*

The motivation behind the notion of an integration architecture such as the one introduced in this thesis is that it shall describe an approach making the development of functional architectures as pain-free as possible under the constraints of the collaborative viewpoint. Ceravola and Goerick [CG06] argue that one important goal of research on software architectures in cognitive systems must be to identify common concepts of cognitive function or necessary computational machinery with the aim to increase the available knowledge on the architectural foundations these functions are based upon. Figure 1.4 takes up this general idea and illustrates the increased condensed knowledge in the fundamental architectural layers during system evolution. While the author of this thesis completely agrees with the general idea of incrementally learning the right architectural abstractions from system evolution, Figure 1.4 utilizes the rather different architectural viewpoints introduced previously.

1.3. Research Questions, Objectives and Approach

The VAMPIRE EU project following a scenario-driven research approach offerred an excellently suited testbed to investigate software integration in collaborative cognitive systems research. In the course of the project, the ambition of this work evolved to find answers to the aforementioned challenges that are generally useful for experimental cognitive systems. In an incremental refinement process the proposed information-driven integration model has subsequently been applied to several other projects of different sizes from small student projects to large-scale integrated research projects. Still, the fundamental research question that needs to be addressed by this work is the following:

> ***What are architectural concepts and paradigms suitable for handling the innate complexity of software development in cognitive systems research projects?***

In order to find an answer to this very broad research hypothesis, it was necessary to focus on different aspects of this task. Therefore, the following set of questions were chosen as more concrete guidelines during the development of the proposed integration model:

Q1: What characterizes research projects in the cognitive systems research domain and what are the requirements imposed on software integration?

Q2: What are the functional requirements that are generalizable and need to be addressed by an integration architecture in the given domain?

Q3: What abstraction level needs to be chosen for the framework to be accepted by its interdisciplinary users?

Q4: Which selection of integration patterns and functionality is necessary for a successful integration of experimental cognitive systems?

Q5: What lessons can be learned from the application of information-driven integration approach with regard to the development process of cognitive systems in large-scale research projects?

In order to find answers to these questions, well-known concepts and novel techniques from the fields distributed systems, (enterprise) software integration, artificial intelligence and software engineering, e.g. requirement analysis, as well as the cognitive systems domain itself have to be taken into account.

To assess and investigate the requirements and challenges of integration in cognitive systems research projects, the author actively participated in the collaborative construction of different integrated demonstration systems as the one described in Chapter 2 and escorted the process of system architecture development from the very beginning. This actual participation in the iterative construction procress has been very important for finding answers to some of the questions outlined above and for the development of the three individual viewpoints on integration.

During the continuous integration of prototypical systems in this and other projects, several developer interviews with expert and rather novice users were carried out. These experiences and the insights from these discussion strongly influenced the design of the integration architecture.

The concept of information-driven integration is informed by ideas from event-driven and service-oriented architectures. Further guidance for the realization of the approach resembles from object-oriented software analysis and design, e.g., the different design patterns, the implementation of the framework is built upon. Data integration, coordination and the content-based routing of information between components is realized by exploiting recent XML technologies, e.g., a native XML database

as the basis for the memory functionalities. In addition to that, the developed architecture is itself based on other, rather low-level middleware for group communication and provides an easy-to-use programming API as well as a set of tools that shall allow researchers to efficiently build and glue together cognitive systems instances. The resulting toolkit provides the core-functionality of the proposed architecture in terms of its services, libraries and programming interfaces for C++ and Java.

1.4. Outline and Contributions

Following up on the introduction, this thesis is coarsely composed of four subsequent parts as shown in Figure 1.5. Part I (Chapter 2-5), *A Systems Perspective on a Cognitive Systems Project*, deals with requirement identification and analysis, whereas Part II (Chapters 6 and 7), *The Information-Driven Integration Approach* introduces the conceptual design of the approach presented in this thesis. Part III (Chapters 8 and 9) presents its application and thus evaluation in actual research projects. To commence this thesis, Part IV (Chapter 10), *Synopsis*, features a conclusion that summarizes and highlights the insights gained.

Chapter 2 identifies generalizable requirments for software integration in cognitive systems from a functional viewpoint with a a particular focus on the paradigms of the VAMPIRE project.

Chapter 3 highlights the particularities of conducting integration in a collaborative and interdisciplinary research project that aims at real-world integrated systems and identifies requirements resulting therefrom. Chapter 4 discusses a concise subset of high-level requirements on distributed systems and software architectures from a software engineering viewpoint already taking into account the results from the previous chapters.

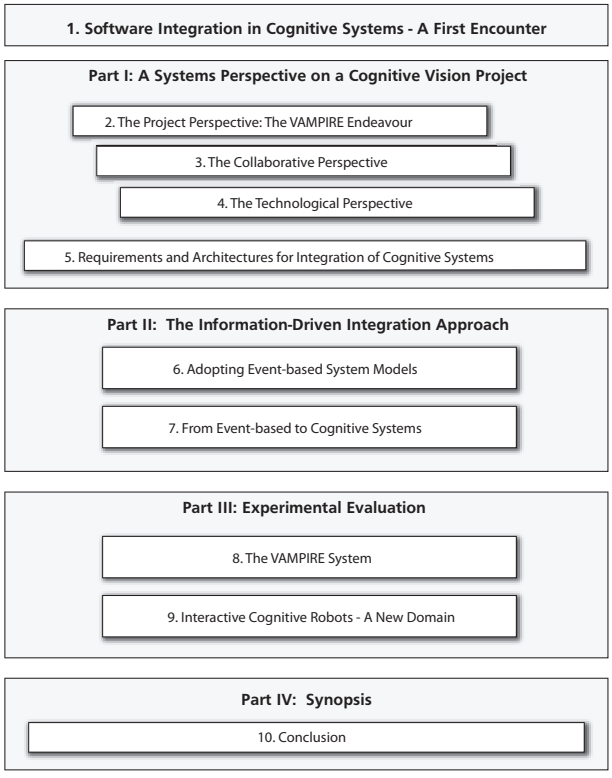


Figure 1.5.: *Structure of this thesis.*

Chapter 5 reviews a qualified selection of related work in the area of software integration and integration architectures for cognitive systems. This review is conducted based on a small number of aspects that are composed from the previously identified requirements. The subsequent selection among the massive number of approaches is informed by a brief overview of related fields of research that face similar challenges in integration. Finally, three approaches are reviewed and assessed according to their strength and weaknesses. The main contribution in the first part of this thesis is the analysis of an EU project such as VAMPIRE from a holistic perspective, not limited to an identification of functional and non-functional requirements, but additionally considering the social complexity of this task in the requirements process.

Part II presents the information-driven integration approach as the key contribution of this thesis. Chapter 6 introduces the core models of the integration approach and describes how the paradigms of event-driven architectures were adopted for information-driven integration. Chapter 7 extends on these domain independent models towards features for the integration of eperimental cognitive systems with a particular focus on the memory, domain and coordination models that address central functional requirements of the VAMPIRE project.

The role of Part III is to evaluate the developed software architecture and the introduced concepts in a system context. It starts with Chapter 8, focusing at the application of the information-driven integration approach in the VAMPIRE EU project in order to show that the proposed models were useful in a real-world project. The actual integration of the augmented-reality assistance system [WHWS06] and in particular the application of the memory and coordination models of the IDI approach are explained. The memory and coordination models represent fundamental functional building blocks within the presented assistance scenario and are thus contributions of this thesis with regard to the VAMPIRE project. A brief technical performance evaluation demonstrates the utility of the realized software architecture for the integration of cognitive systems. In addition to this, a system evaluation with naive users is a contribution, which demonstrates that it was possible to test the assistance system developed with the presented approach in a real-world context. In order to emphasize that information-driven integration is a more general concept, Chapter 9 briefly highlights the application of the introduced approach in the domain of cognitive robotics. Despite the mentioned contributions, a main matter of Part III is to highlight how the introduced IDI models ease the integration of real-world cognitive system instances.

Part IV represents the synopsis of this thesis. Chapter 10 puts the introduced models into the context of the overall integration approach and shortly reports on the experiences and lessons learned both from a system engineers view as also from the component developers perspective. This thesis ends with a an outlook envisioning possible further research directions in software architectures for supporting experimental cognitive systems research.

Part I.

A Systems Perspective on a Cognitive Vision Project

Research on real-world cognitive systems is a broad and technically challenging area spanning a number of different research disciplines. Within the following Part I, a catalogue is compiled consisting of generalizable aspects that are required across many instances of such systems. This analysis is carried out primarily from the viewpoint of software engineering with the aim to gather requirements that are essential for an efficient collaborative construction of experimental cognitive systems.

This requirements identification process is guided by the idea to analyze the overall integration challenge from three distinct but intertwined perspectives. Starting with an analysis of the envisioned scenarios in the VAMPIRE EU project in Chapter 2, architectural consequences are derived which lead to functional aspects that need to be supported in an integration environment for real-world cognitive systems. Implications from the specific integration context, which is set by collaborative research projects, are discussed in Chapter 3. The requirements identified until then guide the subsequent discussion of the resulting architectural and technological challenges in Chapter 4.

To commence this part, the identified set of requirements will be analyzed and clustered in a smaller number of coarse-grained aspects in Chapter 5, which a versatile integration architecture for cognitive system must consider in the first place. These aspects provide an avenue for the selection and assessment of related work and shall serve as a guideline for the conceptual development of the integration approach described subsequently.

At the end of Part I, the interested reader should be aware of the architectural backgrounds considered as important for software integration in cognitive systems research projects from the perspective taken on in this thesis and thus the motivations that guided the development process for the approach to be introduced in Part II.

2. The Project Perspective: The VAMPIRE Endeavour

The aim of this chapter is to gain a deeper understanding of the challenges for software integration in complex cognitive systems with a particular focus on the specific challenge in the VAMPIRE project in order to approach the right targets. Within software engineering, a common property of a requirement is that it shall be clearly testable in terms of functions, usability tests, metrics or performance numbers [PB04]. For the sake of clarity and to not restrict the possible solution space in preface, the requirements term is used more liberally in the following, because many properties for an envisioned software integration architecture are better described here on a high level of abstraction than in elaborated requirement specifications.

In the following, an architecture-driven problem decomposition and requirement analysis is presented that incorporates the knowledge and insights gained in the course of the VAMPIRE project as well as subsequent collaborative research projects. Instead of simply enumerating all the "-ilities", which are indeed important for a software integration approach, like flexibility, adaptivity, etc. the identified requirements shall be explained in their systemic context.

As an initial step, the primary concepts of *Cognitive Vision Systems* are introduced and analyzed to bind and confine the problem space. Adding up on that, the main research themes of the VAMPIRE cognitive vision project, which are important from a system's perspective will be outlined. Particularly, the functional requirements originating from the very characteristics of the *Human-in-the-Loop* paradigm and the fundamental research hypothesis that a *Visual Active Memory* (VAM) is instrumental in artificial visual cognition will be examined for their implications on architectural properties.

2.1. Cognitive Vision - An Emerging Discipline

For almost fifty years now, computer vision researchers are striving to make "*computers see*". Starting as a sub-field of artificial intelligence (AI), computer vision emerged as a discipline in its own right. Over the last decades, innovations in vision research shifted the possible space of applications gradually from laboratory environments, e.g., the famous "*Blocksworld*" scenario, to more realistic settings. Nowadays, vision systems are frequently used for varying tasks, e.g., active tracking of humans [CRTT97], or in machine vision for all sorts of automatic visual inspection [Ver91], to increase product quality and allow for an online failure detection.

While these innovations and applications of computer vision were major developments and paved the way towards further research, they were still rather focused on narrow application scenarios or limited to a specific functionality. In contrast to vision research, AI was in general concerned with rather deliberative and symbolic models of human cognition with an equally limited scope of applicability in the real world [Ver08].

With the advent of cognitive computer vision as a new paradigm emerging in the 1990s, AI and computer vision remarried to a certain extent [Neu04] in order to tackle more ambitious targets in the real world. Consequently, the EU started a new research programme about cognitive computer vision in the year 2000 [Eur01]. To improve the robustness and adaptivity of artificial vision systems, the design of resulting systems is often inspired by findings in biology, cognitive science, and psychology. According to Granlund [Gra05] a Cognitive Vision System (CVS) can "*perceive and learn information in an interaction with the environment and generate appropriate, robust actions or symbolic communication [...]*", which can be treated as visually-enabled cognitive systems. A cognitive *vision* system therefore shares many of its high-level requirements with other types of cognitive systems, making the resulting catalogue largely applicable also for integration architectures, e.g., from the domain of cognitive robotics.

2.1.1. Modularity and Multiple Computation

A basic assumption behind our approach to realize cognitive vision systems is to model the abilities of such artificial machines through the orchestration of highly interconnected processing modules. Following, for instance, Minsky's *Society-of-Mind* [Min86] theory, the underlying hypothesis of many cognitive models is that the mind is made up of a possibly large number of interacting cognitive agents with varying specificity. Rather than entering into a debate about the subtleties of philosophical, psychological or biological viewpoints on the emergence of cognition, the essence from an architectural viewpoint here is the question of decomposition. As decomposition usually breaks down a larger problem into smaller pieces in many iterations, we can safely infer that the result will be a larger number of cognitive processes that will finally be implemented in software modules thereby posing questions of software *modularity*.

While aspects of modularity in software architectures as defined by Meyer [Mey97] extend over several dimensions, discussed in greater detail in Chapter 4, the fundamental requirement an integration architecture must fulfill on the functional level of cognitive systems is to support the modular decomposition of a larger problem, which can be summarized as follows:

Requirement 2.1: Modular Decomposition An architecture yields modular decomposability when it facilitates the comprehensible decomposition of a problem into a smaller number of easier subproblems that are still manageable by the integration architecture. In order to satisfy this constraint, the resulting partitioning should allow for independent, parallel development and interconnection through a structure as simple as possible.

Following up on modularity and inspired by biological findings, the principle of multiple computations [Cru03] is applied for the modeling of processing in a CVS. Hence, a possibly large number of cognitive processes are generally executed in parallel and compute partly redundant multi-modal information, "*playing around*" with the information contained in the system. More technically speaking, for instance, several feature extraction or recognition processes are applied in parallel computing the same information. Following this concept in an artificial system shall not only lead to an increased performance if computation of complex input data is partitioned, but also to a higher robustness within the overall system by exploiting a larger degree of redundancy. Acknowledging this, let us note the subsequent requirement considering parallelism:

Requirement 2.2: Parallelism An architecture must permit to execute multiple computations and parallelize the processing of decomposed modules, e.g., by using multiple processing units on a single physical computer system or by distributing the computations over a set of several networked processors or computers.

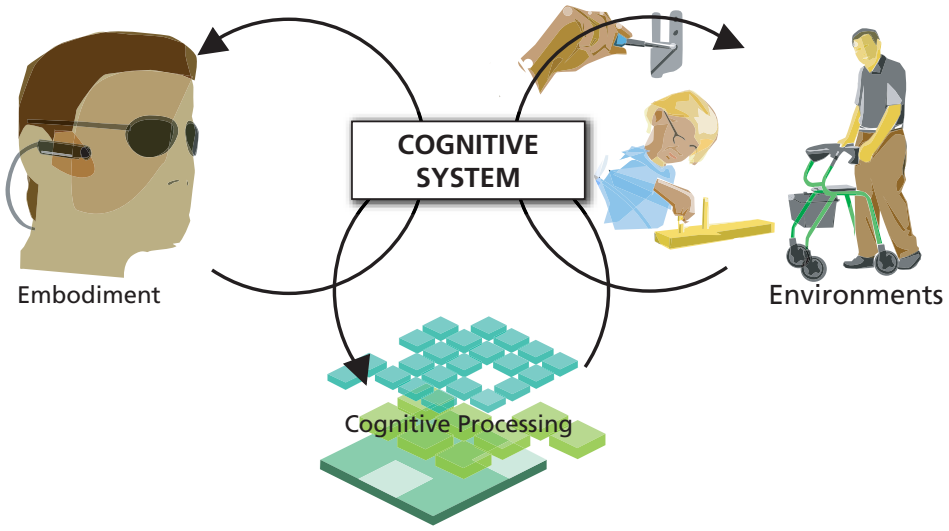


Figure 2.1.: *Embodied cognitive systems integrate numerous processes for multi-modal perception and production in order to interact with their environment in an adaptive manner. They aim at supporting humans, e.g. in performing everyday tasks or decision making.*

As soon as parallel processing is envisioned, at least in technical architectures this raises important questions of concurrency and synchronization, which will be discussed from a technological viewpoint in Chapter 4.

2.1.2. Dynamic Coordination and Adaptation

In addition to these purely technological considerations, the overarching question how an efficient coordination between the various modules in a cognitive system can be achieved is an example for a requirement that can not be cleanly assigned to a single perspective. Here, technological and functional viewpoints mix up and are not fully orthogonal. For instance, features of the integration environment may limit the possibilities how behavioral change can be effected.

Aiming at cognitive systems that are embedded in the real world as illustrated in Figure 2.1, the integration architecture must allow, e.g., coordinated deliberative and reactive behaviors. Let us consider an anthropomorphic robot as an example, cf. Chapter 9. Such a system must coordinate actions it undertakes to achieve its designated objective such as taking a cup, with the reactions forced on it by the environment, for instance, human actions that could interfere with the planned grasping sequence. Thus, let us note the following requirement:

Requirement 2.3: Coordination Dynamic coordination is necessary in a technical architecture that exploits parallelism and providies an avenue for managing the dynamic behaviors that can be executed in the system. While the focus of sequencing is the mapping of serial behaviors to a synchronized series of system actions, coordination goes beyond this and provides structures for executing complex behaviors and tasks that depend on the runtime dynamics, for instance on the current perceptual state of the system or temporal aspects.

In contrast to many classical computer vision system architectures, where the processing and data flow between the constituent modules is often pre-programmed and regularly follows a single architectural

style like pipe-and-filter [SG96], system architectures of CVS reveal a richer set of architectural styles that are employed to build hybrid architectures combining data-driven bottom up with goal-directed and knowledge-based top-down processing. It can even be necessary to connect different processes at runtime, therefore yielding a fully dynamic system architecture, for instance for the ongoing learning of new perception-action mappings, which can be summarized as follows:

Requirement 2.4: Flexible Orchestration Instead of predefined feed-forward processing chains, a CVS uses multiple sensors and recognition pathways to gather information about its environmental context. In order to build architectures for such systems, flexible means of managing the interconnection between different cognitive processes are required, for instance to realize hybrid architectures that allow for sensory bottom-up as well as actuatory top-down processing.

2.1.3. From Sensorial to Symbolic Information

A classical cognitive computer vision system consists of multiple levels of perceptual processing, thereby incrementally increasing the contingent of semantic information. While a multitude of methods and approaches for visual scene understanding exists, the fundamental requirement of computer vision research is that the development of corresponding algorithms by providing reusable datatypes and fundamental operations in the form of a library or prototyping environments must be supported.

Requirement 2.5: Computer Vision Support Modular development of image processing algorithms must be supported in order to foster reuse and the prototyping of novel approaches. It must additionally consider their integration as processing modules in a larger system and provide common computer vision functions.

As indicated in Figure 2.1, an experimental CVS is designed for acting in the real world and is thus usually faced with a large set of high-dimensional input signals. In order to further process these high-volume datasets, abstraction processes that extract relevant information from the input data are necessary. Although there is ongoing debate whether and when to generate symbolic descriptions of the relevant information in the input space, a drastic compression of the input data is needed, e.g., by the generation of abstract models for categorization. These models can for instance be beneficial to deal with missing information or to introduce context within the symbolic domain. Furthermore, symbolic descriptions are the basic means of communication about the perceived entities or events and can be used to interact with other systems or humans [Gra05]. From a system's perspective, this calls for the possibility to exchange these descriptions in extensible representations between the different cognitive modules.

Requirement 2.6: Extensible Representations Within every artificial cognitive system processes exchange information and work on representations of this data. As soon as learning and adaptation is envisioned, for instance to dynamically add new visual features that are extracted by perceptual modules, representations must be able to dynamically evolve. Hence, data structures must represent information in an extensible way.

In order to provide a basis for a seamless communication with humans improved interaction and communication capabilities of cognitive systems are extremely important. A basis for these capabilities and further advanced capabilities of cognitive system is thus some kind of a memory structure or a federation of different memories, e.g. working and long-term memory [HZW07]. This is due to the fact that memorization capabilities are prerequisites of learning and adaptation in cognitive beings, particularly if learning processes are active over a longer period of time. In cognitive vision, memories manage information and knowledge from various knowledge sources like spatial and contextual

information, as well as scene and event descriptions. Hence, an important features of memory systems is the ability to relate new information to already existing information [Gra05].

Additionally, accounting for the fact that memory is basically a limited resource, processes that distinguish relevant from irrelevant information and act upon that decision like forgetting or compacting are necessary [Chr03]. While it is still unclear how artificial memories for cognitive systems are organized, many of the processes in a cognitive system architecture require some kind of memory. Based on the assumption that memories share similarities, an integration architecture should support the notion of a memory.

Requirement 2.7: Memory An integration architecture shall feature a working memory support for cognitive processes, allowing them to store information in relation to already existing knowledge and recall this later in different contexts. In order to allow processes to operate in a general manner on their memory content and taking into account the evolving information in a system, a generic architectural solution to this problem needs to be flexible. As soon as memorization comes into play, a corresponding way of removing outdated information is necessary, too, because memory is a limited resource both in biological systems and artificial systems.

2.2. The VAMPIRE Project

The aspect of memory in cognitive systems provides an excellent link to the VAMPIRE project. The long-term vision of this project has been to proceed towards cognitive assistance systems that serve as memory prosthetic devices and assist human users in everyday environments. The project was funded within the above mentioned thematic priority on cognitive computer vision by the EU and contributes to many of the aspects of a CVS introduced above. It was carried out from May 2002 to July 2005 and involved five international academic partner institutions located in four different European countries[1]. The participating scientists were attributed to be experts in the fields of vision, visual learning, scene analysis as well as augmented reality and human-computer-interaction.

In order to realize a small step into the direction of memory-prosthetic devices, the projects primary aim was to conduct research on the concept of *Visual Active Memory Processes* (VAMP), which shall facilitate artificial intelligent systems to better understand what they see based on what they have previously memorized. Due to the fact that these systems are embedded in the real world, it is inevitable to acquire knowledge through exploration of the environment and its interaction with a human communication partner. Therefore, an important research question was how to couple the model acquisition and recognition processes for an adaptive scene understanding, because it is impossible to predict beforehand all possible sets of objects or actions a system is exposed to over time, e.g., in an office environment.

Picking up on the necessity of interaction capabilities in such systems, the second line of research carried out in the project (yielding the second half of the VAMPIRE acronym) was concerned with the development of advanced techniques for the *Interactive REtrieval* of previously acquired knowledge, for instance about objects that have been recognized and memorized by the system. Conversely, it was necessary to design a multi-modal interaction facility for the system itself, for instance to ask a human user in case of ambiguities or to give attentional feedback thereby visualizing its internal state.

[1]The VAMPIRE project consortium included research groups from Graz University of Technology, Austria, the University of Surrey, United Kingdom, the University of Erlangen-Nuremberg and the Applied Computer Science as well Neuroinformatics research groups at Bielefeld University, Germany.

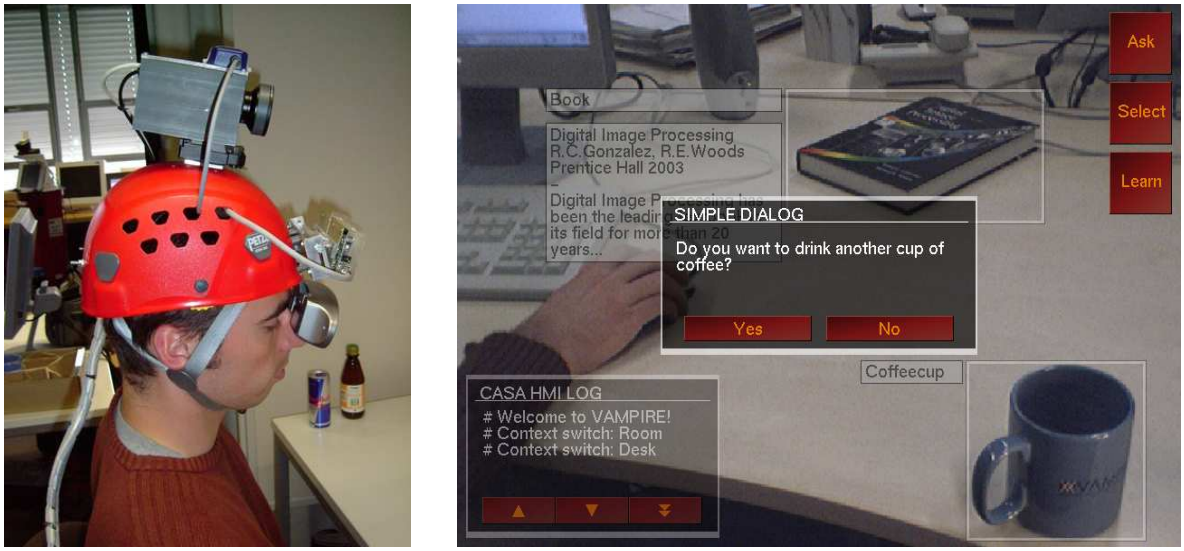


Figure 2.2.: *Scenario: A user is sitting at a table wearing the system's hardware interface. She is supported by the system augmentations while acting on the table.*

Within all the individual research activities carried out in the course of the project to realize the necessary vision, reasoning and interaction processes, two overarching parts stand out that will be briefly introduced in the following, because they are fundamental to the identification of functional and architectural requirements for the proposed software integration architecture for a cognitive (vision) system: the human-in-the-loop paradigm and the concept of a visual active memory.

2.2.1. The Human-In-The-Loop

Recent research on human-computer interaction (HCI) aims at increasingly natural interfaces between human users and information systems. In contrast to virtual reality, *Augmented Reality* (AR) describes an approach by which a user's view is augmented with additional information while still being situated in the real world's context. Looking at current augmented reality applications [ABB+01], the interaction space is often extended to real-world environments; see [KLP04] for an example. AR is thus well suited for interaction with a cognitive vision system. The representations of this information range from text annotation and object highlighting to the projection of complex 3D objects.

As a consequence this also leads to novel applications for computer vision research as this external environment in turn has to be perceived through available sensors. Within the primary scenario in the VAMPIRE project, we extended these ideas and embedded the human user directly in the processing loop of the system. During the project the term *human-in-the-loop* [BHW+05] has been coined for this idea. Within the space of cognitive systems research, embodiement is often realized by robots acting in the real-world. However, in the VAMPIRE project, the hypothesis was whether the human user can actually represent a certain kind of embodiment for the system. For a realization of the human-in-the-loop paradigm, not only a novel type of a cognitive computer vision software with advanced HCI capabilites needed to be developed, but also a special type of hardware device was designed, the so called *Augmented Reality-Gear* (AR-gear).

The prototypical realization of this device, which has been used by naive users in the evaluation studies carried out in the course of the project is shown in Figure 2.2. The design guideline for the development of this hardware platform for use in the VAMPIRE scenarios was to set aside all external sensors. Therefore, the AR-gear, which will be explained in greater detail in Chapter 8 integrates all sensors necessary to realize an interactive assistance system on a mobile platform.

The resulting tight coupling of the human in the processing loop yields a novel type of embodiment, the so called *mediated embodiment* [Han06]. Thereby the perception-action loop is closed, which ultimately allows for the active perception necessary in a cognitive vision system. A concrete instantiation of this concept depicts Figure 2.2, which shows a human user that acts in an office environment wearing the AR-gear and an example of an augmentation of the users' field of view.

The hypothesis that lead to the development of this setup is that it is beneficial for a cognitive assistance system to follow the ego-vision paradigm [Han06] and to take on the perspective of the human user. The resulting situation is characterized by *shared attention* where the system sees what the human user sees and vice versa. Exploiting this situation, the system can pro-actively interpret situations and assist the user in solving given tasks, e.g., by directing him through visual prompts to objects that have been previously memorized. Conversely, the user may execute actions for the system like recording different views of an object or is able to guide the systems' attention, for instance by focusing on interesting objects.

Let us consider one of the classical examples that guided the development of such an assistance system: imagine your somewhat cluttered desk and yourself wearing an AR-gear that is actively monitoring its environment. Coming to your desk, you drop your keys somewhere on the table. After some time acting in the scene, you are placing a sheet of paper on top of the keys without explicity noticing it. After a while you spent working in this environment, you may have forgotten about the location of the keys. This is a situation where you ask your cognitive assistance system: "*where did I put my keys?*" The system queries its visual history and guides you by viusal prompts to the place on the table where it has seen the keys lastly. Utilizing an overlay image, you will quickly be reminded of the position of your keys.

During the course of the project, we experienced the necessity to extend the possible ways of interaction. In order to facilitate a natural communication between the users and the system, for instance for object learning, we enhanced the AR-gear by microphones and a software component that allows for multi-modal interaction, for instance through speech recognition or head gestures [HBS05]. The combined functionality of the AR-gear and this communication component allows for different types of interaction that directly contribute to the goals of VAMPIRE. Firstly, an interactive object learning is made possible by focusing on a previously unknown object, recording views of it and finally labeling it through speech. Further interaction was realized for information retrieval and an envisioned multi-user collaboration, see [SHWP07] for details.

While many properties of the envisioned scenario are functions that directly map to cognitive vision processes, we can identify the following three requirements resembling from this experimental setting that are critical for the design of a suitable integration architecture:

Requirement 2.8: Distributed Processing In contrast to some procesess tightly coupled with the AR-gear that need real-time performance, the higher level perceptual components in the VAMPIRE scenario may process the recorded video images in soft real-time. Because of the limited resouces on the mobile platform, large parts of the processing therfore need to be distributed to external processing nodes.

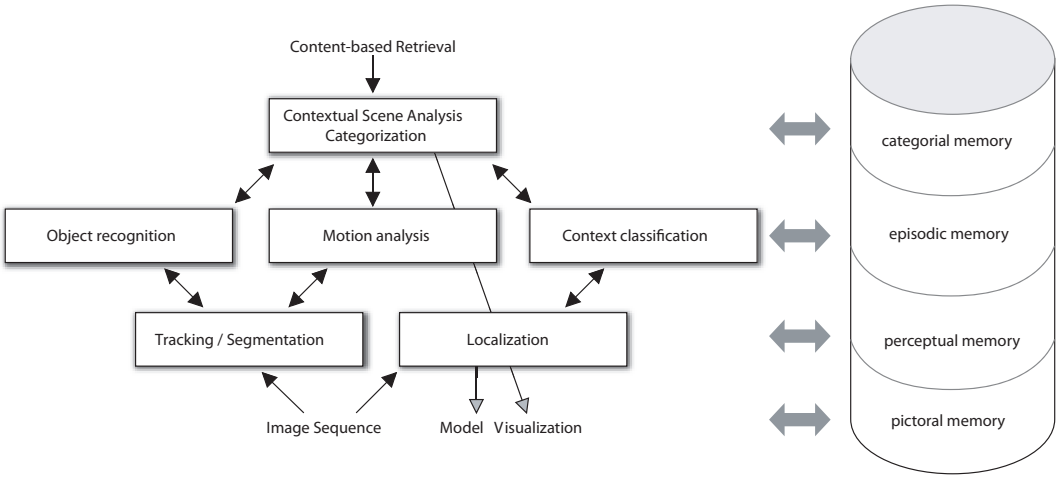


Figure 2.3.: *Sketch of the conceptual architecture of the visual active memory and its processes.*

Adding up on the necessity of distributed processing, an integration architecture needs to ensure a sufficient throughput for the simultaneous distritibution of visual input. Multicast communication will be particularly important as a larger number of cognitive processes may request the video stream.

Requirement 2.9: Reactivity The overall performance in terms of low latency of an underlying communication infrastructure usable within the VAMPIRE scenario needs shall lead to an update frequency of the presented augmentations that is subjectively convenient for users of the system.

The perspective on the performance profile of an integration architecture for a cognitive vision system like VAMPIRE is that system evaluation with user studies is preferred over quantitative benchmarking as reactivity and throughput need to be evaluated in a system context as it is a product of overhead in the integration architecture, its use by the cognitive processes and the compuations in the cognitive processes themselves.

The AR-gear and the envisioned assistance scenarios allowed to focus efforts during the project and helped to implement a scenario-driven research approach that facilitated in the development of a common understanding among the project partners. Section 3.1 will describe the additional effects that scenario-driven research has on the integration process and the developed software architecture. In the following, we will describe and analyze the conceptual architecture within all development of VAMPIRE has been subsumed.

2.2.2. The Visual Active Memory Concept

While the neuro-physiological architecture of the human brain or of even less complex vertebrates like birds is still not fully understood, a number of studies support the idea that memory is a time dependant process that yields at least a separation into short-term and long-term memory. According to Tulving's *SPI-model* [TM98], the long-term memory is structured in a hierarchy were the information of the higher levels is partially grounded in the lower ones and new information is promoted from the lower levels in a serial transformation to the higher levels of the memory. Within this model, memory access is independent from the storage mechanisms and information retrieval is not bound to the state of the memory when the information was initially memorized.

Following these findings, the main research hypothesis of the VAMPIRE project assumes that a so called *Visual Active Memory* (VAM) provides an avenue for learning and development of cognitive capabilities in vision systems. Conceptually, the idea behind this architecture is that it shall build up and maintain a visual history of the world [VAM04]. The functional architecture of a VAM as it was envisioned within the context of the VAMPIRE assistance systems is shown in Figure 2.3.

A VAM features a number of interconnected active processes and information that is shared between these through memory structures. This information is for instance used for the ongoing learning of new object and motion patterns for improved recognition and categorization in a dynamic world. According to the SPI-model, the memory itself is hierarchically structured into four different levels of varying abstraction that are organized successively where (sub-)symbolic data, information and knowledge is processed and stored. Furthermore, it must be possible within this hierarchical structure to setup and resolve associative links between the contained elements through the active memory.

The processing components that provide the feature extraction, model acquisition, learning, fusion and recognition functionalities that operate mainly on the active memory are termed *memory processes*. The constituent processes generate, fuse and promote information within all layers of the active memory. As a central element of the overall system that resembles to well-known architectural styles such as Blackboard [SSRB00], the actice memory represents a generalizable function that shall be directly supported by the integration architecture.

Requirement 2.10: Active Memory Support An architecture yields support for an active memory if it provides memory functionality that extends over local memory functions to a system-wide shared information architecture that allows cognitive modules to store, recall, update and remove mutli-modal information. Furthermore, an important concept is that cognitive modules are being aware of modifications in this data.

As Figure 2.3 suggests, the hierarchical decomposition of the layers within an active memory can follow different discriminators. While several other aspects like the relevance or nature of an information as well as selective activation of knowledge might be important, we focused within the VAMPIRE systems firstly on two distinctive dimensions. The VAM concept explicitly addresses on the one hand the reliability of a hypothesis and on the other hand the “age” of an element. By age we mean at least two different things: the creation time of a memory element and the time when it has been lastly updated. Concerning the reliability of information, we enforce the paradigm of no universal truth. This assumption states there shall be no irrevocable fact stored in the active memory, because even human perception is often error-prone and assumed facts need to be revoked. Therefore, particularly in the sensorial and perceptual layers of the memory architecture, the feature extraction and recognition processes need to support this concept. The hypothesis concept [HBS04] supports another invariant feature of a memory for cognitive system [Chr03]: the ability to actively forget irrelevant information.

Based on the various features that indicate the relevance of an element, every memory layer contains differently parametrized *forgetting* processes that actively compact or remove memory elements, which are no longer referenced, are unreliable or simply outdated. Forgetting is a necessary requirement due to the fact that information with low reliability or which is simply not interesting for the system just increase the cognitive load without being useful and could lead to resource contention in a technical realization of an active memory.

Requirement 2.11: Forgetting As a function of the memory architecture itself, forgetting is not first and foremost a responsibility of the cognitive modules. Forgetting processes designed for active memories must be able to operate on common extensible representations regardless of the specific type of encoded information.

Furthermore, certain types of information, e.g., previously observed elements can be overwritten by novel events or repetitive experiences, thereby indirectly removing old ones. The chances to remember certain experiences increase by consolidation of information, which creates strong encoding. This consolidation sequence of information is a key concept for visual active memory architectures that is directly reflected in the stack of different memory layers. In order to provide these functionalities in an architectural model, the notion of a memory is vastly extended from a local storage to a shared repository that pro-actively manages the acquired information and serves as a mediator for information that has to be exchanged between different cognitive modules of a system.

In order to assess further functional requirements resulting from the visual active memory architecture as we developed it in the course of the project, let us in the following have a closer look on the different layers within the active memory, their corresponding typical memory processes and the types of information involved.

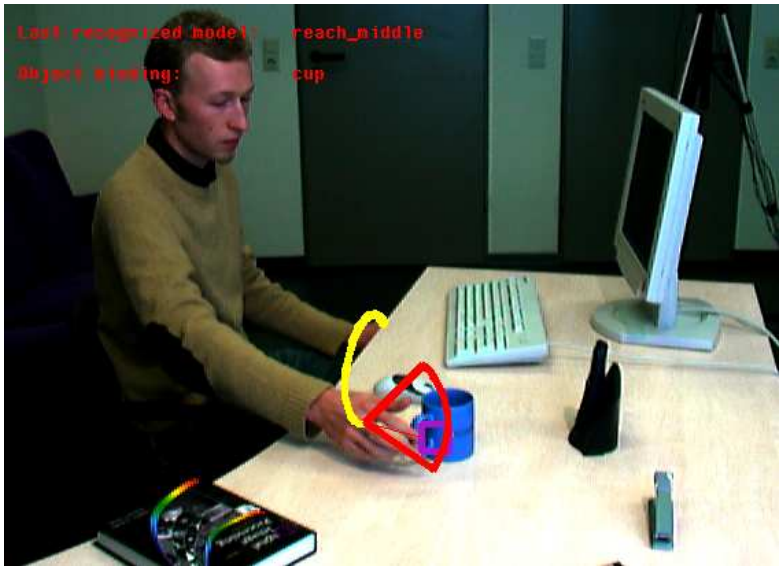
Sensorial Layer The bottom-most layer of the visual active memory contains processes that mainly acquire raw sensor data and provide this information in a suitable representation to modules, which are located in higher levels of the architecture. Within the VAMPIRE project, the function of this layer is to realize a kind of a *pictorial* memory. This is due to the fact that the information processed at this level is memorized for later analysis by higher-level memory processes, for instance to subsequently train a face recognition classifier with a set of image patches that was recorded earlier. Another example is to compensate for the unrestricted head motion and the limited field of view in an ego-vision [Han06] system as it is the case for the VAMPIRE augmented reality applications by exploiting mosaicing techniques [GHC+04].

Perceptual Layer The data-driven processes in this layer extract and track features on the data that is provided by the memory content and the processes in the sensorial layer. They perform an initial detection and recognition of basic percepts. A resulting *percept* is commonly referred to as a compact and partially invariant representation of a significant entity in the respective sensing space [Gra05]. A visual percept for instance is a more compact representation of a relevant feature, object, or any other relevant entity in the image space than its iconic image. Within the VAMPIRE systems, the typical example for a percept has been an object hypothesis, e.g., a cup that has been recognized in a frame of the input data as depicted by the class labels in Figure 2.4(a). The results shown in Figure 2.4(b) additionally underline that the system has to cope with false positive hypotheses as well. Therefore, an architecture for a visual active memory needs to support a fusion of different input cues or use context to increase robustness in an unconstrained setting like the VAMPIRE scenario where the user shall move his head arbitrarily. The overall amount of data in the perceptual layer of the memory is still huge as for example a set of object hypothesis in the present assistance systems was typically generated at frame rate, typically yielding in the production of about 30 to 150 object hypothesis per second depending on the perceptual context.

However, the resulting transformation from iconic representations in the sensorial layer to the descriptors in the perceptual layer already yields a dramatic reduction of the input data volume. The kind of information processed in this memory layer is usually a mixture of symbolic and subsymbolic data. Due to the high frequency of incoming percepts, the rather unreliable nature of this information, and the fact that information is seldom updated by other memory processes, a rigorous forgetting process is employed that turns this memory in fact into a kind of a short-term memory.

(a) Head motion

(b) Gazing



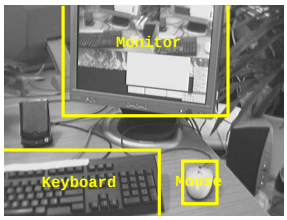
(c) Human action

Figure 2.4.: *Figures (a) and (b) show typical results of an object recognition module in an unconstrained office environment. The generated percepts are submitted to the perceptual memory layer. Image (c) depicts the results of an action recognition module that posts its hypotheses about action events and their context to the episodic memory.*

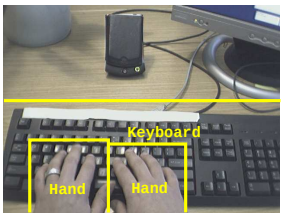
Episodic Layer The next higher level within a visual active memory is the episodic layer. The information processed in the episodic memory defines a symbolic alphabet to represent the relevant entities that have been detected in the perceptual information. The information memorized in this level is of a higher quality in terms of an increased reliability and a longer temporal validity. A representative example of a process located at this level, which in fact links the perceptual and the episodic layers is the anchoring process mentioned earlier. This multi-modal anchoring, which will be explained in more detail in Chapter 8, fuses a large number of percepts into a new hypothesis that is stored in the episodic memory, yielding a memory element with a significantly higher reliability. Within this level additionally the context of visual objects and events is introduced by means of for instance geometric, spatial and temporal relations between individual memory elements. While the memory elements can be linked to (sub-)symbolic data in the pictorial and perceptual layers, the information in this layer is mostly symbolic. The fact that the information in this layer is usually valid for a longer term and is more reliable is directly reflected by a forgetting process that is accordingly parametrized.

Conceptual Layer The conceptual layer contains stable knowledge about the scene evolution of the real-world and the cognitive systems' internal models and categories, which are defined through object and motion models as well as their functions and context. Examples for context in this sense are associations between an object and its usage role as shown in Figure 2.5(c). For instance, it seems rather typical that humans utilize a mug when drinking a "cup of coffee". The knowledge that is build up in this level of the memory originates from three different sources.

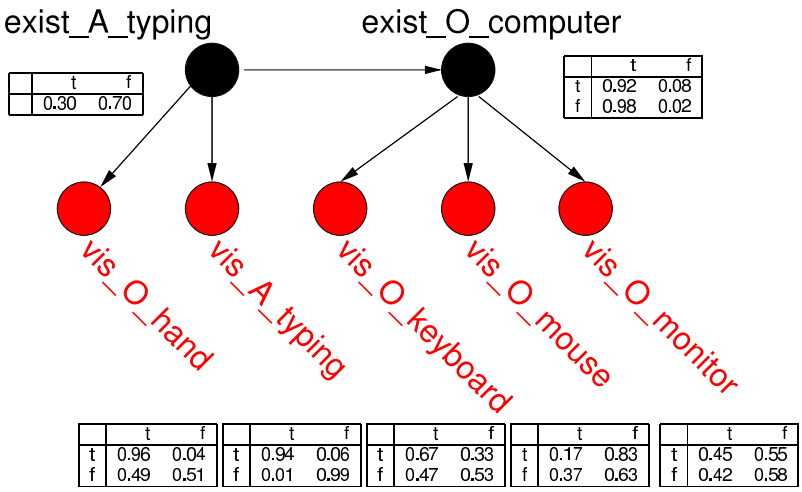
Firstly, the memory processes similar to the anchoring mentioned previously promote hypothesis that proved to be stable over a longer time from the episodic memory to this memory layer,



(a) Typical computer setup



(b) Hands typing on keyboard / monitor invisible



(c) An exemplary functional dependency concept for a computer typing setup, which is maintained in the conceptual layer of a visual active memory.

Figure 2.5.: *The annotated images (a) and (b) show the perceptual hypothesis generated in each situation by the recognition processes. Figure (c) depicts an exemplary Bayesian network that is used in a consistency validation process [HBS04] for memory content.*

which serves as the basis for memory functionality that a human users can utilize to recall specific information. Secondly, large parts of the encoded knowledge like object or action models and contextual information can be dynamically re-encoded by memory processes that for example statistically analyze the information and their associated dynamics. By this means, for instance new object models can be acquired dynamically, traversing the links between stable but previously unknown percepts, their corresponding features and low-level iconic representations. Again, this needs to be carried out in interaction with the user in order to provide the system with the information necessary to annotate the new models with the semantic concepts of the human user. Last but not least, this memory layer allows to short-cut the bootstrapping process of a cognitive system by “injecting” domain knowledge that has been designed by a human expert or that was acquired by other systems of the same kind during previous missions. The kind of data memorized and processed in this layer is again both symbolic and sub-symbolic but represents stable knowledge that is valid even on long time-scales. Even though, memory elements can be removed if the reliability of a knowledge hypothesis retroactively drops under a certain threshold.

The complex algorithmic processing that is conducted in the cognitive modules within a VAM, its repository style architecture and the reactivity that is needed in the assistance scenario strongly suggest an asynchronous processing model for the communication between the different memory processes themselves and the active memory. This is necessary for the sake of reactivity of the resulting system and the fact that this system is likely to break-up if it would be based on a synchronous pull-style communication, for instance if the memory processes would ongoingly query the memory for new information. The shared information in the visual active memory and the asynchronous processing models shall not only improve reactivity but also facilitate background learning, e.g., to train new object classifiers from recorded object views.

To commence this section, let us note asynchronicity as a general requirement for the functions provided by an integration architecture:

Requirement 2.12: Asynchronicity As a synchronous processing model seems unsuitable for the parallel processing that is conducted in the visual active memory. Modules shall not wait passively for the arrival of new information and asynchronous communication models must be supported. This shall allow an improved level of concurrency and increase the reactivity of the overall architecture.

2.3. Summary

This chapter introduced some of the requirements for an integration architecture that can be identified from a functional perspective on general cognitive vision systems and the VAMPIRE EU project and its scenario aiming at augemented reality assistance in particular.

Besides general requirements like modularity and coordination, the VAMPIRE scenario needs strong support for distributed computing to achieve a suitable performance for the perceptual processing necessary. This particular requirement has challenging technological implications that will be discussed in more detail in Chapter 4.

Acknowledging the utility and the concepts of a visual active memory, another set of requirements has been identified that deal with supporting this type of a functional architecture. The development of the required support for an active memory will be one of the distinguishing concepts of the approach to be introduced in Part II of this thesis.

3. The Collaborative Perspective

As outlined in the introduction, the analysis of the challenges for system-level software integration is conducted from three perspectives. Having introduced the functional characteristics that contribute to the inevitable *essential* complexity of architectures, the requirement identification shall proceed along a new dimension: the social complexity of software development and system integration in the context of collaborative research projects.

Let us consider the VAMPIRE EU project as a prototypical example. This project had a duration of about three years with a total of 340 person months, not counting the contributions of involved student assistants. In terms of overall code size as a coarse hint for the complexity of a project, the latest demonstration system we integrated at Bielefeld University featured about 260 thousands lines of code developed in-house or by project partners. While the number of code lines can provide only a rough estimate, its magnitude may indicate that the development processes in such collaborative research projects obviously has to consider questions of *programming-in-the-large* [DK76].

Adding up on the aspect of project size, the *scenario-driven* research methodology that has been pursued within this and other projects such as the COGNIRON EU project aims to bind scientific questions to their evaluation in real-world scenarios through experimental cognitive systems prototypes. This aim demands iterative and incremental development processes that impose additional requirements for the design of a software integration architecture for cognitive systems. Picking up on that, we will discuss further consequences that arise from the heterogeneous and interdisciplinary environment in which software development and integration are carried out.

This new perspective on the challenges of building cognitive systems is explained by firstly introducing the idea of a *scenario-driven* research process. As a consequence, the relevance of the actual construction of experimental prototype systems is emphasized. Subsequently, the impacts of scenario-driven research on the software development process are discussed.

The results of a survey carried out during the COGNIRON Winter School on Human-Robot-Interaction [Cog08] and presented at the SDIR-III [WL08] workshop suggest that the context for software integration in cognitive systems research is indeed particularly challenging due to Interdisciplinarity, heterogeneity and the ambition to actually collaborate on the level of interacting software artifacts.

3.1. The Scenario-Driven Research Process

Scenario-driven research is a methodology for conducting collaborative research that continuously compares the developed hypotheses with their applicability to previously specified evaluation scenarios. Figure 3.1 provides a high-level overview of this concept and highlights the role of system-level integration in this context.

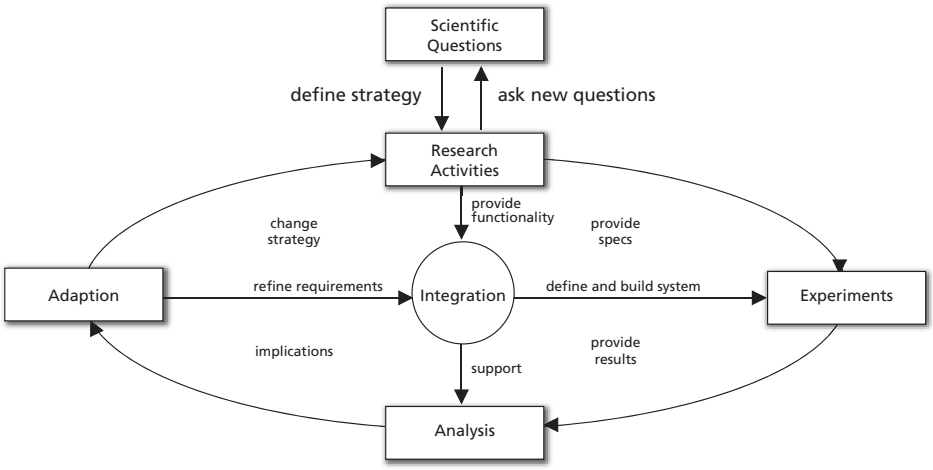


Figure 3.1.: *System-level integration is an enabling method to facilitate experimental research in collaborative projects on cognitive systems. Results from experiments with integrated demonstration systems shall affect the scientific hypotheses of the individual research activities.*

The core scientific questions of a project are usually tackled by a number of so-called *research activities*. These activities often conduct their work in isolation without interfacing other project partners. Following a scenario-driven research approach, the situation is different: researchers define collaborative experiments that involve several other partners on the basis of a common scenario, which will contribute to the demonstration of scientific results and shall provide insights on previously unknown aspects of a problem domain.

To facilitate these experiments, the developed prototypical functionality is provided to the integration research activity, which develops a system architecture that employs the novel functionality in the given scenario. Besides performing the experiments in close collaboration with the researchers of the involved scientific activities, a subsequent task of integration is to assist in the interpretation of the results of the experimental evaluation. As the consequence of each iteration, the interpreted results shall lead to an adaptation of the research and integration strategy.

Looking from a system perspective, this approach as carried out in the VAMPIRE EU project, can particularly yield the following benefits for collaborative research:

- The individual project partners are "*glued*" together and motivated through a common scenario already at early stages in the project. This helps in mutual understanding, which is essential for the success of larger interdisciplinary projects.
- The project development gains momentum from the very beginning, because first results are already visible early in the course of the project.
- Early and continuous integration within the project helps to identify risks in the overall architecture already in early stages of a project. This is particularly important as fundamental architectural changes at later stages in a project quickly become costly.
- The individual research activities can easily test their developed hypotheses in realistic settings that conform to the overall scientific scenarios of a project instead of simplified simulations, which often suppress the complexity of the real world.
- It allows for a better assessment of project progress. An evaluation in terms of a scientific

experiment that integrates a limited number of features like object learning and labeling by speech is more meaningful than to report on the observation that for instance *"80 percent of the code for the integrated system is almost there"*.

An additional example for scenario-driven research are the different so-called *key experiments* (KE) in the COGNIRON EU project that is concerned with the development of robot companions for use in largely unrestricted, natural environments.

Within this project, the KE1 [Cog06] lead by Bielefeld University is about a home-tour robot that is able to interact with a human instructor showing the robot its apartment. This experimental setup not only allows for the ongoing evaluation of the project partners' research results but additionally serves as an indicator for the overall project progress and as a vehicle for increasing communication and exchange between the involved scientists.

Unfortunately, this vision of an incremental research paradigm that is aligned with realized system instances to iteratively conduct real-world experiments in defined evaluation scenarios is difficult to achieve. Apart from the potential difficulties in the individual research activities, there are equally hard challenges for the software development and integration process, which underlines the need for coherent system-level integration approaches.

3.2. Software Development and Scenario-Driven Research

The scenario-driven research approach naturally comes with the way software and systems are developed in such projects. Software development processes in general are concerned with the set of activities that need to be carried out to produce a software artifact [Som01].

Within software engineering, two lines of thought are well established. On the one hand, the sequential software development processes like the *Waterfall* [Roy87] or *V-models* [AR08] that are organized around a cascade of phases like specification, design, implementation and testing. On the other hand, there are iterative models of software development like Boehms *Spiral* model [Boe88] or Kent Becks *Extreme Programming* [BA04] (XP) approach that interleave the different phases and reiterate these many times in short cycles until the desired features of a software product are realized. In order to facilitate a scenario-driven research approach, this thesis suggests to follow an iterative software development approach.

Following an iterative approach in a research project, particularly for cognitive systems is beneficial, because the requirements and the necessary space of designs are inherently poorly understood at the inception of a project. Accounting for the *"myth of stable requirements"* [McC04] a fundamental truth is that the better people are understanding a problem, the more likely requirements will change within software projects. As a primary aim of science is to better understand the innate problems of a given domain, it seems very likely that requirements are going to change and develop in the course of a research project. As scenario-driven research is itself an iterative process, it seems likely that a corresponding software process is chosen. This leads to the first requirement that can be identified from this viewpoint, the aim to support *change*.

Requirement 3.1: Embrace Change A basic requirement that arises from following an iterative software development process is to appreciate changes and to incorporate these easily into existing system architectures.

For fulfilling this requirement, the integration approach needs to explicitly support mechanisms that are as resilient as possible to changes of system structures. For example, the impact of interface changes on an existing (distributed) system architecture should be minimal to avoid a versioning problem as known from CORBA [SV01]. This also relates to the continuity property of modularity, cf. Chapter 4, which states that the impact of a local change to other components must be limited.

3.2.1. Software Integration as Process

The term *integration* has traditionally been referred to as a single activity within a software development process. Due to the fact that integration in artificial cognitive systems is much more complex than plugging a small number of classes together and its importance in larger projects has recently been well acknowledged, integration nowadays becomes itself a process.

Hence, different models for system integration evolved that can be distinguished into phased and incremental approaches as it is done for the software development processes. Following a phased approach for system-level integration of a software artifact inevitably leads to a "*big bang integration*" where a large number of features are integrated in one huge effort. This procedure is very unlikely to succeed due to the fact that many errors surface simultaneously when new classes or features are combined for the first time. These intricate errors often interact between each other and are therefore extremely hard to localize. What follows is usually a debugging step that quickly turns into a "*system dis-integration*" [McC04] process.

In contrast to a phased integration model and to facilitate scenario-driven research, an incremental approach is just as necessary for the system-level integration as an iterative processes for the software development itself. Incremental integration in general follows a simple pattern that starts with the realization of a small but already functional part of the system. This basis acts as a scaffold that allows the integration of additional elements in the system. Thus, this initial part needs to be thoroughly tested and debugged to provide a stable and correct foundation for integration of further functionality.

Subsequently, the first iterative step is the design, realization and testing of new features for the system. Within scientific projects this is done in the separate research activities. The second step is then to integrate a *single* new feature at a time with the help of the scaffold and test it in combination with the already integrated features. These two steps are iterated until all desired functionalities are integrated and operational.

In order to verify the correct integration of new functionality, a central necessity to effect a feature-driven integration processes is to allow for frequent testing of added software modules. Thus, let us note support for this task as an additional requirement on the level of the integration architecture.

Requirement 3.2: Testing and Evaluation Within scenario-driven research that makes use of an incremental development and integration approach, the testing and evaluation of individual modules must be supported by a suitable software architecture.

Christensen and Crowley [CC94] extend this requirement by considering system-level evaluation support as an essential feature on the level of an integration architecture. For instance, it shall be possible by carrying out offline experiments on the basis of recorded data, yielding a basis for analysis of system dynamics and evaluation experiments.

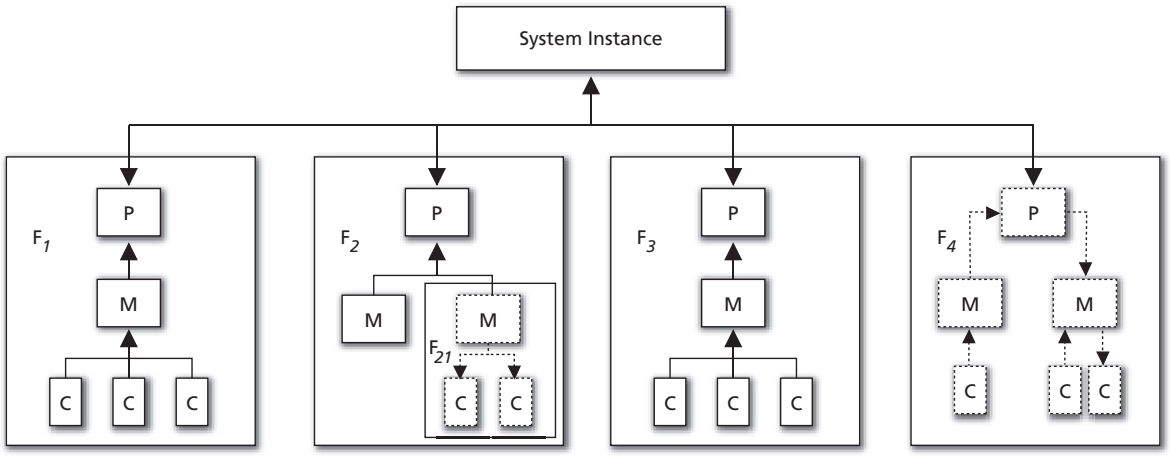


Figure 3.2.: *An example for feature-oriented integration (from [McC04]). An integrated system constructed according to object-oriented design principles is assembled from features (F), processes (P), modules (M) and classes (C). In order to develop the features on different schedules, the integration framework needs to provide the scaffolding for the system and temporary replacements for missing modules or features (shown with dotted borders).*

Exemplary support for this kind of requirement are debugging tools for dynamic introspection of systems at runtime or tools facilitating the recording and simulation of data sources. Within the VAMPIRE project, the synchronized simulation of the video streams gathered from the users head-mounted camera devices in conjunction with the replaying of information about his head pose are examples for this kind of system integration support.

A challenging task in incremental integration is the planning of the integration schedule on the basis of time-based milestones. By the very nature of research projects as explained in the aforementioned paragraphs, it is hard to assess in preface when a component will be available for a first integration in a system. Hence, an integration process is needed that addresses this peculiarity and allows for flexibility with regard to the *sequencing* of integration.

Feature-oriented integration [McC04] (FOI) is an approach that allows exactly this and is therefore well suited for projects carried out according to the scenario-driven research paradigm. Figure 3.2 sketches this process and depicts four exemplary features of a system instance. In object-oriented systems, these features are composed from processes, modules and classes, yielding a decreasing abstraction level. According to FOI, they are integrated one time after another. Features need to be tested in isolation and shall be self-contained to the extent possible. However, to apply this concept, generic scaffolding is necessary, for instance to emulate dependant feature until these are available.

Requirement 3.3: Incremental Development An integration architecture needs to provide re-usable scaffolding that allows functional parts of the system architecture to be iteratively developed and incrementally integrated. To cope with missing parts of functionality, it must be possible to simulate missing components or add mock components that can easily be replaced later on.

Feature-oriented integration allows for a controlled extension of individual functional units that become a visible indicator for project progress and that can directly be applied for improved experimental studies.

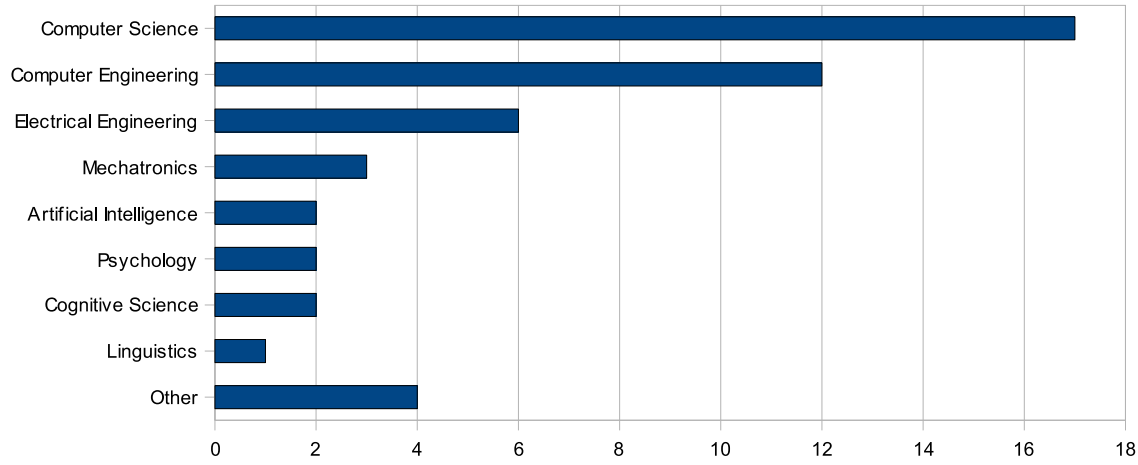


Figure 3.3.: *Education prior to PhD programme. Some participants had an interdisciplinary background, holding degrees in more than one discipline, while 22% of the attendees were persons with no computer science or computer engineering related background.*

3.3. The Social Complexity of Integration

Acknowledging that a fundamental requirement for the envisioned integration framework is to support incremental development processes, we still need to analyze the implications that stem from the people that are conducting research in this integration context and that actually provide the core functionality for the artificial cognitive systems to be constructed: the scientists themselves.

In order to assess the expectations, backgrounds and skills of a typical group of users that are involved in collaborative research projects, we conducted a survey on software integration [WL08] aspects at the COGNIRON winter school on Human-Robot-Interaction [Cog08], which had participants from all over Europe (and one from Korea). It should be noted that the school was organized by the COGNIRON consortium as a dissemination activity for non-members, and correspondingly, there was no project member amongst the 35 participants, which were mainly students in the first years of their PhD period. This group is particularly important as it is usually heavily involved in implementation work but has comparatively little experience.

As already outlined in Section 1.2 a common characteristic of research projects aiming at the construction of cognitive systems for improved human-machine interaction is *interdisciplinarity*. Hence, project teams are often composed of a heterogeneous set of domain experts as underlined by the survey results shown in Figure 3.3. While a large number of the participants has a computer science or computer engineering-related background, many participants did in fact study multiple disciplines, for example, product design, cognitive science, psychology and linguistics. The underlying data also reveals that the fraction of people with no computer science background at all is about 22%.

This is supported by a considerable breadth in the spectrum of reported research areas, see [WL08] for details, encompassing much of the diversity of the HRI field. Taken together, it seems quite appropriate to call HRI a prototypical interdisciplinary area of cognitive systems research. However, the resulting heterogeneity does not reduce but rather increase the need for sophisticated methods and tools that must be known and applied in order to manage a meaningful integration process.

3.3.1. Collaboration and Usability Aspects

The varying levels of proficiency and background knowledge related to software development and integration techniques impose additional challenges for collaborative system development even when merely computer scientists are involved.

This is particularly important because of the stated aim of scenario-driven research to collaborate in terms of building real-world experimental systems. Figure 3.4 suggests that many people actually collaborate even on the software level. After all, even 10% reported to use information from more than five components, indicating considerable integration and corresponding collaboration.

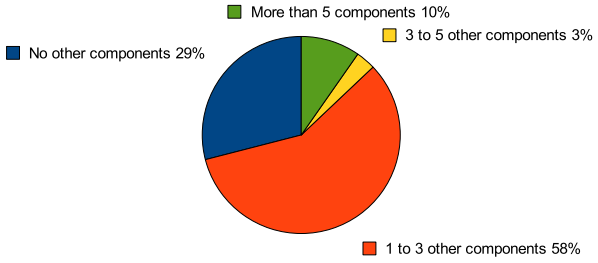


Figure 3.4.: *Number of dependencies. 29% of the attendees reported that their modules do not integrate information from any other components. However, 58% stated that they rely on data from one to three components, while the overall median indicates that modules interact with four other system components.*

One of the most important and most obvious requirement for an integration architecture is that the framework must support collaborative development in an easy-to-use and understandable approach. The survey affirmed the assumption that domain experts in cognitive systems not necessarily are middleware experts. As shown in Figure 3.5 almost half of the participants never used any type of middleware before. Therefore, Martin Fowler's quote "*write programs for people first, computers second*" is particularly important for the design of an integration toolkit in cognitive systems research as the overall goal is to enable researchers *themselves* to provide modules that feature high *integrability*[1].

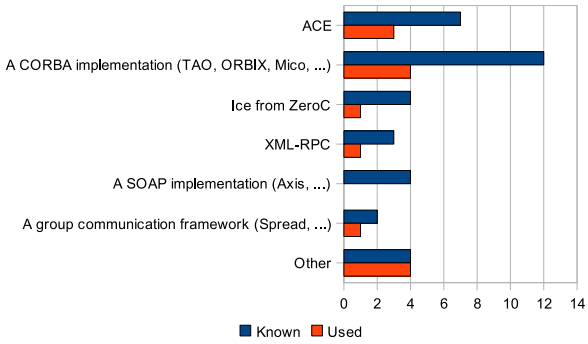


Figure 3.5.: *Use of middleware for distributed systems. In general, there seems to be a large known/use gap exemplified by CORBA that is well-known but seems not to be used regularly. 46% of the participants answered that they did not use any middleware, so far.*

A development process that is purely carried out by explicit software architects may not be eligible in this context with reasonable effort as in cognitive systems research projects system development is intrinsically a joint effort, due to the required amount of interdisciplinary domain knowledge involved. For these reasons and as a general concern, let us note that an integration architecture explicitly needs to take into account usability factors.

Requirement 3.4: Usability This requirement incorporates the goal to design an integration architecture that is not only easy to learn and to use, but equally allows non-expert programmers to efficiently accomplish their desired tasks. Thus, comprehensible abstractions for integration patterns need to be provided with regard to the discovered functional and architectural requirements.

[1] Integrability is the ability to make separately developed components work correctly together in a larger system [BCK05]

While usability is an important quality for the acceptance of a framework, it would be overly optimistic to assume that a software framework, which provides integration on the level of fine-grained operators such as single image filters will get used across different institutions within in a large-scale research project. Almost every research group uses their own toolkits for developing cognitive (vision) functionality. Within the VAMPIRE project, there have been at least five software artifacts involved that belong to this category: Matlab [The08], RAVL [CVS08], Nessy (a toolkit for image processing used by the Neuroinformatics Group at Bielefeld University at that time), the Graz Computer Vision Libaries and IceWing [LWHF06].

While some researchers argue that integration in such a situation is almost impossible [CG06], the practical experiences working in different large-scale projects suggest that this heterogeneity is rather typical, especially for interdisciplinary reseach. This single observation was again underlined by the survey where the spectrum of used robotics toolkits was very diverse. Hence, the viewpoint in this thesis is that an integration framework for cognitive systems *inevitably* has to cope with this diversity. In contrast to the one-size-fits-all paradigm, an integration framework must be able to easily incorporate domain specific tools into its architecture.

Requirement 3.5: Embrace Reuse Within interdisciplinary research or scientific projects that extend over the boundaries of single laboratories it is often hard or impossible due to political or functional reasons to define a single development toolkit for the low-level software development of functionality for a cognitive vision system. Hence, the requirement here is to develop a concept that facilitates re-use by a minimally invasive approach that allows the integration and extension of legacy software frameworks with reasonable effort.

An additional benefit of this approach is that scientists can continue to work in their familiar environments, which shall increase research efficiency. On the downside, the presence of a number of different low-level platforms may increase maintenance efforts and limits to a certain extend the possible outreach of the integration architecture in terms of control over software processes and the level of integration.

3.3.2. Mutual Understanding and Agreement

Even worse than the technological heterogeneity in the software landscape is often the lack of mutual understanding, communicated agreements and social interaction between the members of geographically distributed large-scale research projects. A project with two developers is naturally completely different from a large-scale software project in terms of developer interaction, because of the multiplicatively increasing number of communication paths with every single new developer. Hence, the amount of necessary communication quickly becomes impractical and failures by misunderstanding become propable.

As the actual process of integrating and developing a functional architecture for a system instance inevitable involves communication between human developers, e.g., to discuss interface changes of modules (cf. [WL08]), the stated hypothesis is that collaboration is indeed a software engineering challenge and the integration approach therefore needs to facilitate communication about architectural issues. When developers discuss an architectural aspect, the abstractions of the integration framework should easily be bound to domain entities. The chosen abstractions and techniques shall facilitate the system-level understanding between people from different domains. The goal is to find representations and models that map system models well to the features of the integration architecture like its communication abstractions but are still accessible for all members of a project.

	Monarchical	*Oligarchical*	*Anarchical*
Actors	Exactly one	Dependant	Independent
Modifiability	Not applicable	Coordinated	Uncoordinated
Agreement	Not applicable	Strong	None
Abstraction	Technical, Low-level	Standards-based	Specific, High-level
Technology	Shared Memory, IPC, ...	HTTP, CVML, JAUS, ...	Adapters, ESB, ...

Table 3.1.: *Characteristics and technological implications of monarchical, oligarchical and anarchical situations of software integration.*

From a system's perspective, a long-term vision for the design of integrated cognitive system architectures is to develop a domain-driven design [Eva03] methodology binding models to implementations by employing techniques like *Model-Driven Architecture* [Fra03] (MDA). Nowadays, even the capturing of domain knowledge in robotics, e.g., for the higher-level layers of a cognitive robotics system through a catalog of analysis patterns [Bru07c], architectural description languages [GT07] or through extensions of UML by robotic profiles are areas of current research.

While it is beyond the scope of this thesis to provide a formal descriptive framework for the functional architectures of cognitive systems, a requirement towards these aspects shall be that the system architectures, which are constructed using the resulting framework shall be clearly understandable. Every developer in an interdisciplinary project team should be able to participate in a discussion about system instances and the corresponding functional architectures, for instance by comprehensible models of information that is exchanged in the system. No team member should be hindered by technical peculiarities like firstly learning a specific intermediate language like CORBA's Interface Definition Language [Sie00] (IDL) before she can start thinking about her contribution to an integrated system, which leads us to call for understandable representations:

Requirement 3.6: Understandable Representations Besides being interpretable by computational processes, the representation language should additionally allow for human understandability. Representations that are self-descriptive and accessible to human interpretation are beneficial for the integration process, because they ease communication and contribute to modular understandability.

Questions of modeling, verification and documentation are especially important within this context as the situation for system integration in research projects often exhibits similarities to the anarchical or oligarchical situation of enterprise software integration [Joh02]. Johnson introduced an analogy to political science and compares the context of enterprise software integration to a *monarchical*, *oligarchical* or *anarchical* situations, which are characterized by the properties that are outlined in Table 3.1. Monarchical situations map to small-scale projects where the component developer is also responsible for the integration of a system often utilizing fairly fundamental programming language tools for Inter-Process Communication [ASTMvS02] (IPC). In contrast, the oligarchical situation is largely different because a number of actors, which are somehow institutionally organized and therefore depending on each other are collaborating in a common project. In this situation, modifications are carried out in a coordinated way and the strong agreements between project participants pave the way towards integration. According to Johnson the techniques used in this context are often exploit-

ing domain-specific standards or at least generic standards-based methods like the Hypertext Transfer Protocol [FIG99] (HTTP) for integration. Unfortunately, in the domain of cognitive vision systems, the number of available domain-specific standards is up to now fairly limited, one approach for setting up such a standard is the XML-based Computer Vision Markup Language [LF04] (CVML) for use in Cognitive Vision. A larger number of examples for already existing standards can be found in the area of robotics like the Joint Architecture for Unmanned Systems (JAUS) for the development of air, ground, surface, or underwater systems [Alb00] or the recently evolved Object Management Groups' (OMG) robotics standards [(OM08]. The third type of contexts describes a rather anarchical environment. This situation is characterized by a larger and largely independent set of people working on system(s) to be integrated, yielding largely uncoordinated modifications to whole modules or subsystems in an architecture. Within this situational context, usually concepts on a higher abstraction level, for instance from the domain of Enterprise Application Integration (EAI) are used like sets of specific adapters or in a more recent fashion Enterprise Service Bus [Cha04] (ESB) concepts.

Looking at the integration context of a cognitive systems project like VAMPIRE from this perspective, similarities to the oligarchical situation for software integration can be identified. While researchers usually are willing to participate in a collaboration, which was clearly underlined by the conducted study, the fact that almost no standards are available within this domain and that the amount of coordination between project partners is naturally limited the integration situation can quickly turn into an anarchical one. While the project administration can adapt the collaboration processes within a project to prevent this turn, an integration framework shall additionally support these mechanisms by providing means for more efficiently finding and documenting necessary *agreements*. Hence, an additional goal of the approach developed in this thesis will be on the declarative description of architectural properties like the coordination of modules within a system instance.

Requirement 3.7: Declarative Specification The modeling and communication about architectures for complex cognitive systems requires a description at a high level of abstraction. In order to be able to specify the behavior of the system on an architectural level, a declarative description is needed that specifies the interactions in the possible design space of the provided integration patterns.

The rationale behind this requirement is that strong models that allow validation promote a better understanding and specification on the architectural level, at the same time alleviating some of the innate problems of an oligarchical environment like the stated lack of agreements. Exemplary properties that shall be specifiable are types of component interactions, the coordination strategies in an architecture, the types of exchanged information as well as module interfaces.

3.4. Summary

The software integration task in collaborative cognitive systems research quickly faces problems of programming-in-the-large. Not only with regard to its size, but also in terms of the social complexity involved. If real collaboration is desired, e.g. if a scenario-driven research process is envisioned that emphasizes the integration task, the interdisciplinary and heterogeneous project environment in conjunction with the aim to actually build systems impose unique, partly conflicting, challenges on the design of an integration architecture. The acknowledgment of these specifics in the integration context affects many design decisions of the software architecture that will be introduced in Part II of this thesis for the sake of supporting *collaborative* experimental cognitive systems research.

4. The Technological Perspective

Compared to the project and the collaborative perspectives on software integration addressed in the previous two chapters, the requirement analysis continues within this chapter along a different axis. The fact that integration is rather pointless when it is not actually carried out in the real world imposes some tough challenges for researchers and system developers. These primarily arise from the kind of substrates that are nowadays used to build artificial cognitive systems, which are software and hardware modules.

Researchers need to cope with the technological properties of theses substrates. One exemplary consequence of this fact is that the exploitation of parallelism in distributed systems results in a number of intricate challenges for the scientist acting as a software developer. Within the introduced architectural model, this perspective primarily looks on software architecture from the level of the system architecture. Taking on this viewpoint and a software engineer's mindset, we will look at some of the questions inevitably arising if we aim at developing large-scale distributed software systems.

A fairly large number of textbooks and PhD theses were written in the past solely devoted to the challenges of parallelism and distributed systems. In contrast to these, the subsequent sections only briefly introduce the reader to some of the peculiarities deriving from parallelism. Due to the fact that we already identified the need for a distributed system architecture during the analysis of the project perspective in Chapter 2, the presentation will subsequently elaborate on a number of important challenges that originate from this matter, because the requirement to build a distributed integration architecture proved to be an extremely important and far-reaching issue for the overall approach developed during the course of this thesis project.

Thinking about concurrency, distributed systems and an *intentional* software architecture addressing the requirements identified so far, quickly leads to core aspects of software architecture itself as the architectural development of the integration approach calls for further guidance. Therefore, well-known concepts from software engineering need to be taken into account right from the beginning. Being an equally large field as concurrency and distributed systems, I will particularly address two relationships in the following, critical for the development of an integration approach: the relation between architectural quality and modularity as well as the exploitation but independence of architectural styles for structuring software architectures.

4.1. The Consequences of Parallelism

Fundamental for many artificial cognitive systems is the necessity to run multiple computations in parallel. While this principle could be applied in a serialized manner, instances of artificial cognitive systems usually employ a large amount of true parallelism in order to achieve online performance allowing for their safe operation in the real-world, e.g., imagine a robot that would suspend reading sensor data while replanning due to previous change of environmental conditions.

However, not only for robots but also in cognitive vision, the ability to run several components of the functional architecture in parallel is one of the key requirements as motivated in Chapter 2. Besides the fact that the used algorithms themselves need to be developed and optimized for parallel execution on the functional level, the technological implications of concurrency vastly increases the complexity of the resulting systems and their software development.

Nowadays, computer systems usually feature a single CPU with an ever increasing number of processing cores or are composed of multiple processing nodes in a distributed system connected by some kind of distributed computing architecture. Concurrency in this respect two independent control flows, let alone whether operating systems processes or lightweight threads, appear to be running at the same time. However, often the the high-level tasks these control flows are executing will on a single instruction machine with one CPU be de-composed into a number of possibly interleaved atomic processing instructions. On computer systems featuring multiple processing cores, the control flows will in fact run completely parallel.

Even when the development is carried out in a single programming language, most languages delegate the assignment of execution time to the scheduler of the operating system's kernel. This has at least two important consequences:

1. The order of execution between different flows of control is not guaranteed in preface.

2. Concurrent access to resources by independent control flows, e.g., access to a shared memory region or the same hardware device at the same point in time.

These uncertainties introduce intricate error patterns like stale data or an incorrect orders of updates that need to be addressed by synchronization primitives which prevent other control flows from modifying or accessing a critical memory or code region until the thread that locked the synchronization primitive in first place has finished the protected operation.

Although these problems are understood in theory, only recently solid API's for dealing with common synchronization problems matured from research into main-stream libraries, for instance the new Java concurrency API [GBB+06], which introduces several well-known patterns like Mutexes or Reader-Writer-Locks into the standard API of this modern programming language. Writing portable multithreaded code is even nowadays a challenge for senior software developers and therefore a higher level of abstraction is needed for allowing the *average* programmer to handle these kinds of problems in research software systems.

Besides others, Ceravola and Goerick identified the challenges of parallelism as one of the major points an integration architecture needs to address [CG06] in order to let researchers fully exploit the features of modern hardware setups for the development of cognitive robots.

From personal experience, an additional aspect is that the necessary high-level features for the synchronization of several modules, processes or threads shall not lock-in the programmer into a specific style of programming. This decreases usability and acceptance of an integration architecture. Instead, it must leave software developers the freedom to choose between an asynchronous programming model fully exploiting parallelism which is at the same time more complex to handle and a synchronous, non-multithreaded model that allows fairly simple sequential programming if this suffices in a given situation.

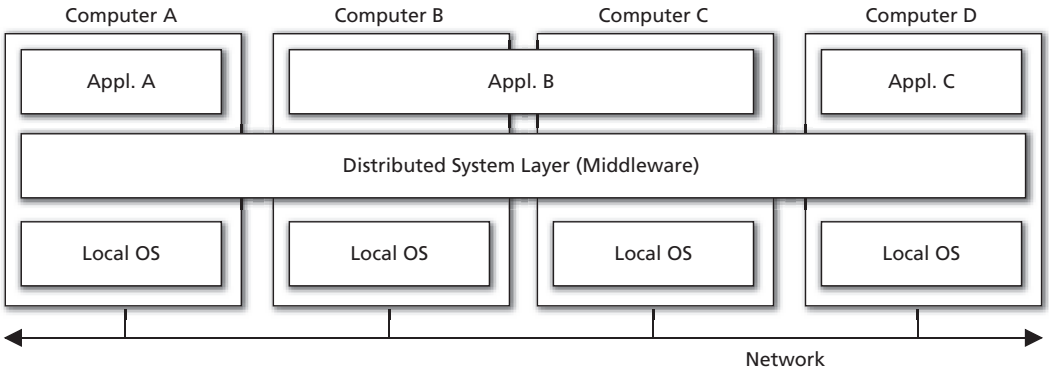


Figure 4.1.: *Schema of a distributed system [ASTMvS02] and the role of middleware services. Middleware provides abstractions for machine-specific low-level operating system resources such as sockets, which are used by applications to exchange data across process boundaries, e.g. over network links. Thus, logically unified applications may be physically distributed over multiple nodes.*

Even large software companies like Microsoft nowadays acknowledge [Gat07] that the design and development of concurrent applications yields one of the major challenges that developers of cognitive systems are exposed to and that problems originating from concurrency are one of the first sources of complexity that hinder robotics research. As a consequence of this observation, Microsoft explicitly addresses these challenges in their recently released *Microsoft Robotics Studio* (MSRS) toolkit, which is an integration software for educative and personal robotics [Jac07]. Its *Concurrency and Coordination Runtime* (CCR) dramatically simplifies parallel and asynchronous programming.

To commence this brief excursion into the threats of parallel computing, let us state the following, rather general but sill important requirement:

Requirement 4.1: Support for Concurrent Processing The integration architecture needs to support means for the development of concurrent software systems, because of the inherent parallelism in the application domain. It should provide higher level abstractions for dealing with these challenges than regular programming language constructs. The resulting programming model must support synchronous or asynchronous use.

4.2. Distributed Systems and Software Integration

Realizing cognitive systems executing multiple computations in parallel often demands for computational power that is beyond the limits of a single standard computer system. Hence, the integration architecture's responsibility is to provide these resources by distributing necessary processing tasks either on multiple CPUs or to multiple processing nodes. The resulting structure of an interconnected process network yields a *distributed system*, cf. Figure 4.1, which can be defined as follows:

Definition 4.1 (Distributed System) *A distributed computing system is a set of computer programs, executing on one or more computers, and coordinating actions by exchanging messages. [Bir05]*

It is noteworthy that many challenges of distributed systems may nowadays also show up if such a process network is solely executed on a *single* multi-processor or multi-core machine.

Despite this observation, many challenge are still specific to distributed systems that are interconnected via some transport layer due to the unreliable characteristics of many types of communication links, the increased latency of interprocess communication and further aspects.

However, as described in Section 2.2.1, the sketch of the VAMPIRE assistance scenario envisioned to distribute as much processing as possible from the mobile setup to off-board computers in order to have the necessary computational power for the cognitive functions available at hand, cf. Requirement 2.8, without letting the users carry bulky hardware around. Hence, the integration architecture presented in this thesis, was designed right from the beginning for network distribution, having to deal with the typical challenges of distributed systems.

In order to access the far-reaching consequences that originate by accepting distributedness as a key requirement, let us consider the intricate aspects of remote interactions in an object-oriented software architecture. In the following, we focus at three aspects - *latency*, *memory access* and *partial failure* - that feature the largest discrepancies compared to local interactions between a set of objects [WWWK97]:

Latency Although not being the most important of the three concerns, the fact that a remote invocation carried out over a wired or wireless medium simply takes orders of magnitude more time to complete, at best around four or five orders [FRF+02], compared to a local method invocation, is the most obvious difference between the two cases. This so-called *latency* of a remote method invocation sums up from the propagation, transmission and processing times needed at the sending and receiving sides of a communication channel.

As we will discuss later on, it is almost impossible to hide this fact from an implementation although techniques like pre-fetching [KS04] can be applied to partly overcome this issue. Unfortunately, these techniques are rather domain-specific and cannot be applied in a general manner. Therefore, an integrated system needs to take into account this difference, particularly for time-critical sub-systems, for instance when active vision techniques are used for visual servoing of robot manipulators.

Concerning these sub-systems an additional issue may arise from (hard) real-time algorithms that need to be scheduled for execution in constant time intervals in order to guarantee a certain algorithmic property. Seriously addressing real-time in distributed systems is an ongoing field of research, which has been addressed for example in the arm control system of the Justin robot at the DLR Oberpfaffenhofen [OEF+06]. Within the VAMPIRE project and additional research projects the resulting integration architecture has so far been applied to, there were no requirements for enforcing real-time properties, so we could safely ignore this issue until the time of writing this thesis.

Memory Access Another important difference between local and remote invocations is the fact that the execution context can change by any invocation to another address space in a completely different processing and language environment. Therefore, every pointer that references an object or data structure in the local address space will instantaneously become invalid when naively transmitted over a network link.

Although modern implementations of *Object Request Brokers* (ORB), for instance the Ice ORB discussed in the next chapter, can overcome this limitation, this approach restricts programmer to use object references for every interaction thereby breaking the transparency that is on the other hand envisioned by such object distribution systems.

Partial Failure Even local interactions of objects are subject to failures for all sorts of reasons. The important difference to errors that occur in remote invocations is that in the case of local invocations, failures are total. Either interactions between objects fail completely or it is possible at least through operating system support to detect an erroneous software or hardware module. Through examination of this error state, it is at least in principle possible to overcome this class or errors. For distributed systems, the concept of *partial failure* is an unfortunate reality, which describes a system state where a number of entities (objects, processes, machines, network links, ...) are in an erroneous state while others are not.

In order to exemplify this, let us think about problematic situations that represent a partial failure [WWWK97] and possible solutions for this while carrying out a remote method invocation by a caller object (the *client*) on a callee object (the *server*) [ASTMvS02]:

1. *Client cannot locate server*: This situation is rather easy to handle, because we just need to report back the exceptional circumstance to the caller.
2. *Client request is lost*: If we can identify by inspecting the local processing state before the crash that the message has been lost within the client subsystem, a straightforward solution is to just resend the message.
3. *Server crashes*: The crash of the server object represents one of the mentioned intricacies, due to the fact that is usually not possible to detect whether the server has already processed the invocation message. In order to handle this, it is necessary to specify the operational semantics of an invocation, for instance *at-least-once* or *at-most-once* semantics.
4. *Server response is lost*: Another example of partial failure is when a response of a server object on an invocation is lost. In this situation, it is impossible to decide whether the server has already processed the invocation. The only solution that is available in this case is to resend the message if and only if the operation is marked as *idempotent*, which means that the invocation is repeatable without any side effects in the server object.
5. *Client crashes*: If the client object crashes and the interaction with the server object is stateful, which means that the server is keeping track of its clients, for instance by holding a transaction lock, these orphan computations are wasting resources that are eventually blocking other distributed processes. Solutions to this class of problems can for instance be to kill the orphans by the client as soon as it is again available or to let the server kill the orphans after a certain period of time.

These failures differ from a simple exception that is raised in a local object interaction as it is usually impossible for a client object to determine whether the source of the problem is a malfunction in a network link or a crash in the server process, thereby leaving the overall distributed system in an inconsistent state. The target object may simply disappear and the thread of control may never return to the calling object. The ultimate consequence of this observation is that partial failure requires the application level programs to account for this indeterminacy.

4.2.1. The Role of Middleware

Novel methodologies for the development and the design of complex distributed systems that take into account the problems introduced above, historically emerged from two different starting points: on the one hand researchers who are trying to extend the model and expressiveness of existing programming languages by incorporating features for building distributed systems. On the other hand, scientists and companies are trying to directly focus on the innate problems of distributed systems by conducting research on advanced communication protocols featuring semantically stronger guarantees for networked interactions resulting in increased reliability [Bir05] or by providing distributed computing environments comprised of code libraries and improved tools explicitly addressing the challenges arising from networked applications. For a number of good reasons, the integration approach in this thesis deals with distributedness in the latter way.

Software that provides this functionality either in terms of language enhancements, specialized tools and protocols or any combination of these is termed *Middleware*. Usually, it provides a connectivity layer and services that allow multiple processes running on one or more machines to communicate across a network or other accessible means of communication as indicated in Figure 4.1. Middleware can be defined as follows:

Definition 4.2 (Middleware) *A set of layers and components that provides reusable common services and network programming mechanisms. Middleware resides on top of an operating system and its protocol stacks but below the structure and functionality of any particular application. [SEI08]*

Although the term middleware is often used in a broad sense, we will stick within this thesis to the primary aim of middleware, which is to provide interoperability between individual applications and software modules across process, platform and hardware boundaries as shown in Figure 4.1. However, middleware can be extended into more versatile software architectures for cognitive systems by integrating generalizable functionality or encoding domain-specific interaction strategies between distributed processes. Hence, it may provide services that are not directly available from the native network layer such as ordering and reliability or add domain support that is directly available over different types of communication media. Examples for both types of middleware and domain-specific approaches will be briefly discussed in the next chapter.

Similar to a certain extent, the development of an integration architecture differs from the design of a middleware in terms of its domain specificity, which is in the context of this thesis defined by what we identified as important in from the project and the collaborative perspectives as well as from its core aim to support the integration process itself, cf. Chapters 2 and 3. Hence, the resulting approach shall *not* compete with fully generic networking middleware but instead be tailored to the aforementioned aspects and may rather provide suitable or novel abstractions adopting methodologies from recent middleware approaches in order to assist users in the problems outlined previously and to fulfill the requirements deriving therefrom.

4.2.2. Requirements of Distributed Systems

Continuing the requirement identification, let us now look in a bit more detail on some of the key characteristics of distributed systems and discuss them in the context of middleware and the perspectives developed in the previous two chapters. Besides *concurrency*, which we already discussed

separately in the previous section, important properties of distributed systems [TB01] are *resource sharing*, *openness*, *scalability*, *transparency*, *fault tolerance* as well as *configurability*, *extensibility* and *security*.

Resource Sharing

A fundamental property of a distributed computing systems is the sharing of resources. In order to share specific resources whether these are free processing time and memory space or in the context of cognitive systems sensors and actuators that allow a system to be embedded in the real world as described in Chapter 2, each resource needs an interface that allows other members of the system to access these.

Within cognitive systems, it is often not sufficient to provide a simple abstraction over the local resource, but more sophisticated arbitration mechanisms are necessary in order to produce meaningful behavior of for instance a navigation module in a robotic system. Due to the fact that distributed arbitration in cognitive systems is still a research topic in its own right, the consequence for the integration architecture from this point is to provide generic functionality to support the arbitration between distributed components, for instance by providing notification services that simplify the development of the necessary arbitration management modules. The resulting requirement from this aspect therefore deals with the possible ways to expose shared resources to the distributed systems. Due to the fact that this is not only a matter of middleware but rather of software architecture, we shall discuss this a more deeply in Section 4.3 in the context of architectural styles for software integration.

Not anticipating the conclusion, supporting arbitration in an integration architecture is one of the challenging areas for future research on integrated cognitive systems. However, Section 7.4 will introduce a method that is tightly integrated with the general concepts of the integration architecture for explicit modeling of arbitration strategies, which has for coordinating access to the sensors and actuators important within the context of the VAMPIRE project.

Openness

Openness in a distributed systems in general deals with the incremental extensibility of these systems. While it is nowadays taken for granted that within one system, extensibility, for instance by adding new components shall be easily feasible if a recent middleware is used, the interaction and integration with services exposed on other systems, irrespective of the underlying software and hardware infrastructure is a more ambitious challenge. Particularly important for achieving a high level of openness are the abstraction from concrete technical environments and comprehensible interface as well as protocol specifications.

The abstraction from low-level software and hardware environments for masking out the inevitable heterogeneity in distributed systems is a key technique to allow for portability of applications. For instance, it should be possible that an application written for a single integration architecture, shall be easily portable to a different operating system, hardware platform and programming language. Protocol and interface specifications are important for openness in order to allow an easy integration with other systems that are developed on a different middleware basis. Therefore, the syntax and semantics of the interface functionality must be very well documented for the provided communication abstractions as well as the protocols that need to be accessible in an open way.

Examples for open middleware systems are architectures such as REST [RTF00], which is based on *Hyper Text Transfer Protocol* (HTTP) as a transport protocol and often uses serialized *Extensible Markup Language* (XML) infosets [CT04] as syntactical basis for exchanged messages. Thereby, REST-based systems achieve a high degree of openness allowing 3rd-party systems to more easily integrate in existing applications.

Concerning the relevancy for an integration architecture, it is natural that openness is an extremely important criterion, because the integration with systems written by others is one of the primary goals as already explained during the discussion of the project perspective. Therefore, we will add openness on our list of requirements:

Requirement 4.2: Openness The ability to integrate additional services and modules as well as being itself integrable with other frameworks by concise definition of the integration interfaces and the used protocols is an important requirement for the overall approach. Ideally, the approach shall be based on well-known standard protocols and techniques that are beneficial for building distributed systems. Additionally, the framework shall support portability in terms of hardware platforms, operating systems and programming languages.

Because within cognitive systems as the one that are in the focus of the VAMPIRE project, the question of how to flexibly represent data, information and knowledge is crucial in the overall architecture, it is important to stress that this is one of the conjunctions where functional and technical architecture meet and mutually extend each other.

Therefore, we will extend the call for extensible and understandable representations, cf. Requirements 2.6 and 3.6, by an additional requirement, which is that exchanged representations are not only interoperable, which is covered by openness in general, but also *interpretable*.

In order to assess the benefits of a representation that is interpretable by other modules without prior knowledge about the contained structures, let us shortly discuss the reverse situation. Imagine a data visualization component for a cognitive system acting as a common service for supporting the integration process. If the representations used for message exchange are not self descriptive and can not be interpreted by the machine on its own, the visualization service unnecessarily needs to be aware of many if not all the interface and datatype definitions that are used within a system, thereby increasing the software coupling and maintenance efforts. In order to prevent such situations we therefore call for interpretable representations:

Requirement 4.3: Interpretable Representations In addition to the benefits that openness provides for an integration architecture, the exchanged data items need to be self-descriptive and dynamically interpretable by components that feature little or no knowledge about the exchanged data types.

Considering this requirement in the design of an integration architecture shall later yield in a decreased coupling of the resulting software components. This requirement is fundamental to the approach described in this thesis. Therefore we shall revisit this topic recurrently though subsequent sections. Low coupling eases the design of re-usable building blocks that provide common functionality on top or within an integration architecture like notification, logging, proxy or visualization services. In order to achieve interpretable representations, some kind of self-descriptive formal language like SGML [GR00], XML or similar techniques may be applicable.

Scalability

Scale is an important goal for the design of distributed systems, because it shall guarantee that a system can grow and is extensible for an increased processing load, for instance take into account the exponential growth of queries on the root servers for the internet's Domain Name Service (DNS) during the 1990s [Bir05] that lead to several internet brownouts. In general, *scalability* in distributed systems can be defined as follows:

Definition 4.3 (Scalability) *A system is said to be scalable if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity. [Neu94]*

Even though the usual meanings of scalability like increasing a user base from one thousand to one million without breaking quality-of-service (QoS) guarantees is rather not in scope of the cognitive systems to be developed, scalability questions nevertheless can become important.

Adding a component such as a global monitoring service that integrates with many other components in an existing system architecture triggers additional concerns about extensibility but imposes to the same extent questions of scalability. Whether or not a system is capable of handling the increased data flow and how it copes with the additional interactions is often unpredictable. Thus, let us consider the three distinct dimensions [Neu94] that comprise scalability. Each of these influences the challenge of building a scalable integration architecture in different ways:

Size scalability needs to be supported in that it must be possible to dynamically add components to a developing system architecture without suffering performance loss and to improve the system performance if a service is properly replicated. Although it for instance never occurred within the VAMPIRE systems that a single source of information published its information to more than 50 receiving modules, size scalability becomes important for cognitive systems with regard to the dynamics of latency, throughput or robustness when additional services are integrated. As a general requirement scalability is often claimed to be supported by middleware solutions but it is usually not proved. Therefore, it will be necessary for a proper evaluation to test the integration approach in meaningful scenarios relevant for the application domain.

Geographical scalability is less important for the work carried out in this thesis as the goal is to build networked systems like an assistance system that features a high degree of local cohesion, which is usually integrated within one local area network (LAN). The question of geographical scalability needs to be addressed differently when the network distance between participating computers is very large, for instance when scalability on a global level is needed.

That said, it is tempting to apply broadcasting or IP multicast protocols on the transport layer of LANs in order to ease the design of scalable applications. Unfortunately, this decision will turn out as an over-simplification with regard to the given integration context outlined in Chapter 3. Even in the simple case of performing integration within a single University network it is rather usual that the systems which are set up cross the boundaries of single LANs. Therefore, a simple network broadcast is no longer possible and the correct handling of IP multicast packets is no longer guaranteed if one is not in control of the network infrastructure. Thus, although geographical scalability is not of prime importance, certain aspects still need to be taken into account.

Transparency	*Aim and Description of Transparency Level*	*Relevance*
Access	Hides platform or language specific details in data representations and differences in local or remote invocation mechanisms.	o
Location	Hides the physical location of a component, for instance on which node a service runs in a distributed system.	++
Migration	Hides the movement of a component from one location to another.	+
Relocation	Hides the fact that a system can change the location of a service to which a client is bound at runtime.	-
Concurrency	Hides from a client the concurrent access to services by other clients.	++
Failure	Hides failures like temporary disconnection from services by applying recovery strategies without client involvement.	o

Table 4.1.: *Levels of transparency in a distributed systems [ASTMvS02] and relevance for the integration approach (*++ important, + desirable, o neutral, - less important*).*

Administrative scalability is concerned with scale in terms of the number of organizations involved in the operation of a distributed system and deals for instance with security matters. It may be of interest for very large-scale integration projects with a large number of involved organizations but is of less importance for the integration context of this thesis.

Concluding, it is important to note that scalability is an important attribute of an integration architecture for cognitive systems, particularly when systems are dynamically evolving, which is the favored development approach for software integration in the given environment as outlined in the description of the collaborative perspective in Chapter 3. The following requirement shall summarize the most important points of scalability from this viewpoint:

Requirement 4.4: Scalability An integration architecture supporting the development of scalable cognitive systems needs to provide at least two properties of scalability. On the one hand, scalability is provided if an increased number of components reasonable for the given domain performing different tasks do not degrade overall system performance. Conversely, the overall system performance shall scale up if the processing of a single component is distributed to a number of different components. Additionally, geographical scalability must be supported at least in terms of addressing the needs of the integration environment.

Transparency

Transparency in a distributed system describes the degree to which the differences between a local and a remote interaction are masked out, e.g., for users working with an application on the system level or software developers utilizing a middleware in a specific programming language. While even more dimensions of transparency have been defined in the literature, Table 4.4 summarizes the most important dimensions of distribution transparency [ASTMvS02] and their relevance to the integration approach in the context of its network functionality. Aiming at full transparency in every aspect is an extremely hard to achieve goal - although incorrectly claimed by many middleware products - that may not even be well worth doing so in every case.

So, what are the relative merits of the different levels of distribution transparency in the context of an integration architecture?

Access transparency aims at masking out the heterogeneity in different software and hardware platforms used in a distributed system, e.g., in terms of byte ordering or language specific representations as well as for instance with regard to the question how procedures are invoked. Aiming at a very high degree of access transparency shall allow software developers to design programs that can be easily broken up into smaller parts when distribution is needed. This shall reduce the complexity in the design of distributed systems. Unfortunately, with regard to the inevitable differences between local and remote interactions in latency, memory access and the existence of partial failure as outlined in the beginning of this section, transparency in terms of, e.g., memory access and method invocation can never be complete [WWWK97].

A high degree of access transparency masks the important fact that a distributed operation is possibly carried out with the explained different characteristics. Just merging a CORBA implementation in an existing object-oriented architecture to improve a performance gain by distribution is very likely to result in an errant architecture [FRF+02] possibly with the same or almost no improved performance. Even worse, errant architectures often exhibit a brittle, tightly coupled system architecture that is hard to maintain and evolve on the longer run and ever harder to understand and test. In contrast, the system design needs to reflect the distributedness through an explicit decomposition strategy, for instance by choosing a suitable interface granularity [FRF+02] at the distribution boundaries.

While certain features of access transparency, for instance to overcome different memory representations are valuable and shall be addressed, full access transparency is for this reasons not of prior importance for the approach to be developed within this thesis. The aim in this thesis is to deal with the inevitable challenges of distributedness to the extent necessary and to reduce the essential complexity in solving those within the given environment.

Compared to access transparency, *location* and *migration transparency* are essential requirements within distributed systems. If these dimensions are supported, the concrete physical location of a component is masked out, for instance by assigning a symbolic name to a shared resource. Consider a fully qualified domain name (FQDN) specifying a host computer in the internet over the equivalent IP address of the machine as an example. While the FQDN hides the concrete location of a machine on the network, the IP address directly reflects the network structure. From a practical perspective, this requirement is important for maintainability of complex system setups as the involved node and their network setup may change any time. Embracing location and migration transparency, location independence is an important requirement in general and also for the work carried out in this thesis.

Requirement 4.5: Location Independence The component developers must not be aware of the physical location of another service they utilize in a concrete system setup. Therefore, the integration architecture needs to provide and use an abstraction strategy that masks out physical locations from the system developers. It must not be necessary to change the code base of a service if the physical locations of other components change.

As an extension to migration transparency, *relocation transparency* hides the dynamic movement of components to other physical locations at runtime from other dependant services. While this may be an important property for distributed architectures in general, this requirement did not show up during the VAMPIRE project as services needed not to change their execution context at runtime.

The requirements arising from the parallel utilization of shared resources are much the same as what we already discussed in Section 4.1 except the additional claim that the services utilizing others shall not be aware of this fact and therefore operate in a loosely coupled manner. Thus, achieving *concurrency transparency* and supporting the developers in coping with the challenges therefrom is an important aspect and already on the list of requirements for the integration architecture.

Due to the fact that the last aspect of transparency listed in Table 4.4, *failure transparency* is highly intertwined with the larger concerns of fault tolerance, let us discuss this now separately in the context of this more general characteristic of a distributed system.

Fault Tolerance

Owing to the effects of partial failure and the existence of Byzantine errors due to the partly prototypical status of research software that is integrated in the envisioned systems, fault tolerance needs to be taken into account in the design of an integration architecture. Fault tolerance in a distributed system is comprised of availability, reliability, safety and maintainability [ASTMvS02].

An individual *failure* is a single cause out of a potential larger number of failures that is responsible for the temporary failing of a distributed system. This may imply that all or parts of the regular functionality are not *available* to human users or other system components. Within this context, the difference between availability and reliability is important, because a highly available system need not necessarily be extremely reliable.

Fault tolerance and failure transparency deal with error handling techniques in order to increase both properties. While safely is naturally extremely important for instance in control systems in the broader context of machine automation, safety can become an equally important concern in the context of cognitive systems that are autonomously acting in the real world like mobile robots. Maintenance is important in order to allow for autonomous recovery without human intervention and is therefore equally important in the application domain once such systems are leaving the experimental state and non-expert users become dependant.

Failures can be classified according to their occurrence frequency, e.g. temporary, periodic, permant and their characteristics. For instance, *Byzantine* failures that may leave components in an undefined state are often the result of erroneous software components and are hard to track down. Furthermore, there are simple *halting* errors where a software component aborts and is no longer available or *fail-stop* failures where a component cleanly aborts by being able to previously notify the rest of the system of its malfunctional state. Besides these, a number of additional models of failures have been defined in the literature, e.g., [ASTMvS02] that need to be treated on different levels of abstraction and with specialized methodologies.

As we cannot assume user code to perform correctly, the integration architecture needs to detect and notify the halting as well as fail-stop errors without requiring modifications of user code. The inspection of Byzantine errors needs to be supported by a good instrumentation within the resulting framework to track the state of affairs between involved software components.

In order to overcome the effects of partial failure different techniques can be applied that often impose further limitations, e.g. non-standard extensions to network protocols or other constraints [Bir05] that are not generally applicable, for instance annotating a method as *idempotent*, which conveys the information that a method call can be repeated in case of a previous error without side effects.

In any case, it is important for software developers utilizing the integration architecture to know the exact semantics of a provided functionality, e.g., whether point-to-point interactions feature at-most-once or best-effort semantics and if in broadcast communication the ordering of the messages is preserved in receiving processes. If this information is not available, both the development process of a system can be cumbersome as well as it will be hard to come up with highly reliable system in the end.

Concerning the level of error transparency that can be achieved in a distributed system, let us note that it is theoretically and practically infeasible to totally mask out failure in such environments [WWWK97]. Thus, the concept followed within this thesis is that errors are an exceptional but regular system state that needs to be made explicit by corresponding mechanisms to be exposed by the integration architecture to software developers in order to handle errors in application specific ways on a higher level of abstraction. Thus, error transparency is not a primary goal of the integration architecture. Providing a high degree of error transparency in development situations may make matters even worse, e.g. by masking out coding errors, thereby ignoring the principles of defensive programming [Lad94].

Requirement 4.6: Error Handling In contrast to providing a high level of error transparency, an integration architecture shall explicitly accept errors as a central reality and provide functionalities that allow to assess the possible cause of failure, actively handle them and eventually recover from them at runtime by reconstruction of a corrected system state without human intervention.

The questions how fault tolerance and error transparency go along with the software design of the resulting systems or components, provides a starting point for the subsequent section on software architecture in the context of the envisioned distributed system.

4.3. The Relevance of Architecture

Complexity in software development usually arises from two different sources. On the one hand there is unavoidable *inherent* complexity in the domain of cognitive and distributed systems. This type of complexity needs to be handled on the level of the functional or integration architecture of the resulting software system.

On the other hand, *accidental* complexity [Bro95] is non-essential for solving the functional challenges. This type of complexity may emerge in this integration context from heterogeneous development processes, a varying level of cooperativeness between project partners and from architectural mismatch between sub-systems. To cope with the resulting overall complexity and the technological peculiarities of parallel and distributed systems, a coherent integration architecture is critical.

The goal of architecture-centric software and systems development is to raise the level of abstraction in order to reduce the overall problem complexity. In the following, we shall discuss what additional requirements origin from this perspective for the envisioned integration approach, particularly why modularity is on a more abstract level the most important factor for reducing overall complexity. Linked to the aspects of modularity, the reasons for aiming at a loosely coupled service approach within this work will be explained. To commence this chapter, questions of architectural style will be discussed in the context of an integration architecture.

4.3.1. Modularity as a Key to Software Quality

Modularity is a fundamental principles that is often found in complex technical and organizational structures as well as in nature. Looking at the definition of a complex system, one approach to deal with the challenges arising therefrom is to reduce the number of distinct parts by aggregating elements into more coarse grained subsystems. As a consequence, the aim is to build larger systems by combining these *modules* and focusing at the interactions between them on a higher abstraction level than one would otherwise do. Decomposing a large number of intertwined elements into a smaller set of less interdependant modules is the general aim of modular software development approaches.

Parnas introduced the classical definition of modularity and the concept of information hiding already in the Seventies as he states that every module "*is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings*" [Par72].

While the ideas of Parnas and others are still influential for the design of programming languages, it will be necessary for the envisioned integration approach to allow for modularity and reduced interdependencies on a higher level of abstraction than that of programming language elements. Examples for more abstract information important on this level are what functional roles and interdependencies an individual module, what interface specifications it adheres to or what rules prescribing the development and integration it needs to conform to.

In order to assess modularity in the context of software architecture, let us take on the perspective that Bertrand Meyer introduced [Mey97]. He gives a more detailed description for the notion of modularity that can be summarized by the following six properties of a modular approach, which we will adopt and consider as refinements for the call to follow a modular approach as stated in Requirement 2.1:

- *Modular Decomposability*: A software architecture yields modular decomposability when it facilitates the comprehensible decomposition of a problem into a smaller number of easier subproblems that are still manageable by the integration architecture. In order to satisfy this constraint, the resulting partitioning should allow for independent, parallel development and interconnection through a structure as simple as possible.

- *Modular Composability*: This property of modularity relates to reusability of already existing building blocks of functionality in different contexts. An architecture satisfying composability shall allow to freely integrate existing building blocks in novel applications that were not foreseen during the initial development of these modules.

- *Modular Understandability*: If a software favors modular understandability, it shall be comparatively easy for humans to understand the system-level functionality of a software module. It needs to be possible to understand the interactions of a module without getting to know all other modules within an architecture. In order to allow for better understandability, our assumption is that traceability of the dynamic behavior of modules is an equally important requirement.

- *Modular Continuity*: The aim of this property is to limit the impact of change. A software architecture conforms to modular continuity when a change in one of the domain modules yields only a minimal number of changes in other modules. This property poses questions of versioning and backward compatibility of interfaces.

- *Modular Protection*: While continuity is concerned with the impact of change, this property is concerned with the impact of failure. It states that the number of modules of an architecture that are affected by an abnormal condition within one module shall be minimal. This challenge relates to the question of combined critical dependencies introduced previously and the aim to achieve a high degree of robustness.

These attributes need to be carefully considered when designing a software architecture as they provide a fundamental basis to allow problem decomposition on the level of the integration and the functional architecture. Even so, the question how modularity can be achieved in the reality of larger systems must not be answered in this analysis but in later chapters of this thesis. However, an architectural property that is closely related to the attributes of modularity is the degree of coupling that a system exhibits at different levels of abstraction.

4.3.2. Software Coupling and Granularity

Coupling in a software system is a "*the strength of association established by a connection from one module to another*" [SMC99]. While a minimal amount of coupling is needed for a modularized system to perform a meaningful task, *tight* and *loose coupling* define the extreme ends of this continuum. From the perspective of software architecture, coupling can be interpreted as the fragility exposed by module interdependencies.

Tight coupling induces a high fragility in the relation between a number of components, which often implies negative effects on the aspects of modularity explained above. Aiming at loose coupling in contrast, acknowledges the benefits of a modular approach while granting the fact that a problem may not necessarily be fully decomposable in a modular manner, for instance due to performance requirements. Realizing a loosely coupled architecture focuses on a reduction of the number of external interdependencies between modules, hiding internal parameters that are only important for the concrete implementation behind the module's interface. Loose coupling is particularly beneficial given the characteristics of the integration context, e.g., to effect independent component development and improved changeability of individual modules.

Coupling can be induced along several orthogonal dimensions [Fai06]. The critical source of coupling for the context of this work is coupling on the component interface level. This type of coupling is comprised of aspects like data and interface formats as well as granularity, version resilience, transport independence and the granularity of expected interaction patterns. Furthermore, stateful interactions and implicit as well as explicit correlations, the ability to mediate data between components through proxies or routers and dynamicity are concerns that influence the degree of interface coupling.

Acknowledging the benefits of loose coupling and the desired abstraction level of an integration architecture, the question of interface granularity quickly arises. Considering this aspect in the context of software integration, boils down to the question of what abstraction level shall be targeted when modules in the functional level of the system's architecture are to be integrated with the services an integration approach provides.

From the viewpoint taken on in this thesis, the assumption is that it is beneficial to focus on rather *coarse-grained* component interfaces for two reasons. Firstly, a decomposition into modules that offer their functions through a coarse-grained interface forces module developers to design system-level interactions on a higher level of abstraction, which reduces overall complexity.

Secondly, the introduction of coarse-grained interfaces has positive impacts on the overall performance in distributed systems as fewer message exchanges between components are necessary. Furthermore, this strategy promotes modular protection, reuse and changeability, e.g., because coarse grained interfaces at the same time exhibit smaller interfaces with fewer methods.

Another argument in favor of coarse grained components for integration within a loosely coupled architecture is that some fine-grained modules need to be more tightly clustered within a larger building block to cope with specific requirements, e.g. real-time guarantees for the visual servoing of a robotic arm or control algorithms that need to be executed with fixed timing intervals.

If these building blocks are encapsulated in coarse-grained component interfaces, they can still be integrated without inducing unnecessary complexity on the overall integration what would be the case if, for instance, real-time would be a first class requirement. For the reasons outlined in this chapter taking into account the integration environment and the distributedness requirement, we state that a suitable integration approach needs to support the development and integration of loosely coupled software modules:

Requirement 4.7: Loose Coupling The integration architecture shall primarily promote loose coupling on the interface level for the services it provides, thereby allowing for reduced coupling on the level of the functional components. The goals are to foster independent development and evolution of components as well as to increase testability, which are all important software qualities in the context of the given integration context.

4.3.3. Architectural Styles and Software Integration

An important aspect for finding dependable solutions to the challenges described previously and particularly in the context of distributed systems and loose coupling is the concept of *architectural style*. Similar to *idioms* on the level of programming languages [Cop91, Lan01] and *design patterns* in object-oriented software [GHJV95], architectural styles encode design decisions which are applied to the construction of systems [SC97]. Each style promotes qualities that are of special interest under certain conditions. For instance, the C2 style, depicted in Figure 4.2, was developed with a particular focus on the development of GUI systems [TMA+95]. Architectural styles are a critical concept in achieving reuse of structures, relations and interactions on a high abstraction level.

According to this understanding, architectural styles are describing the structural organization and interaction of software entities, remaining completely independent of specific domains. Examples for these domain-independent architectural styles are *pipe-and-filter*, *request-reply*, *shared repository* or *object-request-broker* as well as many more [SG96, AZ05]. These styles are extremely important in order to describe, model and communicate the high-level structures of software architectures. While the border lines between patterns and styles are yet unclear, it is nowadays very well recommended practice to design and describe software systems using these building blocks. Specifying and applying well-known architectural styles is not only useful for the realization of an integration framework itself but is particularly beneficial for the services it provides itself to its clients.

Styles that map to software patterns allow to codify proved solutions that encapsulate large fractions of the accidental complexity in distributed component interaction. An integration approach featuring an *intentional* architecture, shall make the rules governing the composition of design elements explicit and document them in one or more architectural styles or system-level patterns.

The quality of resulting system instances and software architectures can greatly benefit from distinguished styles that are followed precisely during system construction at the same time making it easier to communicate about the structural aspects of a larger system.

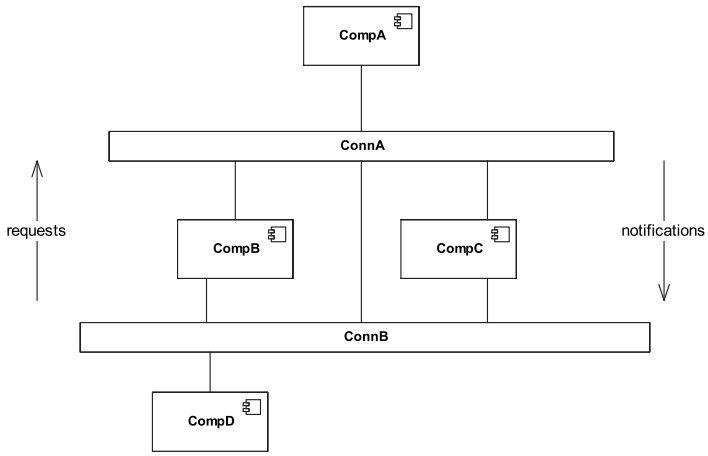


Figure 4.2.: *Exemplary C2 architecture with four components in three layers and two connectors, which actually isolate higher-level from lower-level components [AZ05].*

As functional and technological dimensions are not orthogonal, it will be important to provide a set of architectural styles or patterns on a system level that matches well with the functional and non-functional requirements identified in the previous chapters. However, it is impossible to assess in preface possible future styles needed in an integration architecture and as it is hard to assess at the beginning of a project like VAMPIRE which architectural styles will be best suited for integration, the focus of this work with regard to architectural style is to not be fixed on a single style but rather provide an extensible, open software architecture providing building blocks for easy extension, yielding a further requirement mainly derived from the technological perspective:

Requirement 4.8: Architecture Extensibility Integration architectures must allow for extension of their functionality in terms of provided styles with a reasonable effort, e.g. by offering predefined extension points.

4.4. Summary

The main matter of this chapter was to detail some of the technological challenges involved and suggest strategies for reducing their impact, with a particular focus on parallel and distributed computing. The goal here is to encapsulate the accidental complexity without constraining architectural choice for system developers. The previous sections described requirements that have to be fulfilled in an integration architecture for experimental cognitive systems to realize this.

Following a modular approach is a key requirement for integration and directly leads to the question of interface design and coupling. While a certain amount of coupling between modules is inevitable, loose coupling is especially important here due to the collaborative nature of software development in research projects as described in Chapter 3. To a certain extent, the aim of encapsulating complexity and providing guidance for the design of distributed software architectures conflicts with the required freedom of architectural choices. Hence, the outlined approach strives at a balance which hides accidental complexity while exposing inherent complexity.

5. Requirements and Architectures for Integration of Cognitive Systems

Throughout the past decade, several frameworks for vision systems have been proposed [FJK+96, PUV03, DKRH94, KR94, LWT94]. Most of these frameworks were tailored to certain project specific requirements and thus are of limited applicability for the more general challenges of cognitive systems. However, there are common needs in traditional computer vision as well as in cognitive systems which this thesis tries to address in order to provide a suitable software integration approach that increases the efficiency of research aiming at real-world prototype systems. In the previous chapters, these needs were identified through consideration of three perspectives for a holistic integration approach in this context: the project, the collaborative and the technological perspective. This presentation may suggest that these perspectives are orthogonal. However, this is fortunately not true. The aim of this chapter is to provide a condensed set of key aspects representing the essential thematic priorities within this thesis, thus laying a basis for comparing selected related work against these general requirements. Additionally, this will provide an avenue for the transition to Part two of this thesis that presents the approach introduced with this dissertation.

The first section of this chapter further analyzes the identified requirements by clustering them into an evaluation scheme. This is subsequently used for estimating the strengths and weaknesses of other approaches compared to the key aspects important within this thesis. This catalogue extends a report that prepared for the EC Vision network of excellence [PVWB04] and presented at ICPR04 [WPB+04]. It was requested due to the fact that only less comparative and not very recent reports [CC94, RRH99] were available at that time. While this report mainly focused on computer vision toolkits, the analysis in this section is geared towards approaches to software integration, explicitly considering approaches supporting cognitive vision and robotics systems.

The resulting compact catalogue of the most important aspects and requirements shall provide an avenue for a brief introduction to related research activities from the domains of cognitive systems science. Subsequently, three exemplary architectures ranging from object-oriented middleware and cognitive vision research to an integration architecture for cognitive robotics will be reviewed along the identified criteria. The chapter ends with a short discussion and brief conclusion of Part I, describing how the existing approaches differ from the properties one would expect for the envisioned integration approach that is presented in the second part of this thesis.

5.1. Synopsis of Requirements

The aim of this section is to develop a conceptual framework for comparing existing related work against the requirements identified in the previous chapters, similar to what was done for agent architectures [EM02] or mobile robotics [Ore99, KS07], but constantly keeping in mind the three perspectives explained before.

Requirements engineering usually separates individual requirements into so-called *functional* and *non-functional* requirements, which has not been done for the requirements identified so far. While functional requirements (FR) specify "*a function that a system [. . .] must be able to perform*"[IEE90], the latter are less clearly defined. Within this thesis, we will adopt the definition of non-functional requirements (NFR) according to Sommerville [SK98] as a specification of cross-cutting system aspects placing restrictions on the artifact to be developed, the development process or representing external constraints that must be considered. Thus, the first step in analyzing the discovered requirements is to assign them to either one of these two categories.

As a second step, each individual requirement that *should* be taken care of is assigned to one larger aspect representing an aggregated requirement which *must* be addressed in some way by an integration approach applicable for conducting research on cognitive (vision) systems as in the VAMPIRE project.

Figure 5.1 depicts the results of this process. Each key aspect originates either directly from a previously discovered requirement like the need for a *distribution infrastructure* or is denoted by a newly introduced term that better describes its aim as a whole like the ambition to support *software engineering* methodologies. Furthermore, requirements were adapted or broadened, for instance the computer vision requirement. Looking from the more general perspective of cognitive systems, the relevant question is whether candidate approaches provide some way of support for the functions needed in one of the domains of cognitive systems research, not solely being restricted to computer vision. In addition to discussing the aspects introduced subsequently, I will shortly argue why, e.g., a rather typical requirement such as security is of less importance in the given context and explain the resulting scheme used for subsequent assessment of related approaches.

5.1.1. Functional Aspects

The functional requirements that are motivated by the idea of a visual active memory as well as the interactions within such an architecture yield a set of key aspects an integration solution must support. These aspects are *Data Representation*, *Information Management*, *Distribution Infrastructure*, *Adaptive Coordination* and *Domain Support* with regard to computer vision functionality. The considerations behind each of these key aspects are the following:

Data Representation Finding a common representation for the data processed in a visual active memory was considered important from the very beginning of the project. One of the fundamental ideas about the functional architecture of a VAM system is to let various interpretation processes operate directly on a set of shared information in order to build a history of the visual events in its environment. While this idea is conceptually close to the well known style of blackboard architectures [SG96], which we shall revisit later in this thesis, it is considerably different from procedural integration where services invoke each other directly via their specific interfaces.

Acknowledging the importance of shared data representation it becomes clear that the chosen type of representation must be extensible (Req. 2.6) in order to cope with the data variability inherent to cognitive systems, for instance the different types of multi-modal sensor information like visual and audio information as well as more abstract information processed in a VAM system. Therefore, the data types an integration architecture offers must themselves be extensible, e.g., by using service or data definition languages.

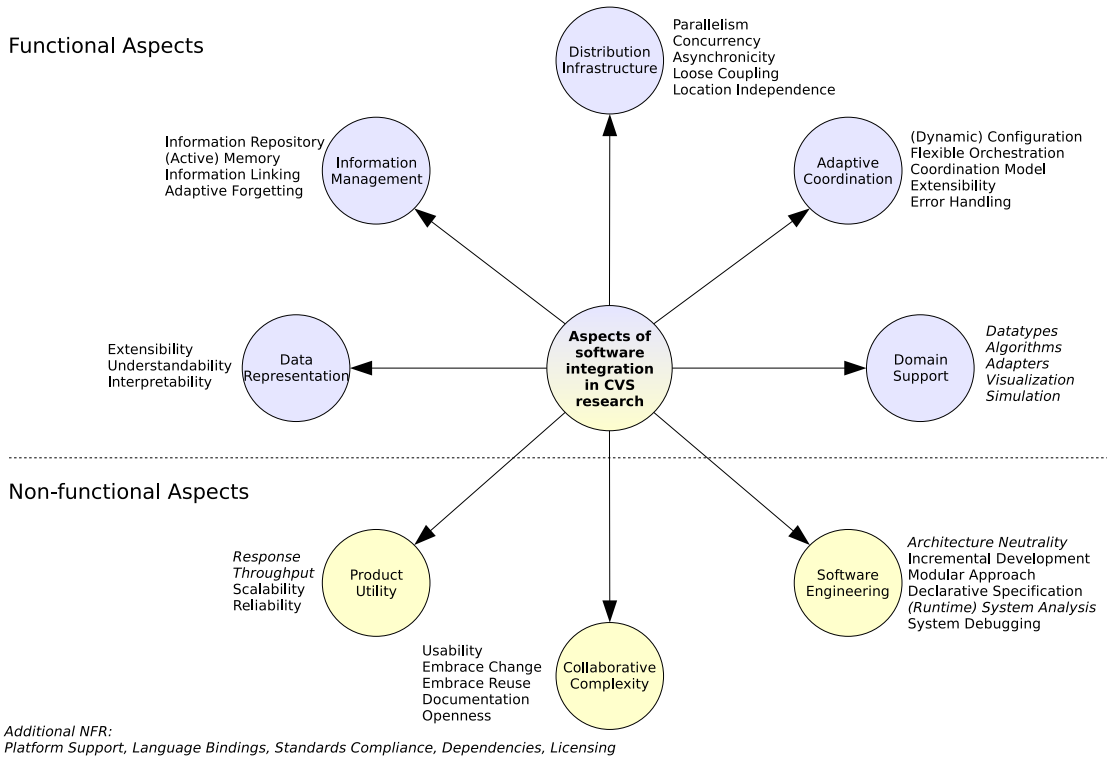


Figure 5.1.: *Key aspects for software integration in cognitive systems research projects consituted by aggregation of related requirements identified in the previous chapters.*

Moreover, representations should be interpretable (Req. 4.3) by software modules without knowledge of the source and types as well as ideally understandable by humans (Req. 3.6) to facilitate communication between system developers.

Concerning other functional requirements for the integration approach, questions of data representation are usually linked to the data formats used for service interaction within a distributed systems architecture and can affect the coupling between components, e.g., when comparing document-oriented interactions versus programming language-oriented parameter passing. Additionally, the induced overhead of a specific data representation naturally has implications on the performance of the overall integration approach. Last but not least, the chosen representations should be accessible for external implementations to offer a certain level of openness.

Information Management The memory metaphor is an important conceptual cornerstone of the VAMPIRE project's vision and for cognitive vision systems (Req. 2.7). Thus, an integration approach must feature substantial support for an active management of multi-modal information, providing a robust basis for cognitively motivated architectures.

Beyond the application of potentially suitable state-of-the-art data management technology like active databases or techniques like distributed shared memory, an integration architecture needs to provide additional means for an active self-management of the information contained within the repository (Req. 2.10), induced for instance by the idea to model a forgetting process (Req. 2.11) that maintains

consistency and performs an automatic garbage collection within the stored multi-modal information. In traditional databases, similar functionality is often realized with rule-based triggers [DBM88] and stored procedures. In addition to these features, it should be possible to provide flexible means for the interlinking of multi-modal information by inserting references between elements within the different memory instances.

Looking from a technological perspective, this aspect is also related to the chosen data representation as it can become cumbersome to provide the necessary conversions if the data model of the information management service does not match the one chosen for the data representation. Ideally, both models shall be equal or shall complement each other in order to let the information management system directly operate on model instances. To the same extent as for the data representation, it is necessary to consider the question of extensibility with regard to the large variability of data types that need to be handled by the information management architecture. Furthermore, the information management aspect needs to be integrated seamlessly with the functions and patterns provided by the distributed processing architecture in order to be accessible throughout a networked system and to improve usability factors.

Distribution Infrastructure In order to support the functional architecture with the necessary computational resources and account for the inherent parallelism as a fundamental requirement of cognitive systems (Req. 2.2), an integration architecture must offer the possibility to distribute multiple computations in parallel over standard networks (Req. 2.8).

While a general purpose middleware has to deal with a larger amount of requirements, still several of the identified requirements contribute to this aspect:

- Asynchronous communication patterns (Req. 2.12)
- Support for inherent concurrency issues (Req. 4.1)
- Partial distribution transparency, at least location independence (Req. 4.5)
- Preference for a loosely coupled distributed architecture (Req. 4.7)

Looking at the interdependencies with other aspects, let us first note that we can identify a conflict with the usability requirement: once a distributed architecture is envisioned, the level of expertise that is needed by software developers due to intricate technological implications raises. Thus, it is not only important *what* methods for parallel processing are offered, but also *how* these are provided and how they affect the overall usability. For instance, ACE [SH01] allows users to provide per-thread memory allocators to reduce latency induced by per-request memory allocation within the marshaling step of the network input/output, but this shifts the burden of a correct implementation of these allocators to the end users. Other functional aspects related to the question of distribution are representation and information management as well as adaptive coordination of parallel processes if carried out across network boundaries.

Avoiding a discussion about non-functional aspects, the question of how to design a programming interface and provide an architecture for distributed processing is related to almost all of the non-functional requirements. For instance, it must clearly deal with the aspect of product utility: while reactivity in terms of low response times is one example, throughput, scalability and reliability are other examples that need to be considered in realizing the distribution functions of the integration approach.

Adaptive Coordination The need for coordination in the context of interactive cognitive systems inevitably arises from the contradicting need to perform multiple computations carried out asynchronously in parallel (Req. 2.2) and the necessity to produce a meaningfully sequenced behavior. Additionally, it shall be possible to fuse information from different input sources that is generated at different points in time with these models. As soon as some sort of robotic actuator or other means for interaction are needed, more advanced arbitration problems arise that can only be handled by an explicit sequencing and coordination strategy taking into account the overall system context (Req. 2.3). While these functions can be encoded in the implementations of the individual components, a desirable quality from the point-of-view of software engineering is that models for coordination and control are encoded externally and controlled by the integration architecture.Considering adaptive coordination mechanisms in the integration architecture shall simplify the implementation of software components, resulting in an modular and maintainable overall architecture, thus improving testability as well as reuse of individual components.

However, in order to put coordination into effect, features must be provided that allow to dynamically change the behavior of individual components, e.g., by modifying their orchestration at runtime (Req. 2.4), for instance by activation and dynamic interconnection of components in a specific system context. Furthermore, dynamic configuration as a new requirement shall be introduced here, which can be exploited for carrying out adaptive changes, e.g., by changing the algorithmic or other parameters at runtime or prior startup without changing implementation code.

Another requirement is to allow for extension either of existing integration abstractions (Req. 4.8) or by introducing complete new patterns, which allow for behavior modification of individual components, once more ideally without changing the component's internal implementation.

While coordination mechanisms can be very domain specific, a challenge is to find models for specifying system-wide coordination behaviors in a generic way at different levels of abstraction which also incorporate environment information and system state. Furthermore, potential errors during the execution of a coordination model need to be taken into account, too. Thus, functionality for handling these anomalies must be provided by a meaningful approach. Adaptive coordination can additionally relate to declarative specification if the coordination models can be specified in a declarative syntax as it can be done, for instance, when using hierarchical finite state machines [RHS07].

Domain Support From the perspective of the VAMPIRE project, the relevant domain specific support that is primarily required considers computer vision and pattern recognition related functionality (Req. 2.5). Considering the field of experimental cognitive systems, different or even multiple domain specific functions need to be encapsulated, which may range from robotics to artificial intelligence or even other areas in order to develop and integrate a fully fledged cognitive system like an interactive robot. Support for a specific domain commonly manifests itself by providing all or a subset of the following functionalities:

- *Datatypes*: Abstract data types representing domain structures, e.g., scene objects.
- *Algorithms*: Implementations of typical algorithms like object detection.
- *Adapters*: Wrappers for directly re-usable building blocks of domain-specific functionality, for instance a software component that is wrapping a specific object learning implementation.

- *Visualization*: Visualization functions for domain-specific information. Both generic or specific for certain data structures or applications.
- *Simulation*: Support for specific hardware (e.g., Player/Stage [GVH03]) or environment (e.g., Vortex [Sim08]) simulation or combinations thereof (e.g., MSRS [Jac07]).

Obviously, all these functions are requirements an integration architecture should consider as important functionality with regard to its domain support. Unfortunately, it is well beyond the scope of the work carried out in this thesis and may not even be possible in larger projects to provide the level of domain-specific functionality needed within an integration architecture itself. Also, this functionality is often readily available by experts in the field or provided by larger organizations in feature-rich libraries like OpenCV for computer vision research. While acknowledging the importance of this aspect, in this thesis the availability of adapters for relevant software packages, cf. Chapter 3, with a particular focus on computer vision and pattern recognition toolkits is a desired characteristic of a suitable integration approach. If corresponding application adapters are not readily available, it must be easily possible to increase the level of domain support by developing additional adapters that encapsulate already existing software packages.

From a functional viewpoint, this aspect relates to almost any of the other ones, simply because developing adapters is itself an integration task. Naturally, writing adapters which are subject to the constraints defined by the VAMPIRE project's scenario and the approach developed in this thesis is after all at least related to the representation and distributed architecture aspects. Furthermore, an approach supporting a set of relevant domain functions obviously improves the efficiency of software development in a research project, e.g. through contributing to the overall agile development process, permitting reuse and incremental development.

5.1.2. Non-Functional Aspects

Practical experience from many larger research projects that were carried out in the past shows that if cognitive systems prototypes are being integrated (usually by teams of researchers from different institutes, backgrounds and countries as explained in Chapter 3), one has to consider not only domain specific requirements but always will face problems of programming in the large. Therefore, non-functional requirements have to be taken into account, too.

Figure 5.1 depicts three aspects, *Product Utility*, *Collaborative Complexity* and support for *Software Engineering* methodologies that represent non-functional requirements which need to be considered in a suitable integration approach. The individual aspects again cluster several related requirements that were identified in the previous chapters. Each maps to one general kind of NFR as defined by Summerville [Som01], which are quality attributes on the implementation level like high availability and external as well as process constraints, e.g., with regard to legal requirements or the development process. The three main non-functional aspects in this context are as follows:

Product Utility Compared to the functional aspects, non-functional aspects place constraints on the realization or feasibility of certain conceptual design decisions within the integration architecture with regard to, e.g., performance, reliability or scalability. For instance, it is important to note that within the VAMPIRE project's scenarios the degree of utility an online assistance system provides is largely dependant on its overall performance. An exemplary requirement that needs to be fulfilled with

regard to this aspect is that the round-trip cycle from image capturing, image processing and object recognition to scene augmentation must allow for an adequate alignment between the visualizations of detected scene objects and the scene the user sees in the head-up display. The latency between visualization and head movement must match the user's expectations with regard to *comfortable* interaction speed, because otherwise the utility of the product will diminish regardless of its functional attributes.

Unfortunately, it is extremely hard to define a set of comfortable parameters with regard to the overall *responsivity* of a system and the reactivity requirement (Req. 2.9) without considering a specific instance of an integrated system. In addition, it is obvious that the framework overhead shall be minimal compared to the time that is allocated to processing in the functional layer. Unfortunately, it seems impossible to compare the performance of other approaches as no standard test sets are available and the semantics of integration are often not comparable to each other. In order to get a coarse estimate for the utility of an approach, either reported performance numbers or the complexity of published systems integrated with a specific framework must be considered.

Another requirement for the utility of an integration approach is the level of scalability (Req. 4.4) for the provided integration services. *Reliability* is an additional requirement introduced here that is obviously important, even in the context of experimental cognitive systems research, cf. Chapter 4. As a non-functional aspect, it relates to almost any of the functional aspects outlined in the previous section.

In contrast to software architectures for other domains, security is not that important in the context of research systems. This is due to the fact that collaboration should not be obstructed by unnecessary security precautions. This is justifiable as the developed prototype systems, operate rather in isolation than in cooperation with possibly external hazardous services. That said, security needs at least be considered to a certain extent in order to avoid obvious abuse of, e.g., the service execution functionality of an integration framework. One way to handle this generally is to shift the responsibility for authorization to the underlying operating systems and utilize the secure protocols and its corresponding tools for externally applying security measures.

Collaborative Complexity Recalling the three types of NFRs as introduced above, this aspect represents a sort of external constraint that must be met by a suitable integration approach. This aspect is largely concerned with requirements that are rooted in the integration context in terms of research projects, heterogeneous stakeholders and user structure and the lack of standards in the cognitive systems domain as described in Chapter 3 yielding an anarchic, at best oligarchic integration situation.

In order to deal with the challenges of collaboration and to handle, for instance, geographically distributed development situations, usability and communication are primary concerns. Addressing ease-of-use (Req. 3.4), an integration approach needs to feature low entry requirements and avoid a steep learning curve. Furthermore, questions of how users can be protected from or overcome errors in using an approach as well as the convenience to work with an integration architecture are important usability factors. Usability in the context of an integration architecture may span across several dimensions including installation, software development, configuration, deployment, distribution and operation. Therefore, usability additionally calls for sophisticated support tools, e.g., for system management.

As discussed in Chapter 3, the need to embrace change (Req. 3.1) in every phase of a software development process is essential for successful integration. Frequent changes of requirements on the level

of the functional architecture are assumed to be rather natural in a dynamically evolving collaborative research project that aims at an integrated software system. Related to the call for a loosely coupled distributed processing architecture, this requirement needs to be considered by an integration architecture, e.g., by avoiding the need for re-compilation of software artifacts if interfaces are extended or when an information source in a system architecture is exchanged by a different module.

Another requirement identified upon looking at the development process in collaborative research projects is the necessity to embrace reuse (Req. 3.5) in order to benefit from already existing legacy components. This is even more important as not every novel functionality may exist at the beginning of a project although higher-level software modules may depend on it. This requirement is related to the aspect of domain support, particularly in terms of simulation functionality and existing adapters.

Last but not least, well maintained *Documentation* and technical support must be available. Introducing this requirement here comes from the observation that especially in software frameworks resulting from research projects this point is often neglected. Documentation must not be limited to an appendix in a corresponding PhD thesis. It at least necessitates an up-to-date reference manual focusing on the concepts and a complete programming-oriented documentation. For successful use of any approach this is critical. Looking at documentation and openness from a different perspective, these qualities are critical for the probability that external collaborators commit themselves to a specific platform or integration architecture. Following an open approach alleviates political reasons for not using existing software in joint projects as external partners feel not locked in to a closed platform.

Software Engineering Although software development and software integration are slightly different tasks, it is beneficial to perform both tasks according to principles adapted from software engineering. This aspect summarizes different requirements originating from this viewpoint.

In software engineering, a frequent goal is to find abstractions that provide generic solutions applicable to a class of similar problems. This is similar to what is expected from an integration approach in the context of this thesis. It shall provide a generic design space for functional architectures relevant in the domain of cognitive systems. The additional requirement which derives from that is to what extent an approach is neutral with regard to domain-specific specific architectural styles and what limits are imposed by the integration approach on the space of possible functional architectures.

Albeit not being used frequently (as shown in Chapter 3), many of the methodologies in software engineering can be useful for the development of integrated cognitive systems. In order to facilitate their use, an integration architecture needs to support these methods. One of these concepts is to support incremental development (Req. 3.3). Combined with a modular approach (Req. 2.1), integration can start early with a basic design to evolve over time.

Another way of reducing complexity in software integration is to support a (declarative) specification (Req. 3.7) of the relevant abstractions used in a system, e.g., by employing a generic modeling language like UML. At least, it must be possible (literally) to write down the application of integration concepts for a given system in order to maintain models of integration-related properties as complete as possible for all project participants. Ideally, it is an executable specification as aimed for in model-driven engineering approaches.

Support for system analysis at runtime must allow for tracing the dynamics of module interactions and global system state, e.g. to inspect the data flow between individual components, which is also important for system testing and evaluation (Req. 3.2).

5.1.3. Implementation-specific and Economic Aspects

So far, we mostly considered criteria, which are important from a functional or non-functional point of view. Some features, however, are less important from the conceptual viewpoint but need to be considered from a technological or economic perspective, which are:

Platform Support This aspect describes for which hardware and software platforms (operating system) implementations of an integration concept are available.

Language Bindings In addition to platform support, this point addresses which programming languages are supported by a specific approach. Regarding this requirement, support for C/C++ is critical as this has been the language used by most domain experts at the time of writing of this thesis.

Standard Compliance The standards compliance of an integration approach describes what standards are defined or supported by an integration approach. This aspect also relates to the openness of an approach.

Dependencies A small dependency graph regarding external libraries is desirable for reasons of software complexity and maintainability.

Licensing As within European union research projects a strategical aim is to support the open source idea, questions of licensing are naturally important.

All aspects outlined above will be used in the following to assess related work aiming at similar goals as the approach presented within the remainder of this thesis.

5.2. Software Architectures and Middleware for Cognitive Systems

Recalling the different architectural layers (*system*, *integration*, *functional*) that constitute a cognitive system instance as introduced in Chapter 1 and taking into account both the aspects identified in the previous section as well as the large number of possible application scenarios for these systems, one of the initial challenges is to get an overview of related work. Within the context of this thesis, relevant related work either stems from application-independent middleware approaches focussed on the innate problems of software integration such as object-oriented middleware or from one of the research areas that are domain specific but related to cognitive systems like pattern recognition or cognitive modelling. Figure 5.2 gives a graphical overview of the relevant areas where research is conducted related to software integration in cognitive systems. This overview and the following descriptions of each activity do not make any claims about completeness with regard to cognitive systems research in general, but nevertheless describe which areas were considered to be relevant for the task of integration in the VAMPIRE project.

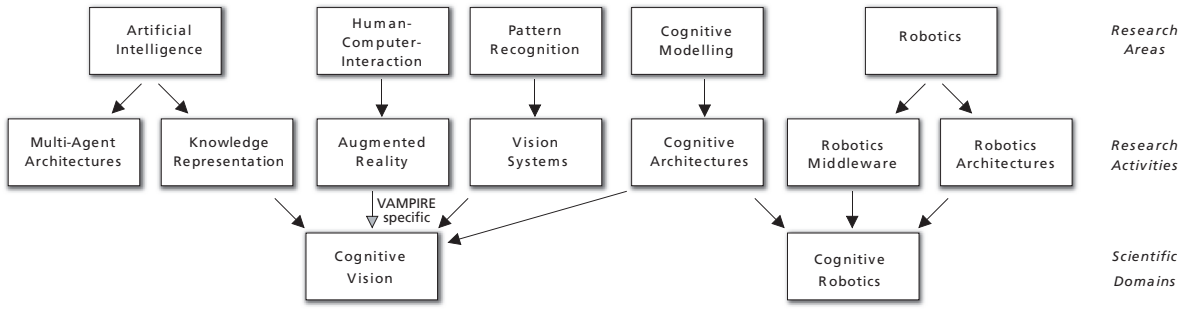


Figure 5.2.: *Overview of selected research areas in cognitive systems and corresponding research activities relevant for software integration. Additionally, their relation to cognitive vision and cognitive robotics as scientific application domains is shown.*

5.2.1. Domain-specific Architectures

Artificial intelligence (AI) is clearly related to cognitive vision research [Neu04]. Within the many subfields of AI, two activities are relevant from the point of view in this thesis. Firstly, multi-agent architectures (MAA) are the primary way of building multi-agent systems (MAS) in AI research. A MAS usually consists of interacting agents that cooperatively solve a given task. Approaches like ADE [KS06], MicroPSI [BBV06] or Jade [BCG07] are generally interesting due to the fact that they need to solve related challenges of integration, too, and are applied in similar domains. However, the properties of a single agent differ significantly from the functional model of an individual process in a visual active memory architecture in terms of autonomy, access to information and the fact that a classical MAS features no central coordination.

Secondly, knowledge representation (KR) is another long-standing area of AI research that is connected to cognitive systems research. KR is, for instance, concerned with the formal encoding of knowledge in a way that it becomes accessible to computational processes like inference engines, which in turn may generate hypotheses, validate rules, etc. Examples for techniques in this area are KL-ONE [BS85], ERNEST [NSSK90] or the web ontology language [MvH04]. Overlaps exist between KR and MAA research in the area of agent communication languages that aim at defining the interactions between software agents in a distributed system like KQML [FFMM94] or FIPA-ACL [LFP99]. While much can be learned from AI research in the area of KR and agent communication languages, many approaches suffer from their special syntactic structures for representing natural language constructs or high-level task descriptions. Other approaches like FIPA employ a specialized syntax for message encoding, which requires the presence of proprietary message parsers in every component, effectively increasing coupling between software components.

Another area that is important for the work carried out in the VAMPIRE project is human-machine-interaction research. Within this again rather broad field, augmented reality systems in particular need to be considered as they often exhibit a high degree of integration. Furthermore, AR systems are the primary application scenario within VAMPIRE. It turns out that in the context of AR research indeed specific frameworks for software integration are developed, e.g. the Distributed Wearable Augmented Reality Framework (DWARF) [MRB03] or the Studierstube Augmented Reality Framework [SFH+00]. While the latter provides limited functionality for distributed processing, the former features an event-driven integration concept allowing for network communication.

Both projects provide domain-specific AR functionality, but fall short on other required aspects, e.g., by supporting only very specific styles of interaction between the components in a system architecture or not providing any data management functionality. Therefore, we refrained from using one of these frameworks as a basis for integration in VAMPIRE.

Research on pattern recognition algorithms and computer vision systems was naturally central to the VAMPIRE project since it was defined as a cognitive vision project. Although a huge number of libraries encapsulating domain-specific functionality like OpenCV [Int08], RAVL [CVS08] or VXL [Vxl08] exist in this area and even MATLAB (with extensions like the Image Processing Toolbox [The08]) is commonly used for software development, a smaller number of approaches explicitly target the modular construction of vision systems.

Selected approaches that address this goal explicitly are:

- *HALCON*: Providing a huge number of image processing operators for many areas of computer vision and pattern recognition and featuring an integrated development environment, HALCON [ES99] is a commercial toolkit especially designated for the development of machine vision systems used in industrial environments. Figure 5.3 shows an exemplary screenshot of the included development environment.

- *IceWing*: The aim of IceWing [Lö04] is to provide a toolkit for the development and prototyping of real-time vision algorithms. It supports typical use cases in the development process of a vision system like grabbing and recording of image streams, visualization of intermediate data and dynamic parameterization. However, it featured a monolithic architecture by the time the VAMPIRE project started.

- *VisiQuest*: With Khoros [KR94] being one of its ancestors, VisiQuest [Acc08] provides a large set of libraries implementing vision algorithms, 3D vision, GUI construction and visualization. It can be utilized for image analysis and ships with its own visual programming environment for developing computer vision systems. Even so, resulting prototypes are usually restricted to a to pipe&filter style architecture.

The strength of these toolkits, which is their domain-specific vision functionality is at the same time their critical drawback, because almost all of them lack sufficient support for building larger systems in a heterogeneous environment as is the case here.

Nevertheless, the computer vision-specific functionality of these toolkits is an important feature that must in some way be available in an integration architecture for a cognitive vision system. We shall revisit this aspect later on in this thesis.

The last research area directly related to the goals of the VAMPIRE project are computational models of human-like cognition. Within this area, particularly work on cognitive architectures and corresponding toolkits like SOAR [WJ05] or ACT-R [And93] is important when looking, for instance, at arbitration and coordination in cognitive systems. These approaches provide strong support for instantiating a particular cognitive model within a specific software architecture. However, this conflicts with the general aims of the work done here. In fact, the concept is to provide an integration layer allowing for a space of possibly different functional architectures, with the primary use case to support the development of a visual active memory.

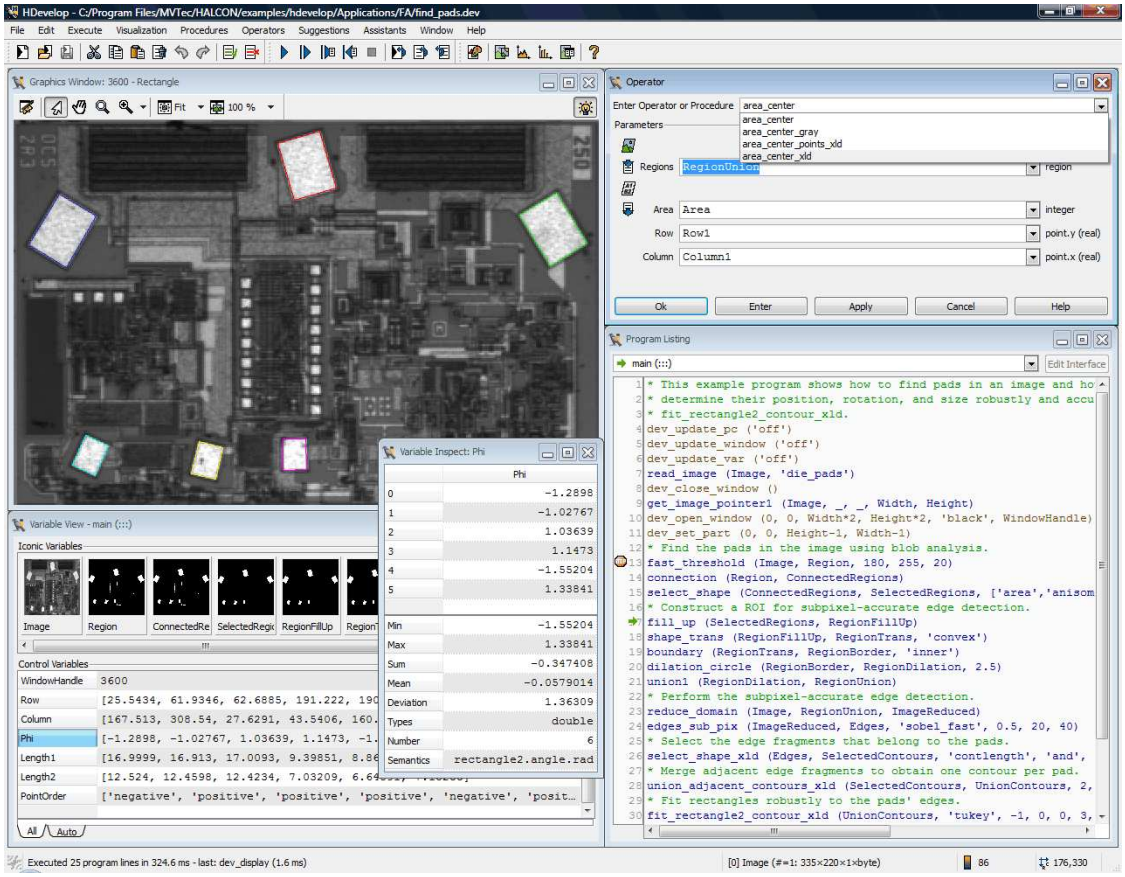


Figure 5.3.: *Exemplary screenshot of the HALCON development environment.*

In addition, it seems rather impractical to build large-scale integrated systems acting in real-time with those toolkits. This is due to limited support for the non-functional aspects defined earlier as well as the specific representations module developers must adhere to when using such an approach. However, cognitive models provide many of the functional requirements for an integration architecture for cognitive systems similar to the role the VAM architecture plays for the approach described in this thesis.

Another area that turned out to be related to cognitive systems and integration of such systems is cognitive robotics research. This is mainly due to two reasons: On the one hand, a sophisticated vision system is a critical constituent for robots resembling to a cognitive system. On the other hand, many of the requirements that were identified in the previous three chapters are important for robotics research, too. For instance, the need for concurrent and asynchronous processing as well as methods for coordination and arbitration appear as well, not to mention the fact that robotics projects are usually carried out by a larger number of people, thereby posing many of the question described in Chapter 3. Recent initiatives particularly address the problems of software integration in robotics [TS08, RoS08] and a large number of toolkits exist that support robotics software development [KS07] by encapsulation of domain specific functionality like mapping, localization or navigation. These and other functionalities like grasping objects and higher level symbolic processing are often based on cognitive vision techniques.

Therefore, the requirements for a cognitive robotics development environment need to incorporate many cognitive vision-related aspects and vice versa - with the notable exception of the low-level control aspects in robotics. Looking at the state-of-the-art in robotics integration, two overlapping research activities can be identified that need to be considered here: work on robotics middleware and work on robotic architectures.

Research on robotics middleware focuses on providing the connections between the modules in a robotic system and offers device abstractions as well as simulation environments for sensors and actuators of one or more robotics platforms. Typical examples of robotics middleware are the Player/Stage project [CMG05] or MIRO [USEK02] for mobile robotics, YARP [FMN08] for humanoid robotics or the recently introduced Microsoft Robotics Studio [Jac07] (MSRS) as a generic and easy-to-use robotics development environment. Despite their focus on rather low-level robotic functionality, robotic middleware is qualifying as related work here as it is specifically addressing communication and system integration. Unfortunately, many of these approaches lack the necessary extensibility, usability and flexibility that would be needed in order to easily apply them for the integration of a cognitive vision system. However, much of the functionality in robotic middleware is related to the concepts considered in this thesis, e.g. the component model in OROCOS or the Decentralized Software Service Protocol [NC07] (DSS) specifying the interactions of services in a distributed MSRS architecture.

Research on robotic architectures is usually either based on top of an existing robotics middleware or subsumes those aspects and adds an architectural model that specifies how sequencing and deliberative processes act in a coordinated manner towards the goals that a robotic system pursues. Examples for approaches at the borderline between robotics middleware and a robotics architecture are the Open Robot Control Software [Bru08b] (OROCOS) project or URBI [Bai05]. Both introduce features for robotic control and modeling of basic robot behaviors. While OROCOS focuses on real-time robotic control, URBI provides an event-driven scripting language that addresses the challenges of concurrency and asynchronous programming.

A prominent example of a more holistic robotics architecture that features a sound integration toolkit is CLARAty [VNE+00], which is developed by the NASA Jet Propulsion Laboratory within the Mars Technology Program and serves as technological basic for the different Mars rover prototypes. CLARAty, which is an acronym for Coupled-Layer Architecture for Robotic Autonomy excels beyond robotics middleware in that it introduces a coupled layered architecture featuring a deliberative decision layer and a functional layer that provides low-level functionality for hardware access up to higher level features for e.g. navigation realizing already mid-level autonomy capabilities.

The decision layer interacts with the functional layer and provides a framework for global reasoning taking into account system resources and mission constraints. It monitors the execution of behavior in the functional layer and can interrupted or preempt its behavior depending on mission priorities and constraints.

While the open source release of CLARAty in 2007 will almost certainly incite developments in software integration in robotics, it equally conflicts with the goal of architecture neutrality aimed at in this thesis. In contrast, it solely allows for a specific task-based functional architecture and does not explicitly address the challenges of collaboration and distributed processing. Similar to the above mentioned models of cognitive architectures, the integration architecture introduced by this thesis shall allow for the development of the higher-level functions found in functional robotics architectures like the task-based coupled-layer approach that CLARAty is pursuing.

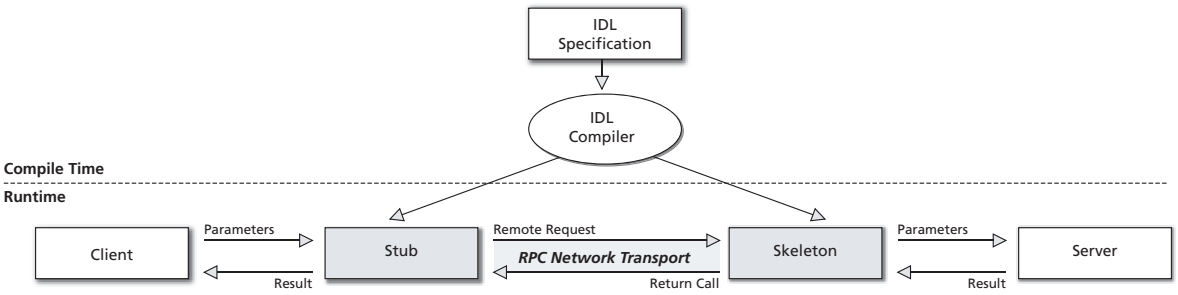


Figure 5.4.: *Use of stubs and skeletons in operation-oriented middleware communication.*

5.2.2. General Middleware Architectures

Recalling the introduction, a second domain-independent class of approaches exists that explicitly targets software integration from a purely software-engineering viewpoint, often providing the technological basis for the above mentioned domain-specific integration toolkits. As the field of middleware is extremely broad and a detailed overview of the field is beyond the scope of this work, the following description focuses on three organizational principles that allow for clustering a larger number of approaches that can be used for the integration of cognitive systems: *operation-*, *message-* and *resource*-oriented middleware.

Operation-oriented Middleware The main concept of operation-oriented middleware is to enable interprocess communication across network boundaries by providing means for calling individual functions of independent software modules. Operation-oriented middleware approaches like remote procedure call (RPC) as introduced by SUN Microsystems in the early 1980s impose a client-server style of distributed computing. A server program offers parameterized functions to its clients that can call these functions via the network with the support of an RPC library and marshalling code that is generated at compile time from an interface definition language file as shown in Figure 5.4. This functionality is encoded in so-called *stubs* and *skeletons*. Despite the transparent marshalling and unmarshalling of function parameters into a network representation, they often provide proxies [SSRB00] that are local representatives allowing transparent access to functionality in remote address spaces.

With the advent of the object-orientated programming paradigm, operation-oriented middleware extended towards a remote method invocation approach, which allows to remotely call member functions of individual objects, e.g. in Java using the Java RMI [MvNV+01] approach. While Java RMI as well as other approaches are focusing on supporting a single language, the CORBA standard [Sie00] addresses a greater audience, envisioning a language independent object-oriented middleware model. Due to the great importance of this standard and the fact that many of the aforementioned domain-specific integration architectures like MIRO or OROCOS are built on top of CORBA toolkits, we will evaluate an instance of this approach subsequently in greater detail. Recent additions that try to address different shortcomings of this class of middleware approaches are XML-RPC or the SOAP protocol.

Although operation-oriented middleware has been used to build very large-scale and mission critical systems, it is not directly applicable for integration given the required aspects introduced in Chap-

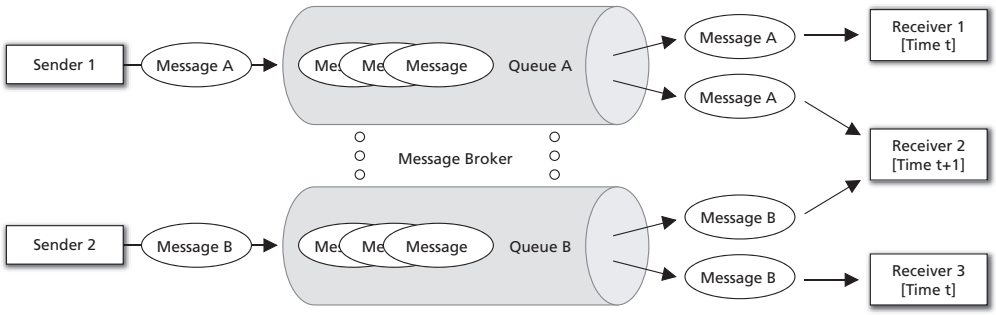


Figure 5.5.: *Message-oriented middleware architectures focus on temporal decoupling of participants utilizing indirect communication models.*

ter 5.1. First and foremost, the fine-grained per-method integration approach in a point-to-point architecture can lead to a high degree of coupling. Another aspect of coupling is that interface changes in classical RPC architectures often involves recompilation of both clients and servers. Thus, versioning and the goal to embrace change is not supported. Furthermore, advanced functionality like data management and asynchronous coordination or the general lack of scalability in operation-oriented middleware is only addressed in vendor specific ways.

Message-oriented Middleware Compared to operation-oriented middleware, message-oriented middleware (MOM) architectures directly address the challenges of distribution, taking into account a decreased level of transparency in their programming models. MOM is a concept which emerged in the mid 80s as a way of decoupling processes over existing RPC layers. This de-coupling is achieved in MOM architectures through the introduction of self-contained messages. In contrast to the automatic marshaling of RPC communication, encoding and decoding of messages in MOM architectures need to be implemented by module developers.

Using message passing instead of RPC for communication thus allows for greater flexibility as the message's content is not bound to the signature of a specific operation. On the downside, all modules connected in an architecture must share compatible message parsers. In addition to direct communication through message passing, MOM systems support indirect communication through intermediary message brokers and message queues. The resulting level of de-coupling can vary greatly and span over referential and temporal dimensions [ASTMvS02]. MOMs feature message brokers enabling deferred asynchronous communication where de-coupling in time is critical due to the fact the lifecycle of the message producing components may be independent of the consumer's lifecycle. MOM systems allow for point-to-point or broadcast communication.

Figure 5.5 shows the data flow in a typical MOM system. Messages are sent from a producer process to a message broker containing various queues, often termed *mailboxes*, which are usually identified by name. After the message is reliably received by a queue, the message can be *pushed* to or *pulled* by a receiver. Publish-/Subscribe [BMRS96] models of communication are extensions to MOM architectures where interested clients are able to subscribe to specific subjects, so-called *topics*. After subscription, clients will receive all messages that are posted to a queue and conform to a given topic specification.

Message-oriented middleware provides good support for building loosely coupled systems and often features good database integration yielding in high reliability. However, they are not directly applicable to cognitive systems integration for a number of reasons. Firstly, the focus on temporal decoupling very likely prevents their application for the integration of systems that need to act at least in soft real-time conditions. Secondly, many of the burdens of distributed programming are pushed to the module developers.

For instance, it is comparatively challenging to realize multi-threaded request-reply communication on top of a MOM architecture. Last, but not least there exists only few overarching standards in this area and many technologies are thus proprietary to specific MOM architectures like IBM's MQSeries, TIBCO Rendezvous or XMLBlaster. Nevertheless, many aspects of MOM match well to the required aspects defined earlier. The concept of message-oriented middleware evolved recently into the concept of event-based middleware, which is a central cornerstone of the approach that will be described in the remainder of this thesis. Thus, we will not discuss these concepts now but revisit them later on.

Resource-oriented Middleware Middleware architectures that focus on *resources* as central abstraction and allow the referencing of individual entities using a global identifier system shall be denoted here as resource-oriented approaches. Resources are representing specific sources of information, e.g. database items or files served by a web application. This concept is the basis of the web architecture (WWW), which is following an architectural style denoted as Representational State Transfer (REST) introduced by Roy Fielding [RTF00]. Fielding describes the externally visible behavior of a REST-based distributed system as follows:

> *"Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."* [RTF00]

The fundamental concept of a resource in this sense is that application state and functionality are divided into separate resources. That said, REST differs from the previously introduced middleware architectures in a sense that it proposes and enforces a specific architectural style for building distributed systems. However, this style turned out to be extremely scalable, which is obviously proven by the World Wide Web, and is recently considered to be a more general model for designing distributed systems. As every resource is identified by uniquely addressable through unique uniform resource identifiers (URI), it is possible in a REST-based system to navigate from one resource to another, which is what we use all day in a web browser. As Figure 5.6 depicts, another fundamental difference to, e.g., RPC concepts is that all resources share a uniform interface for the necessary state transfer, consisting of a limited set of well-defined operations. Furthermore, the message format is bound to a constrained set of content types.

Additional important properties of REST-based systems are that the communication between two parties follows a client-/server model and is usually stateless. This implies that the server does not keep track of the identity of its clients and the possibility to add caches in such an architecture to increase scalability. Despite its scalability, REST promotes loose coupling due to the document-based data exchange and the uniform interfaces concept. Due to the latter and the hyperlinking of resources with URIs, modules within a REST-based system only need a very limited view of the overall system and can traverse a link network to get access to required services. While URIs, uniform interfaces, state-

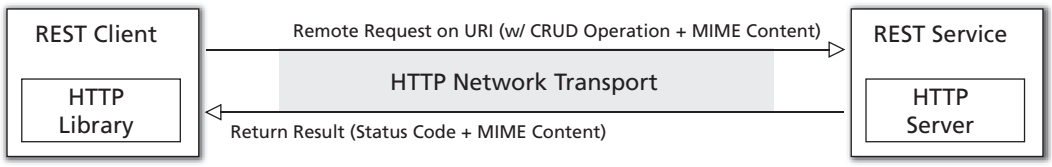


Figure 5.6.: *Overview of a client-server interaction according to representational state transfer based on the HTTP protocol and uniform resource identifiers.*

lessness and document-based data exchange support loose coupling, clients still need to understand the format of the exchanged representations. Another drawback of the statelessness of REST architectures is the fact that the communication between client and server always follows a pull puttern, not allowing for pushing notification of representation changes to subscribed clients as it is possible in MOM systems. However, REST approaches are drawing increased attention. A recent domain-specific example that follows this approach is the Decentralized Software Services Protocol [NC07] (DSS) used in the Microsoft Robotics Studio architecture. In contrast to MOM approaches, a large number of middleware tools for building generic REST-based systems following WWW standards like HTTP [FIG99] or MIME [FB96] are freely available. However, the level of integration in these tools differs from the aforementioned systems due to the fact that REST is constituted by different standards, which are supported by individual products like web servers or HTTP libraries.

Given the number of approaches to software integration in research and industry, it is impossible to review all facets that are somehow related to what is in the focus of this work. For instance, work on transactional middleware technologies, the specifications of the web services stack were intentionally omitted as they are not used much for software integration in the given domain. In contrast to these, the other approaches outlined in the previous paragraphs are more closely related to what is presented in this thesis and are directly important for the integration challenge in cognitive systems research.

5.3. Evaluation of Selected Approaches

The previous section concentrated on a presentation of the overall fields and approaches that can be considered important with regard to the identified coarse-grained functional aspects. However, neither individual approaches nor their level of fulfillment compared against the more fine-grained requirements were discussed. While this is clearly impossible for all the areas mentioned, the subsequent sections will assess three state-of-the-art instances of integration architectures in greater detail.

The assessment of the individual approaches shall be guided by the aspects and requirements identified in Section 5.1. In the following, an evaluation scheme will be applied that rates the effort needed and the difficulties to get support for a single aspect on a 5-point likert scale (--=*aggravated;* -=*difficult;* o=*neutral;* +=*supported;* ++=*strongly supported*). While most of these levels are rather self-explanatory, an aspect that is aggravated by an approach means that it is even made harder to achieve its requirements following the approach under evaluation. A caveat is that especially for the non-functional aspects the assessment is based on the subjective experiences of the author.

For these reasons, all assessments will be explained for each of the ratings due to the fact that it is very hard to measure them quantitatively. Nevertheless, the graphical visualization of the individual strength's and weaknesses compared to the defined aspects shall allow for a quick assessment of the focus of each evaluated approach.

Given the number of approaches in this area, three approaches were selected due to their *relevancy*, *recency* and *availability*. Relevancy implies that at least one instance of an integrated system must haven been built on that basis and reported on in a scientific publication, where recency calls at least one software release during the last two years at the time of writing this thesis. Last but not least, availability implies that the software license must allow for free usage for non-commercial purposes and academic research.

Applying only these criteria, still too many approaches would have to be considered. Therefore, the selection is based on the areas outlined in Figure 5.2 and the overview in the foregoing section: On the one hand, a popular representative of the object-oriented middleware paradigm will be reviewed. This is based on the frequent use of object-request brokers for integration in cognitive systems. Recent examples where object-request brokers are used have been the German service robotics initiative's project DESIRE [DES08] and as a basis for approaches like OROCOS, MIRO, SmartSoft [Sch06a] or ORCA [BKM+05]. On the other hand, two approaches from the scientific application domains of cognitive vision and cognitive robotics will be reviewed for the reasons mentioned at the beginning of this paragraph and the fact that not only the targeted application domains are similar, but also the goals these integration architectures are addressing match very well to the aspects defined earlier. The individual selection of an approach within these subfields will be argued in the beginning of each of the following three subsections.

5.3.1. Object-oriented Middleware

Object-oriented middleware approaches extend remote-procedure call systems at least with regard to three conceptual elements from the object-oriented programming model: inheritance, object references and exceptions. Operations are no longer called on processes but on individual objects. Despite proprietary approaches like Microsoft's DCOM [Mic08] or Ice from ZeroC [HS08b], the main standard in this area is the Common Object Request Broker Architecture [Sie00] (CORBA), which is a set of specifications that is being maintained by the Object Management Group (OMG). Due to the openness of the CORBA specifications, a large number of corresponding toolkits implementing them exist, including several open-source variants. The CORBA standard is completely independent from hardware or operating system environments, thus many different platforms are supported. Another claim of CORBA is interoperability in a way that a CORBA program shall be able to invoke methods on objects in any other CORBA environment.

The most important architectural concepts of an object request broker architecture usually supported by compliant CORBA middleware are shown in Figure 5.7. While certain aspects are similar to general operation-oriented middleware as described earlier, some additional elements are introduced with this approach:

- *ORB Core*: The core of the ORB handles the transparent communication between networked processes. It manages object identity through so-called Interoperable Object References (IORs) that are used for transparent access to objects regardless of their physical location in a distributed system.

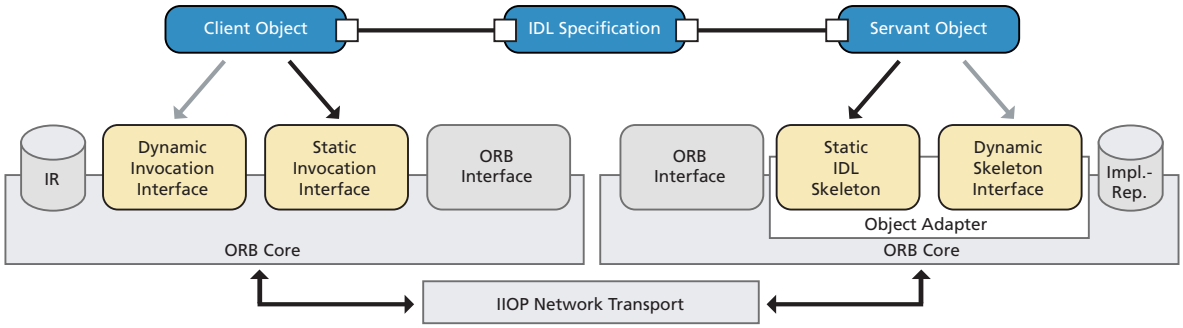


Figure 5.7.: *Overview of a standard CORBA architecture.*

- *IDL*: The CORBA interface definition language is used for describing object-oriented interfaces and data structures. Additionally, specifications for language mappings allow for the generation of standardized stubs and skeletons.

- *Servant*: The domain functionality that needs to be provided in order to serve the request on an object interfaces are usually implemented by module developers in servant classes, which provide the implementations for one or more IDL specified remote objects.

- *Object Adapter*: The (portable) object adapter is an abstraction, which maps object requests to corresponding skeletons and servant implementations.

- *IIOP*: The Internet Inter-ORB Protocol is the standard network protocol for communication between instances of object request brokers. Supporting IIOP, different ORB implementations are able to communicate with each other. Additionally, the CORBA specification allows for custom transport protocol implementations.

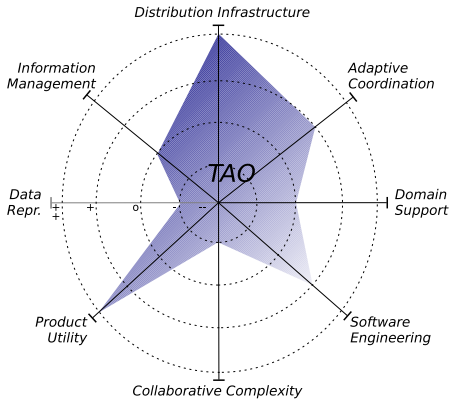
In addition to these basic features of a CORBA architecture, a large number of advanced concepts was introduced into the standard. The more important examples of these are the Dynamic Invocation Interface (DII), which allows to invoke methods of remote objects without having access to the IDL-compiled stubs for a remote object, thus removing this compile time dependency. Using DII, a client needs to explicitly specify the operation to be performed and the types of parameters that must be encoded in a request. Client objects may query the details of a remote interfaces at runtime by accessing so-called Interface Repository (IR) services.

Analogous to the DII concept, the Dynamic Skeleton Interface (DSI) allows dynamic dispatching of requests according to operation name and type parameters on the server side without using a statically compiled IDL skeleton. The DSI concept is often required in scripting languages and similar applications, which may dynamically instantiate new remote objects.

Another server side concept is the Implementation Repository (Impl.-Rep.), which provides a service that permits to dynamically instantiate required servants once requested by a remote object. Based upon this functionality, the OMG defined a larger number of common CORBA services that can be used in a system architecture if provided by a concrete implementation.

Frequently used services that provides location transparency in CORBA systems are the naming service, which allows to bind IORs to symbolic names or trading services, allowing client objects to lookup an IOR based on published properties of the offered services.

One of the approaches that defines the state-of-the-art in this field is The ACE ORB [Sch06b], a CORBA-implementation based on the Adaptive Communication Environment [SH01]. The concepts of TAO are undergoing constant development, which is lead by the Distributed Object Computing (DOC) Group of Douglas C. Schmidt. TAO is used at many universities and companies including Boeing/McDonnell Douglas, Siemens and Motorola and is available in a free *research* version that includes experimental features and as a commercially supported stable version. The target group of TAO are developers of distributed and embedded applications. As TAO is the basis of, e.g., OROCOS and additionally features real-time capabilities important in specific cognitive systems application domains (e.g., robotics) we will review this ORB as a representative for high-end CORBA approaches.



Platform Support	Unix, Windows, VXWorks, ...
Language Bindings	C++
Standards Compliance	CORBA 2.6 / 3.0, IIOP 1.2
3rd Party Dependencies	ACE
Licensing	Open Source / Commercial

Figure 5.8.: *Aspect assessment for TAO.*

Figure 5.8 shows the qualitative assessment of TAO when it is compared against the required aspects as defined in Section 5.1. Not surprisingly for a high-performance distribution middleware, the distributed processing aspects and the product utility aspect in terms of performance, latency, etc. are rated as very well supported by TAO. The reasons for these assessments and the considerations that lead to the score for the remaining aspects are as follows:

Data Representation: The CORBA IDL allows for an object-oriented data model, focusing on operation interfaces and not on data representation. The data is inherent modeled in type signatures of method parameters. A conceptual drawback of CORBA IDL is the lack of a sound concept to support extensibility and versioning of object interfaces [SV01]. Therefore and due to the natural consequence of method orientation the exchanged information is only implicitly available and understandable.

Information Management: While TAO certainly allows for the implementation of a networked information repository geared towards an active memory, it does not directly support any kind of database or data management related technology. However, the Persistent State Service (PSS), which is the successor of the persistent object service and supported by TAO may be a basis for a realization of this aspect.

Distribution Infrastructure: This aspect is fully supported by TAO. It offers functionality for all the identified requirements either through its internal architecture, its API, or external services as is the case for the question of location transparency, which is realized by TAO's naming service. Loose coupling is not directly supported by the operation-oriented and fine-granular object interface but is achievable with TAO's notification service realization.

Adaptive Coordination: Following its object-oriented programming model, anomalies are reported to communicating parties as exceptions. Concerning extensibility, TAO provides different interception points for developers to add extensions to the functionality of the ORB core. Even so, advanced features for modeling the interactions between objects are not supported.

Domain Support: As a pure middleware solution, TAO does not support domain specific features for cognitive systems research. However, a notable exception is that TAO supports an OMG CORBA specification for control and management of audio and video streams [SM99].

Software Engineering: TAO (and the underlying ACE framework) are built with many proven software engineering-related principles in mind and generally allow for a modular approach by enforcing an object-oriented programming model without further restrictions on possible architectural styles in the functional layer. Concerning system specification, standard techniques used for object-oriented modeling like UML can be applied. However, CORBA implementations such as TAO usually do not feature concepts or tools for incremental development. Furthermore, tools for runtime analysis or debugging on a system level need to be realized by system developers.

Collaborative Complexity: A drawback of CORBA-based solutions is that frequent changes are rather complicated to incorporate in an IDL-based CORBA architecture due to the reasons described above. The necessity to use an IDL precompiler adds further complexity to the development process for module developers, not mentioning the complex concepts of dynamic CORBA (e.g., DII/DSI). Despite the openness of the CORBA specifications, they are nowadays the biggest disadvantage of the CORBA concept. Since the OMG defined an overarching standard for middleware integration, the CORBA specifications embrace a large number of different requirements in integration and communication. Therefore, the amount of available functionality (and specifications) is enormous which dramatically decreases usability. Another rather problematic aspect of CORBA is that other implementations might not adhere completely to defined standards. Thus, the original idea to provide one interoperable standard for distributed systems was foiled and typical advantages of a standards-based solution are lost.

Product Utility: In contrast to the collaborative aspects, the performance and utility characteristics of TAO is superior, which is not suprising as TAO was developed and specified as high performance real-time object request broker.

5.3.2. Cognitive Vision Middleware

As introduced in Chapter 2.1, the paradigm that guided research in the VAMPIRE project was the development of cognitive vision systems. Looking at the results of the other eight collaborative projects that were funded in this research area by the European Union, it turns out that only two of these projects addressed questions of software development and integration either in scientific publications or publicly available integration architectures.

From these two, the software integration approach of the ActIPret [The05] project on interpreting and understanding activities of expert operators features a component-based software architecture termed ZWork [PVZ05], which focuses on dynamic service selection according to quality of service parameters, e.g., describing the performance of an object recognition algorithm under certain environmental constraints. However, the software has not been publicly released at the time of writing of this thesis. Similar to the goals of this approach, but with completely different concepts, a central aim of the CAVIAR [CAV07] project on image-based active recognition was to achieve adaptation in cognitive vision systems. Thus, the CAVIAR architecture [LBFT05] aims at partly autonomic coordination requiring self-describing, self-regulating and self-optimizing modules. Relying on this information, a global controller orchestrates data-flow and parameterization in a system instance according to contextual information.

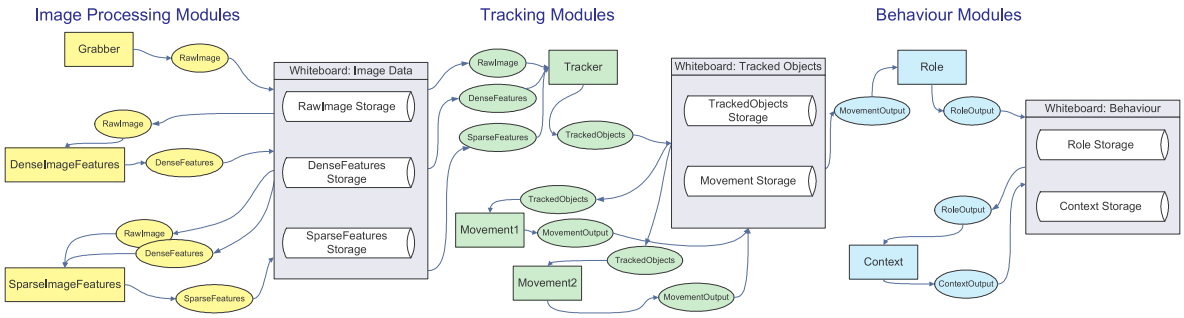


Figure 5.9.: *Software architecture of an exemplary Psyclone cognitive vision system for person tracking and scene interpretation (from [LBF+05]).*

While both approaches provide concepts for different kinds of adaptive coordination, they lack support for many of the other aspects that are important for the work described in this thesis. Interestingly, an additional integration approach was developed during the course of the CAVIAR project that already addresses many of the aforementioned aspects in a more holistic way. Due to the fact that this approach was applied in a cognitive vision scenario and that it has similar aims as what is envisioned in this thesis, the following section shall review the Psyclone architecture in greater detail.

Psyclone

Psyclone is a an integration architecture that shall facilitate the development of integrated artificial intelligence systems following the *constructionist design* methodology [LBF+05]. This software architecture has been applied for the development and integration of cognitive vision systems as well as recently to humanoid robotics [TPLD04]. The main integration abstraction provided by Psyclone is the so-called *whiteboard*, named in analogy to the concepts of a blackboard architecture, which we will discuss in the next chapter. While the authors claim to introduce this term, it can be traced back to early work in mobile robotics architectures [SST86] as reported in a reference book of software engineering [SG96]. Figure 5.9 depicts a cognitive vision system instance as developed in the CAVIAR project performing a scene interpretation task that is integrated according to the Psyclone concepts. The basic idea of Psyclone is to mediate all data flow through central server instances, i.e. scheduling blackboards [TLPD05] that feature a *generic data format*, a *type ontology* for messages and data streams as well as *routing specifications* allowing for a declarative setup of module interconnections. Whiteboards act as publish-/subscribe services for registered modules. The system architecture shown in Figure 5.9 consists of three whiteboard instances that dispatch data of different system layers, namely images, tracked objects and high-level role and context information to modules in higher layers. Thus, Psyclone systems are conceptually similar to classical bottom-up *pipe-and-filter* [SG96] software architectures as no top-down links between the different layers are established.

Whiteboards allow to store the exchanged information for a reasonable amount of time (the authors do not provide any further information about the capabilities of their approach in that respect). While the stored information is made globally available via a query interface, it is not possible to update this information. However, whiteboards support *push* (publish) and *pull* (query) communication styles, which will be discussed in more detail in subsequent chapters. Psyclone uses a protocol termed OpenAIR [Min07], whose specifications are freely available from the author's web page.

```
<sequence name="Fight_OneManDown">
  <frame number="192">
    <entitylist>
      <entity id="1">
        <orientation>151</orientation>
        <box x="81" y="101" w="31" h="21" />
        <appearance>visible</appearance>
        <movement>walking</movement>
        <role evaluation="1.0">walker</role>
        <event evaluation="1.0"></event>
        <scenario evaluation="1.0">immobile</scenario>
        <situation evaluation="1.0">moving</situation>
      </entity>
    </entitylist>
    <grouplist>
      <group id="0">
        <orientation>103</orientation>
        <box x="228" y="110" w="55" h="126" />
        <entities>4,5</entities>
        <appearance>appear</appearance>
        <movement>active</movement>
        <role evaluation="1.0">fighter</role>
        <event evaluation="1.0"></event>
        <scenario evaluation="1.0">fighting</scenario>
        <situation evaluation="1.0">merge</situation>
      </group>
    </grouplist>
  </frame>
</sequence>

```

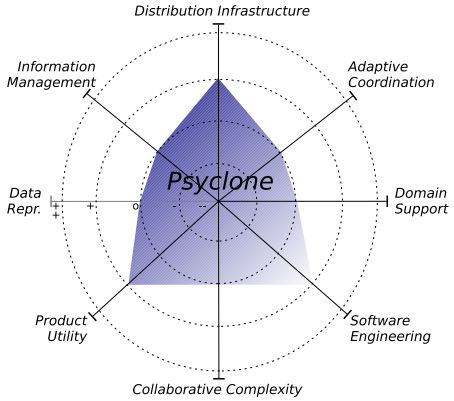
Listing 5.1: *Exemplary instance of a Computer Vision Markup Language XML document.*

Psyclone employs the *Computer Vision Markup Language* [LF04] (CVML) as a general purpose message format encoding the data exchange between components in a vision system. Figure 5.1 shows an excerpt of a CVML document, encoded high-level information about a scene that is observed by a system as shown in Figure 5.9. In contrast to special data areas within the whiteboards, the exchanged messages are additionally tagged with information about their unique identity, individual priority and framework-generated timestamps.

In order to subscribe to different types of data, interested modules connect to a whiteboard and register their interest in a specific message type. These types are encoded in a dot-delimited list yielding a straightforward type hierarchy. For instance, the expression "*Input.Perc.UniM.Hear.Voice.Speak*" describes a message type that was generated by speech recognition module. In Psyclone, this type ontology serves as the basis for routing of new data that is published on a whiteboard.

While Psyclone supports the concept of continuous data streams, this type of exchanged information is not mediated via whiteboards. Stream-based connections and their datatypes must be defined at compile-time due to certain technical restrictions [TLPD05]. As the authors do not elaborate further on this issue, it appears that the sole purpose of the whiteboards for stream communication is to act as a nameservice for the clients of a streaming service.

In contrast to other approaches, Psyclone offers both a library for the integration of external processes and a local runtime environment for modules, which allows for a more efficient coupling between a set of modules that feature more sophisticated timing constraints. However, integrating internal modules via so-called *cranks* is done by calling a C-function with a specific signature that must be exported within dynamically loaded libraries, acting as a hook for these modules to get called by the Psyclone runtime. Through this mechanism, the Psyclone runtime provides a proxy to the actual whiteboard object, and, in turn, the individual module may access any of the whiteboard's functions. Last but not least, a *context* mechanism is supported that allows to switch module configurations dependant on this single system-level string-value parameter. The context can be set by any module that is part of a Psyclone system.



Platform Support	Unix, Windows, MacOS
Language Bindings	C++, Java, Lisp
Standards Compliance	XML, *CVML, OpenAIR*
3rd Party Dependencies	Core Library
Licensing	Commercial

Figure 5.10.: *Psyclone assessment.*

The psyclone runtime environment is realized in C++ and runs on Unix, Windows and Macintosh operating systems. Language bindings for remote integration of process are available for C++, Java and Lisp. Psyclone is freely available for non-commercial use in a closed-source version with certain restrictions. Additionally, the version that, at the time of writing this is available for download appears to be a limited evaluation version. Figure 5.10 shows a qualitative assessment of Psyclone compared against the previously identified aspects. The reasons for this subjective evaluation are as follows:

Data Representation: Psyclone itself does not make any assumption about the user's data model and does not prescribe any, except that all information must be converted to character data (or well-formed XML elements) due to the fact that it is embedded within OpenAIR messages, which ultimately are XML [BPSM+04] documents. Consequently, Psyclone suggests to utilize CVML for data exchange between the modules within a vision system's architecture, thus providing at least some guidance for module developers. However, no advanced features of XML or CVML are exploited or specifically supported within Psyclone and the data model for stream-based communication as a binary protocol is independent from the textual format used for whiteboard communication.

Information Management: In contrast to blackboard systems that resemble database management systems, Psyclone lacks most of this functionality and is closer to publish-/subscribe [BMRS96] architectures for distributed systems. Therefore, no global state can be updated through the whiteboards, except by distributing state information across all connected modules. The question remains how this is handled across different whiteboards. However, the ability to recall information within whiteboards and a so-called *catalog* function that serves as a persistent data storage for information to be reloaded after a system shutdown outside the whiteboards provides at least a limited level of information management support.

Distribution Infrastructure: Psyclone offers networking functionality following a publish-/subscribe pattern utilizing an XML-based protocol that shall yield loose coupling between system modules. Additionally, media streams are provided allowing for a more efficient way of dealing with audio and video data although no detailed information about their realization is available and the benefits of a document-oriented approach are decreased due to the binary protocol used here. Equally opposed to the idea of loose coupling is the fact that whiteboards and media streams must be defined prior to system startup and location independence is only provided for whiteboard-based interactions. While asynchronous integration is naturally supported by this model, Psyclone offers no asynchronous processing API for the attached whiteboard clients and no further interaction semantics than messaging via publish-/subscribe.

Adaptive Coordination: The Psyclone architecture supports a basic level of coordination in a sense introduced earlier due to the ability to fully configure the system architecture and its dataflow in a central point of configuration. However, no dynamic reconfiguration is supported by way of this mechanism. Nevertheless, modules can be dynamically instantiated and connected to whiteboard instances at runtime. Coordination in Psyclone systems is based on a simple type ontology for the module's message subscriptions, no further functions for filtering, matching or the based on module identity are available for subscription specification. The fairly simple context mechanism provides a very limited system coordination facility compared to other approaches developed in the CAVIAR project [CAV07]. It is neither possible to model the coordination dynamics in a meaningful way nor to change this at runtime. Furthermore, there exists no possibility for framework extension or features for intercepting the data flow between modules and the whiteboard for system integrator's or module developers.

Domain Support: Psyclone offers no special domain-specific extensions or adapters, except that the author's propose to encode vision system related information in CVML. However, on the one hand it is questionable whether this proposal fits to a larger class of vision systems and on the other hand, no further support seems to be publicly available for the datatypes defined in this domain-specific XML dialect.

Software Engineering: Despite its similarity to a blackboard architecture, Psyclone allows for different functional architectures. It offers a limited level of modularity as it permits to decompose a problem into several different whiteboard clients or whiteboard servers to the extent this is possible within a blackboard architecture. However, advanced features for a modular development like hierarchies or subsystems are not available. As far as this could be considered by reading publications and looking at the documentation, the main feature of Psyclone that matches this aspect is a tool called PsyProbe. It provides a web-based mechanism for analyzing whiteboards at runtime and allows for manual modification of system behavior.

Collaborative Complexity: Psyclone scores with regard to collaborative complexity first and foremost with regard to its good documentation and the openness of the used networking protocol and of the CVML specification. Unfortunately, the whiteboard implementation itself is closed source and therefore contradicts the idea of providing an open integration architecture. Furthermore, it remains unclear how change is handled in the Psyclone messaging architecture and whether OpenAIR or CVML actually are endorsed by a standards body.

Product Utility: As all communication is usually mediated via whiteboard servers based on a textual protocol over a standard network, the overall performance may decrease. In order to overcome this, Psyclone offers the ability to integrate modules within an instance of a whiteboard, thereby avoiding the overhead of network communication. While this is in general a useful feature for a close coupling of modules, the Psyclone approach leaves many questions unanswered, e.g. about the life-cycle of cranks, timing issues if a module blocks within a hook and in general the error handling procedures Psyclone implements here.

5.3.3. Cognitive Robotics Middleware

Research conducted in the domain of cognitive robotics is in many ways related to what is done in the area of cognitive vision systems. First and foremost, the goal to enhance a robotic system by cognitive abilities implies a strong need for building an integrated system. Secondly, a cognitive robot usually consists of several interacting components, which again are developed in an inherently interdisciplinary approach. Last but not least, cognitive vision functions can be considered as important subsystems of cognitive robots.

Although the distinction between robotics middleware and robotics architectures on the one hand and cognitive robotics approaches on the other hand as depicted in Figure 5.2 is sometimes arbitrary, a number of approaches define this intersection due to their specific support for the needs of software development in cognitive robotics. Examples of these approaches are the component-oriented approach that is used for the development of cognitive abilities at the Honda-Research-Institute Europe [CJD+06], the CAST/BALT integration architecture [HZW07] that emerged in the context of the Cosy [COS04] European Union collaborative research project or MARIE, which is an integration approach that explicitly targets the re-use of larger building blocks of domain-specific software needed for the development of cognitive robots.

In contrast to the aforementioned approaches that provide rich integration functionality based on a fine-grained component-model, requiring researchers to adapt their software modules to specific constraints imposed by the integration architecture, MARIE aims at integrating more coarse-grained software modules. Similar to what is in the focus of the approach described in this thesis, MARIE aims at allowing for a *minimally invasive* integration strategy, connecting software modules without large modifications to their internal structure to a system architecture. Therefore, MARIE's concepts shall subsequently be compared against the aspects, which were identified as important for integration in the context of the VAMPIRE project.

MARIE

MARIE [CBL+06] (Mobile and Autonomous Robotics Integration Environment) is a middleware-oriented approach that is geared towards application integration in robotics. The specific motivation of the developer's of MARIE has been to overcome the lack of standards in the robotics domain [Ore99] by a unifying abstraction and integration layer. The MARIE approach has for instance been used to develop Spartacus, which is a socially interactive mobile robot [MBCC+05] that features localization and mapping, a certain level of visual processing and dialogue interaction integrated on the functional level according to a behavior based approach called motivated behavioral architecture.

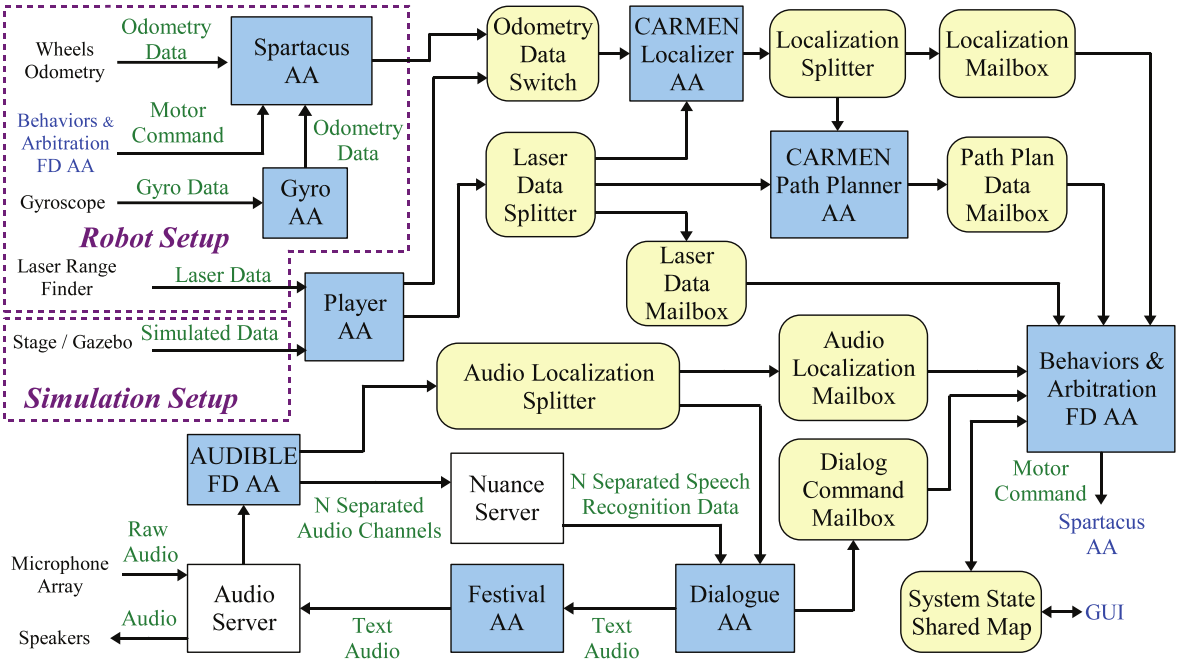


Figure 5.11.: *Software architecture of Spartacus [CBL+06], integrated utilizing MARIE.*

The fundamental ideas behind the concept of MARIE are to facilitate reuse in the domain of robotics research to foster the development in the field. Therefore, this approach aims at supporting multiple concepts as well as abstractions for integration as different experts expect a different set of specific functionality on varying levels of abstraction.

Another vision of MARIE is to support a wide range of different communication and integration platforms in robotics. The architectural concept is based on the idea of a so called *component mediation layer*, which is a network- and system-wide implementation of the mediator pattern as known from object-oriented design [GHJV95] on the level of a distributed software architecture. The design rationale for the chosen approach are to support loose coupling and to hide the internal implementation of each individual component.

In order to support modules developers and system integrators, MARIE provides a component framework that shall allow for the management of integrated applications with regard to component lifecycle (e.g. initialization, (re-)starting, ...) as well as dynamic configuration. Apart from that, MARIE itself does not provide a component execution container or a runtime environment. The concept of MARIE introduces four fundamental functions that are responsible for the desired abstraction from individual applications and their specific communication protocols as well as their integration with the mediation layer, which in the end carries out the necessary interaction.

The notion of an *application adapter* is used in similar ways as the term adapter is defined in application integration or software engineering. It adapts and connects the local interface of a specific application to the mediation layer, thereby integrating this application into a larger system. In contrast to the concept of an application adapter, the idea of a *communication adapter* is to apply protocol transformations necessary for interconnection of otherwise incompatible components.

While adapters can become specific for each software application, the following general types of communication adapters were used in the Spartacus integration project as shown in Figure 5.11:

- *Mailbox*: a data buffer for interactions between asynchronous components.
- *Splitter*: forwards incoming data to multiple outputs.
- *Switch*: sends only one of its inputs to an output port.
- *SharedMap*: a push-in/pull-out key-value data structure accessible by all components.

Both types of adapters are used by *communication manager* processes, which are responsible for the orchestration and management of the communication links between components. One instance of a communication manager needs to be run on any single node that is part of a MARIE system. However, at the time of this writing, no implementation of the communication manager concept was published that demonstrates this feature. Last but not least, *application manager* instances control the execution of the different components that run in the overall system. Once more, an instance of this type of manager must be running on each node participating in a distributed system setup. Although in a prototypical state, implementations of this function are available.

An implementation of MARIE's concepts is available under the GPL and LGPL open-source software licenses. It is based on ACE [SH01] that provides portable abstractions from low-level operating system functionality like threads, memory access or sockets as well as many more. Therefore, the framework is usable on recent Microsoft and Unix operating systems. On the downside, no other language bindings than those for C++ are available. Although MARIE implements a Port concept, which shall in principle allow to change the underlying communication strategy, it turned out that only ACE sockets are used for network data exchange in the mediation layer [CBL+06]. Furthermore, all data exchange in this layer is solely based on text-only XML documents, binary data is not supported in these mediated interactions.

One of MARIE's strengths, which is the available support for the robotics domain, leads to the qualitative suitability rating of this approach shown in Figure 5.12 and its available implementation compared to the aspects summarized in Section 5.1. The considerations that lead to this assessment are as follows:

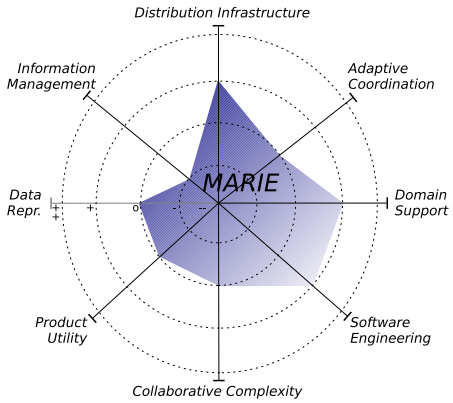
Data Representation: The MARIE concept does not prescribe the use of a specific data model for software integration. In fact, it suggests to use XML for data serialization in network communication without making further use of XML concepts, thus not adding value with regard to software integration except a possibly improved understandability on the protocol level. While the data types that can be used with MARIE are extensible, corresponding factories and parsers must be provided by the users of the framework. In general, questions of data representation have little impact on the integration concepts of this approach.

Information Management: Except for the fairly simple SharedMap communication adapter that allows to store text data in a map-like structure, no advanced functionalities for this aspect are available.

Distribution Infrastructure: With regard to distributed systems functionality, MARIE's strength is its support for the integration of heterogeneous infrastructures via the Port concept. On the downside, this capability comes at the prize of added complexity, e.g., with regard to the necessary configuration. While asynchronous processing is supported and parallelism as well as concurrency are addressed by the underlying ACE framework, location independence is not considered. For example, concepts for hiding the physical location of a component are not available. Despite that, MARIE supports loose

coupling to a certain extent, mainly by decoupling component interaction through the mediation layer. Nevertheless, some questions remain, e.g., with regard to the fact that for fast exchange of multi-modal data the mediation layer is circumvented and direct network communication is applied [CBL+06].

Adaptive Coordination: Concerning features for adaptive coordination, only dynamic configuration of components is supported, which fully depends on the implementation of the application adapter with regard to relevant adaptation. While the communication manager may very well be a concept for flexible orchestration, no working examples of this idea are available. Application managers are supposed to provide a limited support for fail-stop errors but descriptions of how these are handled are not available. Neither other types of coordination functions, nor extension points on the framework level are considered here. However, at least the additional RobotFlow software toolkit interfaces with MARIE and thus allows for graphical modelling of the data flow in a robot architecture and the specification of coordinated robot behaviors.



Platform Support	Microsoft Windows, Linux
Language Bindings	C, C++
Standards Compliance	XML
3rd Party Dependencies	ACE, adapter dependencies
Licensing	GPL, LGPL

Figure 5.12.: *MARIE assessment.*

Domain Support: The aspect of domain support is a strength of MARIE, but limited to robotics functionality. Besides a number of robotics related datatypes, adapters for a number of relevant robotics software packages such as Player/Stage [GVH03] are available.

Software Engineering: MARIE's mediation layer and integration concepts do not require a specific functional architecture. Modularity is supported by the component concept although no further guidelines for conducting software development in a modular manner or how to achieve incremental development are provided. MARIE provides a straightforward set of abstractions describing component interactions, but does not suggest any way of modeling instances of integrated systems. It features a tool for inspecting system interactions, thereby allowing runtime analysis. Beyond advertising unit testing and available benchmarks there is no additional support for system debugging or testing.

Collaborative Complexity: While it remains unclear how change is addressed in this integration architecture the already available application adapters underline MARIE's aim to provide a framework that allows re-use of existing software packages. Concerning usability, an observation is that integrating additional components needs the module developer to deal with a number of abstract classes that must be used to build a novel application adapter, which can become cumbersome.

Moreover, the invocation of target methods needs to be handled by the framework user and that component configuration is everything but a trivial task. The interchangeable protocol implementation on the one hand, but the incomplete (design) documentation on the other hand, a limited level of usability and openness can be attributed.

Product Utility: Unfortunately, this aspect cannot be assessed given the available information. None of the desired features is evaluated by the author's of the framework. In particular, nothing is reported on the scalability of the mediation layer. The only available fact is that MARIE has been applied for the integration of several robotic systems, which are able to act in the real world.

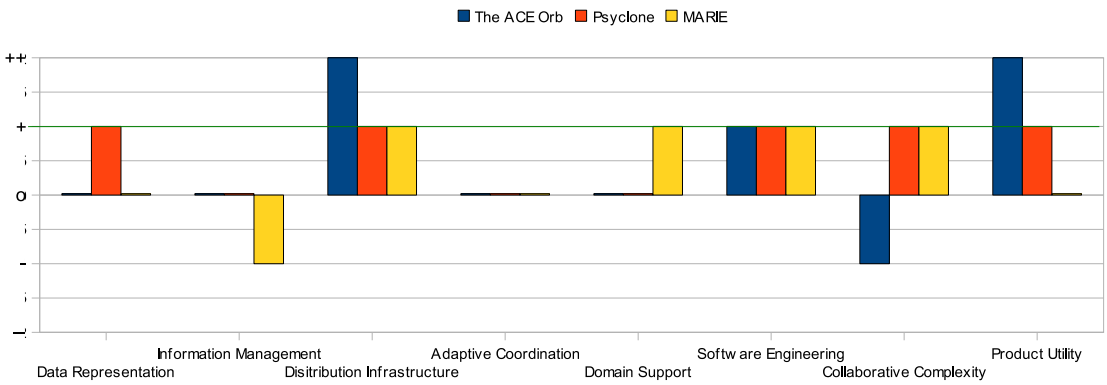


Figure 5.13.: *Level of support for the defined functional and non-functional aspects. The green line represents the desired minimum level of support for all of these aspects.*

5.4. Conclusion

The domain-specific integration architectures reviewed in the previous two sections represent related research that emerged in parallel to the approach presented in this thesis. However, it turned out that no single approach sufficiently considers all of the requirements as depicted in Figure 5.13.

TAO represents *the* state of the art in real-time CORBA and scores in performance-related aspects as well as all other areas of distributed processing. However, while TAO may be a basis for an integration architecture in cognitive systems, its direct application in an interdisciplinary integration context as is the case for cognitive systems projects is prohibited mainly due to its complexity and missing support for an iterative development process with constant change being a central reality.

Looking at Psyclone and MARIE as integration approaches developed in related scientific projects and with similar goals in mind, it is natural that they already consider many of the aspects defined as requirements. Psyclone offers an approach that shares at its core many similarities with publish-/subscribe systems and less similarities with classical blackboards. On a functional level it addresses most of the aspects that could be useful for the integration scenario within VAMPIRE, but in a way that many functional requirements are only briefly addressed. MARIE has good support for the domain of socially interactive robotics and provides at least on the conceptual level a sophisticated architecture for integrating coarse grained software components. On the other hand, it introduces overly complex abstractions for functions that are even unused up to now (communication manager) or in its early stages (application manager). Finally, it remains unclear how component interfaces can be specified and how the considerable additional effort of maintaining a number of additional infrastructures can be managed with MARIE's concepts.

However, particularly the aspect of information management, which is central for the realization of a visual active memory is not directly support by Psyclone and by MARIE. To commence this chapter, I would assess that Psyclone represents the approach that comes closest to the architecture described in this thesis. Even so, despite the fact that there are certain conceptual similarities, the two approaches feature important differences. As none of these architectures completely match the requirements identified for the VAMPIRE project, the remainder of this thesis will bear this challenge and explain how these aspects are supported by the architectural approach to be introduced subsequently.

Part II.

The Information-Driven Integration Approach

Information-Driven integration describes an event-based approach for the collaborative development of complex software architectures as needed to realize experimental cognitive systems. The following part consists of two concerted chapters that in conjunction describe the models of information-driven integration.

Chapter 6 describes the models that consitute the core architecture, which is primarily adopting concepts of event-based systems yielding a versatile communication environment for software components that operate in a distributed system architecture. The abstractions presented in this chapter provide the conceptual and technological basis for advanced integration functions.

Chapter 7 explains extensions built on these basic models that realize the additional requirements prevalent for software integration in experimental cognitive systems. Exemplary models that particularly support the software development in this domain are the memory model, which provides a foundation for a visual active memory as envisioned in the VAMPIRE project or a coordination model for flexible perception-action linking in an asynchronous architecture.

6. Adopting Event-Based System Models

Developing an architecture for software integration that is used right from the beginning of a new project is a challenging endeavor for the component and the framework developers. This is particularly true in the given integration context because changing requirements not only yield challenges for the individual functional services within a complex system but also progress of the attributes and structures of the overall system architecture towards the realization of a project's scenario. Therefore, it is a necessity that the integration architecture meets these changing requirements. Consequently, the approach that will be introduced in this chapter is the streamlined result of an evolutionairy development process, particularly conceived during its application in the VAMPIRE and COGNIRON EU projects. However, the evolution of the subsequently presented approach once more confirms Lehman's first law [ER03], which states that a system that is used *will* be changed. And since the fact that it actually *is* used is good, change shall be appreciated here.

Setting out from a first implementation of a visual active memory, which was based on a rather fixed client-server architecture, the whole approach developed into a generic and extensible integration architecture comprised by a set of stable methods representing the core of information-driven integration. This chapter presents the first part of these concepts primarily considering event-based communication. They represent the foundation all other models that will be described in the next chapter are built upon. In terms of the functional aspects introduced in the previous chapter, the following sections mainly address *data representation*, *distribution* and features for *adaptive coordination*.

This chapter starts with a manifesto of information-driven integration. It highlights a number of strategic aims that are considered important for the given task and help to weight conflicting requirements. It also discusses well-known architectural styles like service-oriented architecture, which provide valuable insights adopted for the concepts of information-driven integration. The assumption in this work is that in particular elements from event-driven architectures match best with the stated requirements. Thus, the remaining sections in this chapter shall explain the core features of information-driven integration along the lines of event-based systems models. The different models are described from an architectural viewpoint focusing on their concepts. Specific implementation aspects are explained in excerpts that describe technology-agnostic features in some level of detail.

6.1. The Manifesto of Information-Driven Integration

The transition from the vision of a visual active memory architecture to an actual implementation of a system-level software architecture that facilitates cognitive systems research needs careful consideration. The first step in doing so has been requirements identification and analysis. However, as indicated in Section 5.1 some of the stated requirements are even contradicting. Thus, the first step to move from analysis to design is to highlight some aspects that shall be particularly accentuated by the architecture, which helps in selecting and developing suitable software engineering concepts.

6.1.1. Strategic Aims

While the overaching aim of this dissertation is to support collaborative research projects on experimental cognitive systems by providing a software architecture that is broadly applicable across different application scenarios, the following list presents specific strategic aims resulting from this integration context, which represent a kind of a *manifesto* for the core architecture to be developed:

1. *Loose Coupling*: Reducing the dependencies between functional modules on the system level in terms of temporal, spatial and referential coupling is a primary concern of the information-driven integration architecture, see Chapter 4 for a detailed discussion. While coupling is necessary for meaningful systems, the decisions *where*, *what* and *how* to couple software artifacts must be made explicit and be supported by the architecture. In general, loose coupling is paramount for handling oligarchical or anarchical integration contexts, cf. Chapter 3 in business enterprises or in collaborative cognitive systems research.

2. *Explicit Boundaries*: In contrast to operation-oriented middleware that aims at maximizing distribution transparency, cf. Chapter 4 this approach takes on the position of Waldo et al. [WWWK97] that it is undesirable to hide too much of the fact that an interaction with a software artifact executed in a different execution context takes place. The message sent to or received from a module, its contract, and a representation of the remote communication partner itself should all be first-class constructs within the integration architecture. Hence, the resulting programming models shall provide an API that exposes these concepts to the module developer.

3. *Increase Autonomy*: Interacting modules such as those integrated in the VAMPIRE systems, cf. Chapter 2, should in general not rely on a specific execution context, due to the fact that the loose coupling principle mandates that dependence on implicit assumptions should be as small as possible. In contrast to operation-oriented infrastructures that require synchronized evolution of client and server program code and interface descriptions, the interaction between modules in this architecture shall not be based on class types but instead on sharing of interaction contracts, which shall facilitate an agile, independent development process as envisioned in Chapter 3.

4. *Focus on Usability*: Despite the stated aim to make application boundaries explicit, the programming models that allow developers to make use of functionality offered by other modules should naturally be as easy as possible. However, this is even more important in the given interdisciplinary context, cf. Chapter 3, due to the fact that usually only few middleware experts will be among the users of this architecture. The goal is to pick people up whatever they already know about distributed systems technology, making simple things easy while still allowing experienced users to develop complex functionality. Thus, the viewpoint taken in this thesis is that the architecture shall focus on a minimal core of important functions needed for the design and integration of distributed cognitive systems, exposing only a small set of recurring patterns in its external API for use by regular module developers.

In addition to these non-functional aims which influence the conceptual architecture, a secondary but nevertheless important aim that constraints the actual software development is to endorse the ideas of *free and open source software* as this is often a requirement for collaboration between academic institutions. By endorsement, the reuse of existing packages and the free provisioning of the resulting software toolkit according to an open-source license is meant.

This work's ambition is to become actually used in large-scale research projects, thus the resulting software architecture needs to be developed from a holistic viewpoint, aiming at supporting most stages in an iterative development cycle of cognitive systems: design, construction, integration, test and operation. In order to achieve this aim to a certain extent, pragmatism is preferred over evangelism, rendering useful concepts from otherwise orthogonal paradigms such as service-oriented and event-driven architectures or tuplespaces into a coherent novel architectural concept.

Apparently, the reader could ask why the domain specific functional support for a cognitive vision project like VAMPIRE is not prominently considered in the above list. The answer is two-fold: on the one hand, many of the above mentioned aims extend to the development of individual functional processing modules and from a functional viewpoint already many well designed libraries for domain specific tasks exist. On the other hand, the proposed architecture provides domain support by easily integrating itself into existing domain specific toolkits as explained in Section 7.5. The challenge as it is understood in this thesis is to support the development of experimental systems for cognitive interaction which exhibit *composite behavior* based on the sum of all individual functional modules in a software architecture.

Given the aforementioned aims and the requirements from Chapter 5, the question arises what concepts eventually can be adopted to find a solution for the imposed challenges. From a high-level perspective, a first answer is to draw inspiration from service-oriented architectures.

6.1.2. The Service-Oriented Viewpoint

Nowadays, *Service-Oriented Architecture* [OAS06] (SOA) represents a popular approach that composes systems of autonomous services. The vision is to promote integration to become a forethought rather than an afterthought, which is not only important for enterprise application integration but also for software architectures of experimental research systems. Naturally, not all of the SOA concepts are new but rather evolved out of the experiences associated with designing and developing distributed systems based on technologies explained in Chapter 5. Similarly, many service principles have their roots in earlier techniques from object-oriented analysis and design such as encapsulation, abstraction and clearly defined interfaces.

A service can be defined as a high-level application function that can be interacted with via well-defined message exchanges. Services shall be designed for both availability and stability. The basic interaction model of web services is request-reply, which is similar to what standard operational middleware provides with remote procedure calls. However, SOA raises the level of abstraction in these interactions and focuses on the semantic functions of a module. While the granularity of request-reply in SOAs is usually more coarse-grained than in classic operational middleware, the interaction is still identity-based, leading to a stateful coupling between caller and callee [MFP06]. While usually applied in business contexts, the assumption of this thesis is that some of the general SOA concepts implemented on a loosely-coupled infrastructure can as well lead to much more open and changeable software architectures in experimental cognitive system's particularly compared to the development of systems based on operational middleware.

So, the question arises what actually differentiates a service-oriented architecture from a distributed system that is integrated utilizing well known techniques.

The characteristics of a SOA primarily are the same as the five aspects of modularity that were introduced by Bertrand Meyer (*decomposability, composability, understandability, continuity, protection*, cf. Chapter 4), but with distinguishing extensions [Per08]. The following list interprets these from the perspective of service development as these properties are not only dependant on the capabilities of the integration architecture but also of the design of the individual modules:

- *Introspection*: Services must be able to query the structure of modules and their communication at runtime.
- *Remotability*: Services in an architecture should be designed and planned for existence in a distributed and heterogeneous computing environment.
- *Asynchronicity*: Services shall not assume an immediate response from an interaction and take into account latency either in the transport mechanism or the callee.
- *Document Orientation*: Services must not implicitly share state across single interactions and shall explicitly communicate via well defined messages.
- *Standardized Protocol Envelope*: Service share a common envelope message format for module communication.
- *Decentralized Administration*: Services should be designed and planned for decentralized administration, which allows their reuse in different organizational contexts, e.g., projects.

All these aspects contribute to the central goal of loose coupling on the level of an integration architecture. This list partially overlaps with the functional requirements that deal with distribution and some of the desired non-functional requirements. However, the SOA paradigms are not necessarily bound to any specific implementing software technology, even if SOA is frequently reported to be linked to web service standards [NL04]. Fortunately, gaining insight from the principles of SOA is not coupled to the use of these overly complex stack of standardized specifications. Nevertheless, the abstract characteristics of a service and its underlying design considerations can provide an avenue for the collaborative integration of experimental cognitive systems, which will be detailed in the remainder of this thesis.

Acknowledging these considerations, the following section and the models of the IDI approach explain the realization of a software integration architecture that take into account the SOA paradigms and realizes these as well as further functions suitable for cognitive systems engineering by adopting concepts from event-driven architectures, which shall be introduced next.

6.1.3. The Event-Driven Perspective

While the focus of SOA is on decomposing system functions in a command-and-control style, the general ambition of *Event-Driven Architecture* (EDA) as interpreted here is to support the exchange of events that contain information about semantically important observations. This notion of an event matches well with the natural characteristics of the application domain. The real world provides many good examples of occurrences that can easily be described with events like a person entering a scene, a robot that hits a wall with its bumpers or even better, the obstacle avoidance that detected this barrier already some seconds in preface.

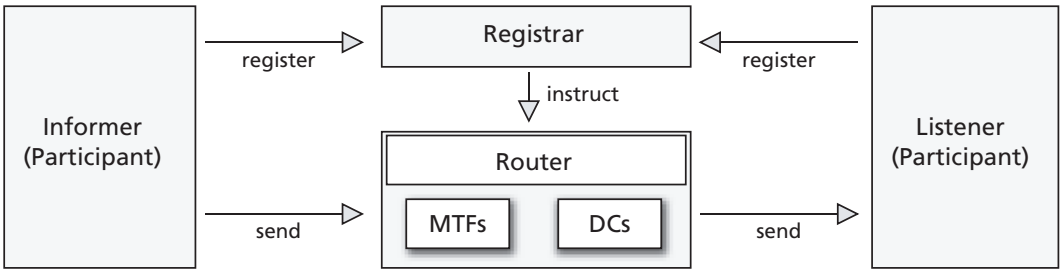


Figure 6.1.: *Functional components of an event-based integration infrastructure.*

The architectural concepts that define the information-driven integration approach are largely based upon the ideas of event-based systems research. This section briefly introduces the abstract key concepts of event-driven architectures in order to lay the foundation for the explanation of the adopted event-based model in the IDI architecture.

The largest difference to other approaches is the inverted model of interaction [MFP06]. The initiator of communication in event-based systems is the provider of data, the so-called *informer*, which sends event messages, termed *notifications*, to other participants. However, notifications are usually not addressed to any specific set of *listener* modules. Instead, listeners receive notifications by expressing their interest through so-called subscriptions. If a notification matches a subscription, it is delivered to its listener. The nature of the event-based interaction can be described as follows [MFP06]:

> The essential characteristics of this model is that producers do not know any consumers. They send information about their own state only, precluding any assumptions on consumer functionality. A component "*knows*" how to react to incoming notifications and it publishes changes to its own state, but it must not publish a notification with the intention of triggering other activity.

This definition underlines the aforementioned aim of increased autonomy by explicitly calling for the design of *self-focused* services from a technical perspective, yielding coherent modules that solely process information restricted to their own task and exercise control only over their own implementation. No implicit knowledge about the state of other modules must be used. Following this paradigm, the overall behavior of the system arises from the implicit interaction between the event-based system modules. Withdrawing control of interaction from the participating components, the necessary coordination has to be handled externally.

Many different realizations of EDAs, cf. [MFP06] for a recent overview, were developed in industry and research over the last decade, however, all with different focus and with just only a few gaining wide acceptance. Reasons for this may have been that software architectures were only retrofitted with event-based extensions that introduced this architectural style into otherwise operational middleware [Sie00, HS08b]. Thereby, they share many of their drawbacks, e.g, such as fine-granular event structures.

The integration architecture presented in this thesis has a similar aim that actually adopts many methods from event-driven architectures to provide an environment that combines the suitable aspects of both approaches in a coherent architecture usable within the given integration context. The assumption is that the functional composition from SOA *and* the event-based interaction form EDA is a promis-

ing foundation for the development of reusable system modules in an agile development process as needed for efficient research on experimental cognitive systems. With regard to SOA, recently the borderlines between SOA and EDA are beginning to diminish, mostly from the web services community that introduced standards such as WS-Notification.

From these concepts, particularly the notion of document-orientation is of capital importance in the context of information-driven integration. As documents actually encode all necessary information about interactions the *communicated messages are the loci of state change* yield the basis of information processing in a distributed IDI architecture. This in-band communication paradigm will be fully exploited to design an architecture that targets the aforementioned strategic aims, utilizing the concepts of service-oriented and event-driven architectures with specific extensions for memory, coordination and domain support.

6.1.4. Guide to the Reader

Several attempts to describe event-based styles and to classify event-based architectures according to well-defined frameworks have been proposed. The effort presented in [BCTW96] will be used in the following sections as a common vocabulary for referring to the core functional components of an event-based infrastructure, which are depicted in Figure 6.1.

According to this scheme, *participants* can either acts as an *informer* that sends messages encoding the occurrence of some event or as a *listener*, which is receives event notifications. Before sending or receiving any message, a participant may inform some kind of *registrar* of its intention to do so. The actual delivery of event messages is in the responsibility of the *router* component. It may contain additional elements, e.g. so-called *message transforming functions* (MTFs) and *delivery constraints* (DCs). While MTFs are in charge of transforming, e.g., filtering, messages on behalf of some listener, DCs define some extra conditions with regard to event delivery, e.g., on the order in which events are received.

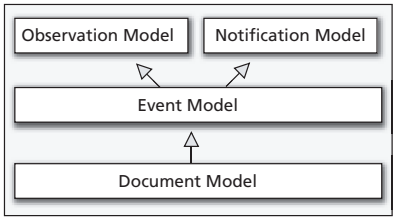


Figure 6.2.: *The adopted event-based models.*

The explanation of the information-driven architecture in the following two chapters is loosely informed along the lines of the general of event-based system proposed in [CNF01]. While not all of those models are applicable to the presented approach, e.g., the subscription model is subsumed by the observation model, and the document model is introduced as an additional model, this still provides a coarse framework for the following sections as shown in Figure 6.2. This notion of introducing models for describing coherent building blocks of concepts and technology will be continued in Chapter 7 with the extensions that are geared towards facilitating integration in a collaborative research project on cognitive systems like VAMPIRE.

In this and the following chapter, the different models and functions of the IDI architecture will mainly be explained from a conceptual viewpoint. Nevertheless, within each of these sections, interesting or important details of the architectural realization will be explained in an excerpt-like style. If implementation details are reported, they describe a general object-oriented design of an available implementation independent of a specific programming language if not otherwise stated.

In order to visualize the different aspects of the software architecture from different viewpoints, class and composite structure diagrams are used for describing static viepoints and activity diagrams for dynamic aspects. All diagrams utilize the notations of the *Unified Modeling Language* (UML). All depicted diagrams restrict the visibility of classifiers within the shown part of the software model to the absolute minimum necessary to understand a particular aspect. This is needed, because otherwise the resulting visual complexity of the UML diagrams would impede the communication of the main matters.

In order to get a grip on the concepts behind the IDI approach, let us now turn to the document model, which yields already an important technical and conceptual foundation for the overall architecture.

6.2. Document Model

The document model is fundamental for many of the advanced concepts in the information-driven architecture as event notifications are encoded in accordance with its principles. For instance, the chosen representation in the document model has a large impact on the available functionality in the observation model which allows content-based matching on information elements encoded in accordance with the document model. Hence, it serves as an underlying theme used across almost all models of the information driven integration architecture.

The IDI architecture actually employes a *documented-oriented* data-model for semi-structured information. Document-orientation is an important mean that promotes loose coupling due to the fact that the documents shall be self-contained and encode ideally all information that is needed by a participant to process a received document-based event notification, which is well suited for the asynchronous interactions in an event-driven architecture. Taking up on the point of loose coupling, document-orientated messages that follow the ideas stated in the next section, represent a variant of the *value object* pattern for distributed systems, which recommends to communicate objects that are immutable and that can be identified based on their state rather than on their object identity.

Guided by the requirements defined in 5 regarding data representation in an integration architecture and given the desired application independence and document orientation, the use of the *Extensible Markup Language* [BPSM$^+$04] (XML) as the underlying basis for data description nowadays is a quite natural choice. Using XML documents instead of a hardwired binary protocol, plain text, ASN/1 [Dub08], or JSON, which is used in web environments, has several benefits for integration such as extensibility, declarativity and standardization. The latter fostered the development of excellent tools and programming APIs for processing of XML documents. In conjunction with additional XML standards like *XPath*, *XSLT*, *XML Schema* and *XLink* [BDG01] it provides a domain- and programming language independent representation model, which is widely known and promotes openness. If XML is used as proposed in the subsequent section, it also contributes to the understandability of system-level interactions. Utilizing XML allows to store, retrieve and process information from different abstraction levels and semantic domains, yielding a *unified data model*.

Using XML structures such as the object document shown in Listing 6.1 certainly yields performance penalties for its textual encoding and verbosity. However, the hypothesis with regard to this point is that the performance loss is outweighed by the positive impact on many of the introduced requirements and that XML construction and access is fast enough for performing system-level integration in real-world cognitive systems as shall be substantiated in Part III of this thesis.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<OBJECT>
  <HYPOTHESIS>
    <GENERATOR>Object Recognizer BU(N)</GENERATOR>
    <RATING>
      <RELIABILITY value="0.6"/>
      <RELEVANCE   value="0.5"/>
    </RATING>
  </HYPOTHESIS>
  <CLASS>Cup</CLASS>
  <REGION image="img_office210703_122">
    <RECTANGLE x="335" y="245" w="65" h="80"/>
  </REGION>
  <CENTER x="32" y="44"/>
</OBJECT>
```

Listing 6.1: *Example of a basic object recognition event as used in the VAMPIRE systems.*

Even so, as cognitive computer vision systems make wide use of subsymbolic information, binary data like images would need to be encoded in an XML document by treating it numerically and translating it into a *base64* representation, which is unacceptable. Thus, the event model in Section 6.3 introduces the notion of an *attachment*, which handles this class of data more efficiently and binds it to the event notification concepts.

From a collaborative perspective, XML schemas help to check exchanged XML documents for their validity. XML Schema files define a grammar that a valid XML document has to conform to and can be used in collaborative project to formally define messaging contracts between interacting participants. Excerpt 6.1 briefly highlights the use of schemas in the COGNIRON project.

Since XML only specifies the syntax of a document, let us discuss a general strategy for how information shall be encoded in the IDI architecture and what the resulting implications for the information exchange in integrated cognitive systems are if extensibility is taken into account.

6.2.1. Information-oriented Representation

In contrast to other approaches like *tuple-*, *record-* or *object*-based event models [MFP06], the XML data model as defined by the *XML Information Set* [CT04] (XIS) specification is a hierarchical tree-structured data model. The XIS recommendation defines a number of information items, primarily the *document* information item as the single root node as well as *element* and *attribute* items that can be used as nodes in a tree to model hierarchically structured event information like the availability of object recognition information as exemplarily displayed in Listing 6.1.

A hierarchical model was supposed to be well suited for the encoding of scene information and other important data in the context of the VAMPIRE project. While tree-models lack the expressiveness of graphs, they were chosen as a compromise for the sake of clarity and simplicity while already allowing for greater flexibility in describing events than a tuple- or record-based approach.

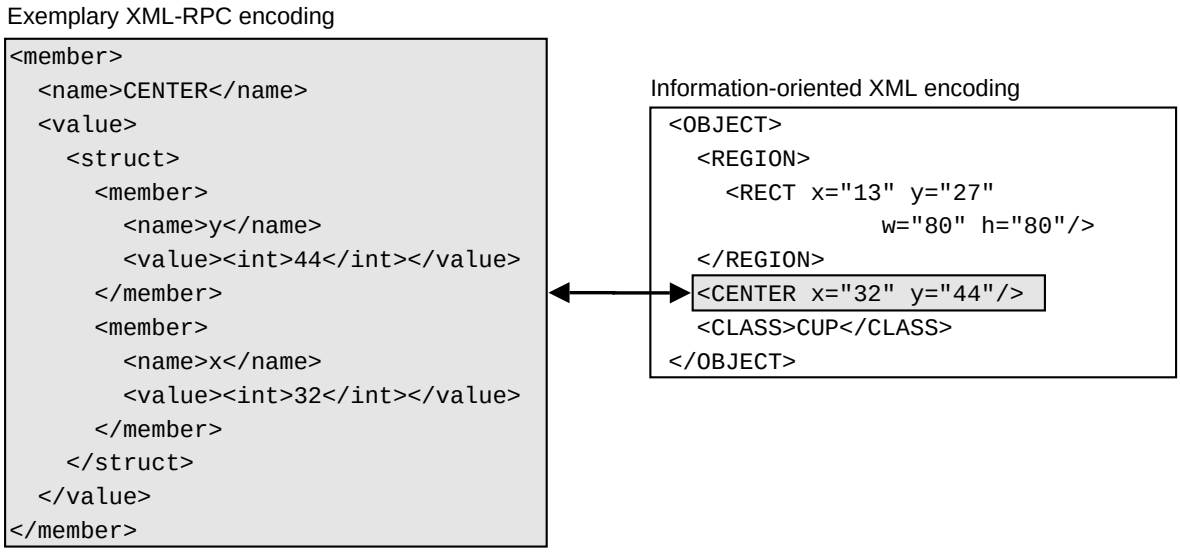


Figure 6.3.: *Contrasting XML-RPC with document/literal information encoding.*

The document model assumes that the different functional modules of a cognitive systems generate symbolic information from sensing their environment, which permits them to encode these results, i.e. object percepts, as XML documents. These include results of for instance *object localizations*, *spatial relations* of objects, and *actions*. As different sensors often share the same attributes, parts of the knowledge fragments are common for different documents and some are specific for the respective type. These *shared representations* allows processes to handle different types of knowledge fragments transparently, since they can only consider the data relevant for their processing.

An example of a shared representation that was widely used in the software architecture of the VAMPIRE project is the *Hypothesis* [HBS04]. Since perceptive modules typically do not provide complete accurate results, cognitive systems should not process information as irrevocable facts but as hypotheses with a given *Reliability*. This common data structure describing the uncertainty of a knowledge fragment are called the *Metadata* of the hypothesis. Listing 6.1 shows an example of a hypothesis containing a common metadata part. As this metadata is available for any kind of hypothesis, processes are developed that only consider this information and can therefore handle any kind of hypothesis, as for instance the “forgetting”-process that is described in Section 7.3 in more detail.

Information-oriented representations conform to guidelines on the design of interactions in service-oriented architectures to enforce loose coupling. Ideally, messages in the IDI architecture shall be:

- *Reference free*: Representations containing reference types to data structures that are in the state space of other distributed participants shall be limited to the extent possible.
- *Feature a coarse granularity*: Messages should at best be of coarse granularity making it not necessary to issue a sequence of fine-granular message exchanges. In general, they should contain all information that is needed to process them in a cognitive system architecture.
- *Free of technological details*: The encoded information must not contain details about the specific component implementation that generated the message.

The use of the XML Infoset as the unifying data model for the IDI architecture enforces the desired document-orientation and contributes to the required extensibility of event interaction. Please note that XML is used here differently as in data exchange protocols like XML-RPC [KAU04]. Those protocols are often designed from a programming language or solely from a marshaling perspective and usually result in a lot of overhead through text-based representation of binary data and parameter encoding rules which in turn leads to poor understandability of the textual XML representation.

Looking at the object recognition example shown in Figure 6.3 you see two alternative encodings of a "CENTER" element. The example shows an *information-oriented* encoding of the center item (embedded in an object recognition result) on the right as well as a serialization of the same item in XML-RPC encoding on the left. It is not only this obvious overhead induced by a naive serialization of data to XML that is not desirable, but even worse is the loss of comprehensibility at all processing levels that is imposed by this type of encoding.[1] Using XIS as a unifying data model and XML as the publication language in the IDI architecture not only yields great flexibility in event notification encoding, but at the same time provides a programming-language independent representation of information.

Excerpt 6.1: Event Specification and Validation

Meta-information, e.g., about allowable data types, is kept separate in corresponding XML schema files and is not encoded in the instance documents themselves. Specifying data types with XML schema has several advantages in contrast to traditional programming language constructs.
First of all, the data types are independent from specific programming languages. Even so, tools for using them are available on almost every platform. Furthermore, XML schemas are able to specify content models and ranges of allowed values in great detail. Providing fine grained sets of semantically grouped declarations in separate schemas with associated XML namespaces makes them reusable throughout different systems. Complex schemas for individual modules can then easily be composed out of these basic type libraries, only adding specific complex types. If taken into account, extensibility of data types is possible with schema evolution. Even complex grammars for components capable of interpreting and validating XML documents originating from different robot modules are easy to compose and well understandable with a sophisticated schema hierarchy. Information-oriented encoding of XML event notifications and the use of XML schemas for validation of the exchanged information are both very useful for system integration in interdisciplinary research projects. The focus on simple XML messages to describe exchanged information helps during project inception as almost every developer will be able to contribute to the discussion about the data flow in the system. Later on, XML grammars like XML schema allow for a rigid specification and validation of the datatypes a project consortium has agreed upon. For example, schemas have been defined in the European project COGNIRON [Cog06] to ease the integration of the partner's contributions in the realized robot prototypes.

[1] The idea to directly encode information and not data in XML has recently gained more interest even in the Web Services community where SOAP-RPC encoding (which is somewhat similar to XML-RPC encoding) is being more and more replaced by the document/literal encoding as favored by the approach presented here.

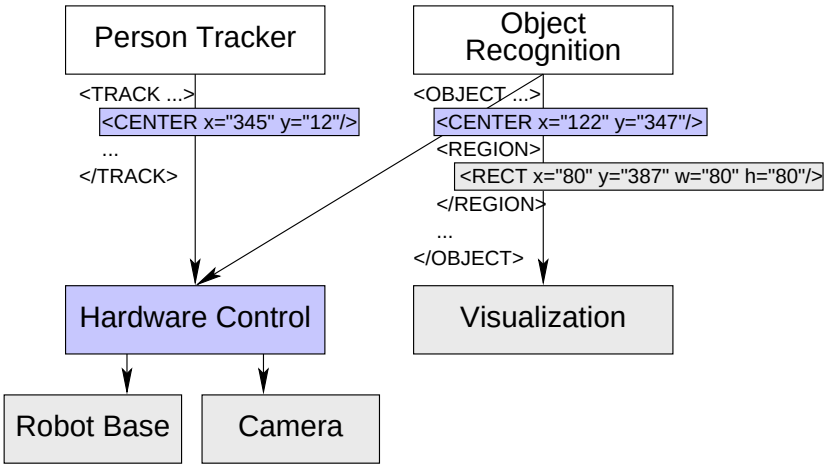


Figure 6.4.: *Accessing common information at arbitrary locations with XPath.*

6.2.2. XML Processing and Extensibility

To promote the goal of loose coupling by using self-contained XML documents, the policies for handling XML structures are critical. In the IDI architecture, a two-fold strategy is proposed. On the one hand, modules that receive an event notification must not remove information they do not understand. In contrast to the must-understand policy, the IDI architecture proposes a *must-ignore* policy, which means that unknown document structures are not interpreted as erroneous but are left in place but possibly changed or augmented by additional information before they are communicated to other system components. On the other hand, the answer comes from the recommended way of accessing the information in an event document. In contrast to template-based access like in tuple-based approaches or the direct reference to a field in a record-based data structure, a selection of XML information items through XPath expressions [CD99] helps in building extensible systems that will not break as soon as modifications of the event encodings occur. Carefully designed XML vocabularies and path-based access therefore are important methods to overcome prevalent versioning problems [SV01].

An example that explains the benefit of the XPath-based selection of XML information items for system integration is depicted in Figure 6.4. Four modules processing partially equivalent XML data structures are shown. This allows a component (e.g. “Hardware Control” for adjusting the robot base and the pan-tilt camera) to process information from different other system modules. Only the part in the XML document that contains information about center points (“`<CENTER>`”) has to be present in an exchanged data item. Starting with a simple partial path specialization to access the context node, e.g. in this example as simple as “`/*/CENTER`”, the extraction of contained information is easily possible although the context node itself might appear in varying places of different XML structures. Thus, this path expression works on both documents shown in Figure 6.4.

As long as no necessary information is removed, this strategy yields loose coupling and facilitates interoperability between separately developed modules. This serves as an initial example of how the XML infoset-based representation of event notifications contributes to the goals of the requirements identified earlier. The path-based access to information will be revisited as a fundamental part of the content-based observation model in Section 6.4.

6.2.3. Exploiting Reflection

Reflection is a general programming language concept, realized for instance in Java and Smalltalk, which allows applications to query information about objects and classes at runtime. In terms of middleware, e.g., dynamic CORBA [Sie00] supports reflection at runtime using, e.g., the dynamic invocation facilitates as explained in Chapter 3. However, even CORBA advocates admit that these reflection facilities are "*hard and tedious*" to use. Even so, they argue that this is a rather small problem in practice for CORBA applications, because most of them exhibit a static character utilizing stubs and skeletons generated by an IDL compiler. Hence, CORBA clients and servers generally already know all necessary type information to evaluate received data and issue calls on remote objects. In fact, they *must* be aware of this information as CORBA's *General Inter-ORB Protocol* (GIOP) omits type information from requests, requiring participants to know messages types a priori.

Excerpt 6.2: A Content-based Similarity Metric

Exploiting reflection in the XML-based document model, we developed a similarity measure that includes a data integration mechanism and can thus process data from a variety of sources coherently. It uses the label information in XML document trees, the element name, to identify comparable values and to transparently handle missing, repeated or re-ordered occurrences of an element or sub-tree.
In any data integration task, care must be taken not to mix up data with different semantics. E.g., in object recognition, the coordinates of an object and its label are not on the same abstraction level. Therefore, the hierarchical *nesting* as a generic indicator of semantic differences is exploited, taking advantage of an existing and established way of formulating this crucial bit of information.
Hence, the similarity measure constitutes a kernel. It has been shown that many machine-learning methods can be *kernelized* in a straightforward manner, either by using the kernel in place of the scalar product or through a distance measure constructed from the kernel, e.g. $d(x,y) = \sqrt{K(x,x) - 2K(x,y) + K(y,y)}$[Hau99].

Kernel Over XML Documents An XML document is a labeled tree rooted at the *document node*. In the following, for a node n, let $L(n)$ be its label, $V(n)$ its value and $C(n)$ be the set of children and attributes. In the XML infoset, only attributes and text nodes have a value assigned but for the purposes of this paper, we take element value to be composed of the immediate text nodes:

Definition 6.1 (Element Value) *The value of an element n with level l is the concatenation of all text nodes with root n and level $l+1$.*

For the kernel definition, two cases are special: The empty comparison and non-matching labels. For these, $k(0,0) = 1$ respectively $L(s) \neq L(t) : k(s,t) = 0$.
For nodes, we adopt the idea of Gärtner et al. [GLF04] to exploit possible functional dependencies by combining the similarity of parent and children:

$$k(s,t) = k_{L(s)}(V(s), V(t))k(C(s), C(t))$$

Nodesets, despite the name, have document order but may be treated as both a set or a list, with the corresponding kernels (and using the above). For sets: $k(u,v) = \sum_{i,k} k_n(u_i, v_k)$ and for lists: $k(u,v) = \sum_i^n k(u_i, v_i)$. Last, but not least, for basic numeric values, a Gaussian: $k(a,b) = e^{-|a-b|^2/h^2}$ and for strings, a Hamming similarity: $k(m,n) = 1/k \sum_{i=1}^{k} \delta(m_i, n_i)$ is applied.
While this can be extended with kernels for domain-specific information, the mapping of the basic XML infoset items already yields a variant of a similarity metric that can be applied in a general manner. How this can be used for the purpose of clustering similar XML documents will be explained in Excerpt 6.3.

While in CORBA and other operational middleware, this information is typically used to perform dynamic invocations, the main utility of reflection for the integration architecture is different. As event-based systems do not feature operational semantics, the focus is set on the dynamic interpretation of event notifications. With regard to that, the self-describing nature of XML documents yields a kind of type reflection, which allows to develop generic modules which would, e.g., in CORBA only be possible by exploiting its full complexity.

Excerpt 6.2 gives an example how the reflective characteristics of the XML approach can be exploited to design an XML kernel for measuring the similarity between two documents in a generic way.

6.3. Event Model

The IDI architecture introduces a generic event model to allow participants in a system to signal relevant event occurrences independent of a specific abstraction level, content and operational context. *An event describes any occurrence of a happening of interest that an informer in a cognitive system wants to communicate instantaneously to listener participants*. Types of events in the systems that are discussed here range from the presence of new sensorial information like the availability of new laser scanner data sets to the detection of an unknown person entering a scene, which is visually observed by a surveillance system or the encoding of actuator state changes. *Events* represent the central abstraction that is used by module or system developers to model their domain objects in an IDI-based system.

However, not only developers but also the proposed architecture is itself fundamentally based on event concepts in order to provide higher-level integration functions like the services offered by the memory model that will be described in Section 7.3. The higher level components of the architecture and the application specific modules in an IDI system communicate by generating and receiving *event notifications*, which actually are physical representations of events in terms of programming language constructs, e.g., an object instance in an object-oriented class hierarchy.

Figure 6.5 depicts the structure of an event notification as the fundamental mean of communication in an IDI architecture. As in many other event-based systems, an event encodes *information* that conveys the semantic information associated with an event. Based on the principles of the document model, each notification shall be self-contained to facilitate efficient asynchronous communication, which is a key characteristic of event-based systems. Therefore, the message content is represented according to the document model in exactly one underlying instance of an XML document.

Rephrased in terms of the event-based system paradigm, XML is used here as *publication language* for event notifications. An individual notification n is thus constituted by a set of attributes $a_1, ..., a_n$ available through the event document m where the value of information items in the underlying tree model is accessible through associated pairs (x_i, ns_i) of XPath expressions x. The corresponding nodesets ns are returned by evaluating XPath expressions with their types conforming to the XPath 1.0 data model [CD99]. This generic accessibility allows for the introduction expressive content-based filtering functions in the observation model of the IDI architecture.

Besides the message and the underlying document, an individual event is consituted by *metadata* and a number of optional *attachments*, which are explained in the following before the introduction of user-definable event notification types in object-oriented programming languages will be explained.

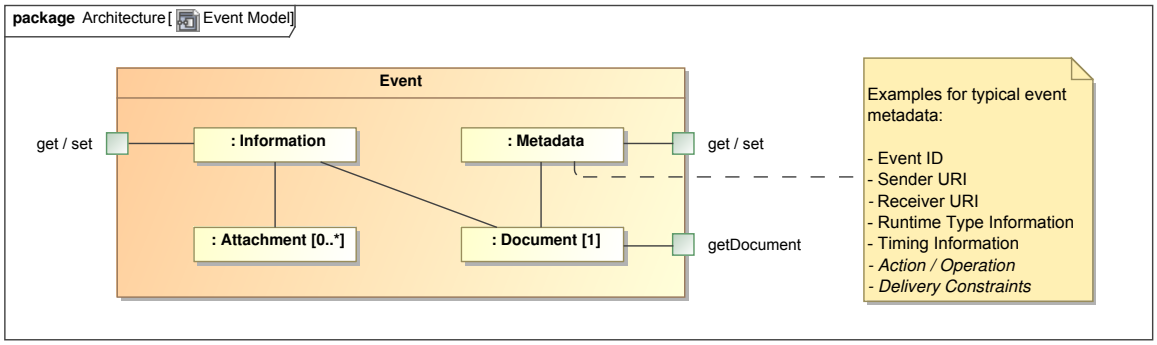


Figure 6.5.: *Composite structure diagram for the document-oriented event model. An event object is constituted by domain-specific information, referenced binary attachments and metadata. Attributes in italics are examples for optional metadata elements.*

6.3.1. Event Metadata

Apart from the basic data model that module developers use to encode information in event notifications, the architecture itself inserts an extensible *metadata* block into the event representation as exemplarily shown in Listing 6.2. Conceptually, this metadata is a dictionary featuring an additional set of predefined information-oriented attributes that are augmented to event notifications by the integration architecture in order to facilitate advanced functionalities like scoping, which is a feature of the notification model explained in Section 6.5, which restricts visibility of notifications or to enforce other constraints such as delivering only recent hypothesis to a listener.

The metadata elements are located in a separate XML namespace [BDG01] (see *line 4* of Listing 6.2), thereby protecting them from other information encoded in an event representation.

Typical metadata attributes are identity information for the event itself, which is represented as a *Universally Unique IDentifier* (UUID) that serves as a *Uniform Resource Name* [BLFM05] (URN) for individual event notifications (*line 4*). The identity of informer and optionally listener participants is encoded in accordance to the URI scheme of the resource model that will be explained in Section 7.1 (*lines 5 and 6*). Furthermore, the time of event publication and retrieval (*line 7*) is added by the IDI architecture as it is fundamental for synchronization in this architecture. Adding up to this, specific delivery constraints (*line 8*) like the *Time-To-Live* (TTL) information that specifies a lease time in which an event remains valid or runtime type information (*lines 9–11*) are represented in the metadata structures.

By integrating this metadata into the documents itself the architecture complies with the goal to interact through self-containend event notifications. This has the advantage that the core architecture may largely use similar processing strategies as they are offered to module developers. Details about the use of different metadata attributes will be explained in the subsequent sections along with the description of the higher level functionalities they contribute to.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<OBJECT>
     ...
    <xcf:meta eid="d96f80c9-c1b2-4519-a6bc-d7b250490af5" xmlns:xcf="http://xcf.sf.net/">
        <sender uri="xcf://or.vampire.agai/boost"/>
        <receiver uri="xcf://vampire.agai"/>
        <timing pub="1209295281648" rec="1209295294015"/>
        <dc ttl="100" timeunit="ms" />
        <event type="xcf.event.PublishEvent">
            <isa type="xcf.event.XcfEvent"/>
        </event>
    </xcf:meta>
</OBJECT>
```

Listing 6.2: *Exemplary XML metadata element that is transparently attached to each event notifications by the integration architecture.*

6.3.2. Optimized Packaging of Binary Data

As pointed out in the beginning of this section, a regular counterargument against the use of XML is the inefficiency of its textual serialization. The IDI architecture extends over the purely infoset-based model along two dimensions in order to provide improved performance while keeping most of the benefits: on the one hand, an optimized packaging of binary data in XML documents is used to increase the efficiency of serializing event notifications and to save bandwith on the transport medium. On the other hand, the notification model, cf. section 6.5, introduces a transparent optimization scheme that allows transport-layer specific serialization strategies, which may dramatically increase efficiency of marshalling event notifications. As part of the event model, let us now focus on the efficient handling of binary data.

As document-orientation and XML encoding is fundamental for many aspects of the IDI models, the decision was taken to keep this scheme for as much information as possible but treat *Binary Large OBjects* (BLOBs) like image data or general array-like data types differently. Thus, as depicted in Figure 6.5, BLOBs can be added to an event notification with *attachments* that can be referenced and described in the event document and are transmitted in the native encoding of the underlying transport layer implementation. This style of adding attachments to XML documents is inspired by the *XML Optimized Packaging* [GMNR99] recommendation.

Marshaling of attachments is directly supported for a number of predefined vectorial data structures, e.g. vectors for floating point types. Additionally, a container type for unsigned character data is provided which may be used to communicate any user-defined data structure efficiently in an event notification. However, this implies that the module developer already serialized domain specific objects into a byte vector representation as besides byte ordering no further operation on the data is carried out by the implemented serialization strategy.

However, as binary data formats re-introduce versioning issues and are not accessible in a generic manner, it is recommended to limit the use of binary-encoded elements and better describe relevant event data either in an information-oriented symbolic event description or to extend the metadata

dictionary. More details about attachments and the marshaling of events in one specific transport implementation will be briefly presented along the explanation of the notification model in Section 6.5.

6.3.3. Domain Events

The IDI architecture defines a logical taxonomy of event types, tailored to its own functional needs, e.g., defining event types for system management and the like. This taxonomy is realized in an object-oriented class hierarchy, which wraps the domain information that is contained in an the event notification document into a programming language specific interface. In comparison to the generic XML-based document interface, these wrapper objects increase usability and allow for polymorphic dispatch of event notifications in object-oriented programming languages, cf. Section 6.5. Besides usability concerns, the strategy that is presented in the following, dramatically increases the efficiency of read access to event data due to the fact that it is carried out as a language specific operation on cached data value objects.

In order to be extensible for module developers, implementations for domain specific event types can be added to the event taxonomy. As event marshalling is handled in the transport layer 6.5, the necessary mechanisms to deal with user defined event types have to be provided there. Event notifications are converted in language-specific object representations if serializers for the received event or one of its super types are available in the used programming language. Even though it is in general possible to use different technologies to realize the transport layer, it is mandatory to provide the backing XML document that encodes the data of the event notification, because this is used as a fallback information source if a specific function of any of the IDI models cannot be mapped to an optimization, e.g., in the transport layer. For instance, scope or identity information are supported to a very different extent by underlying middleware implementations.

The IDI architecture's standard implementation of the event marshalling mechanism in the transport layer uses a rather simple XML plus binary object serialization to transport the event notification. As event type as well as its super types are encoded in the metadata block of the document, this allows to instantiate the most specialized type that is available upon unmarshaling of a serialized event notification. User defined event types can be dynamically registered with the transport layer in order to be serializable, but since all events have to be derived from a base type provided by the IDI architecture, deserialization of the event base type is always possible. This yields an important difference to stubs and skeletons as known from operational middleware because event notifications can be interpreted in a language specific way but participants can communicate even if no specific type information is available.

The native event implementations use a *location*-based data-binding concept to access parts of document passed to them, therefore no further marshalling is necessary. Locations provide type-safe and cached access to their values. They also allow for optimizations in the transport layer as it will often be possible to determine certain values, usually metadata, using native mechanisms of the underlying communication technology. Values that have been cached by a location either because they have already been extracted from the document or because they have been set in a more efficient way will not be extracted again. Based on the assumption that writes to an event structure occur much less frequent than reads, the underlying document and the cached content of a location are both updated during the setting of a value. This representation of event notification provides an avenue for efficient but expressive filtering in the observation model that shall be introduced next.

6.4. Observation Model

The observation model describes the concepts that allow a listener component to express its interest in specific events produced by informer components and the mechanisms that are involved in the process of propagating incoming notifications to listeners. A consumer specifies the events it is interested in through a *subscription* [MFP06], which is in this approach registered at the local router component of the IDI middleware stack in the address space of the participant and evaluated against all incoming notifications. If a subscription matches a notification in the evaluation process, the contained event is dispatched to the callbacks of the listener components that are associated with this subscription.

Event-based architectures usually support a subset of one of the following mechanisms for subscribing to event notifications [MFP06, Fai06, BCTW96]:

- *Type*: Matching based on runtime type information. In many architectures the event type refers to the type of data that is contained in an event notifications.
- *Channel*: Subscription matching based on a physical or an abstract communication channel. Different event types can be published without further selection.
- *Group*: Event matching based on group memberships. Subscribers belonging to a group will receive the same set of event notifications.
- *Subject*: Instead of operating on the payload of an event notification, pattern matching operations are applied to event descriptors.
- *Filter*: Subscription models that utilize filters allow listeners to restrict published notifications to a suitable subset.

Filter-based approaches provide high flexibility as they allow to narrow down the received set of notifications for an *individual* listener. In event-driven architectures that support this concept, listeners can specify subscriptions as stateless boolean-valued *filter* functions operating on a single notification. By applying a test on specific properties of the incoming notifications let alone whether this is based on the whole content, certain attributes or an expected sequence [Fai06], they return either $true$ or $false$ representing the success of the filtering operation.

The IDI architecture permits all the aforementioned ways of subscribing to events by introducing a number of predefined but generally applicable filter types and a filtering subsystem that transparently optimizes the matching step if possible by exploiting platform specific operations. In cases where this is not possible a content-based filtering approach is applied that is based on the introduced information-oriented representations introduced in the previous sections. This strategy will be explained in more detail in the following.

6.4.1. A Hybrid Subscription Model

A central question from the viewpoint of the observation model is what elements actually comprise a subscription. At this point, the notion of a filter is extended towards a more generic *message transforming function* (MTF).

An MTF differs significantly from a traditional filter in a sense that it is stateful and (optionally) applies a transformation to the notification message it receives as input. An MTF shall in this approach be defined by

$$\text{transform}(X, m_i, \phi_i) = (\{m_{i+1}\}, \phi_{i+1}) \tag{6.1}$$

where X is an individual participant, m_i is a message delivered to component X, ϕ_i is the state of the transform function before processing of m_i. If an MTF is not able to apply a transformation a message, it returns an empty set as result of this transforming step.

While the abstract concept of a transforming function in event-based systems was originally introduced by Barret et al. [BCTW96], the definition introduced here differs in that the individual MTFs are not directly associated with individual listener instances and are defined here in a sense that provides a generalization of the non-mutating filter concept found in the aforementioned subscription models of event-based systems. While this constraint does not allow, e.g., to re-route messages to other listeners, it enhances the observation model with capabilities that provide the basis for advanced integration functionality.

Particularly MTFs that are stateful and the ability to re-structure event messages allow to encapsulate integration functionality like simple frequency filters or more advanced concepts like a novelty detection as described in Excerpt 6.3 in re-usable software components that can neatly be integrated into the observation model of the IDI architecture.

In order to observe specific events that occurred in a system, e.g., that the interaction partner's face was detected in front of the VAMPIRE AR gear's cameras with a certain probability, listener components can register expressive subscriptions that specify the conditions that describe a listener's interest in observing specific state changes in an information-driven system architecture.

In this example, the subscription would possibly feature a type filter matching representation of faces, which would be linked to a reliability filter in order to discard notification messages containing detections with low probability. One could even imagine to consider this request only if a number of occurrences of observations matching the previous filters happenend during a short time interval in order to be sure that someone is interested in attracting the robot's attention.

The subscription process in an event-driven architecture is consituted by all steps necessary for a listener to become subscribed at runtime to event notifications issued by an informer. From a developers perspective, the above mentioned example can be expressed and registered in the IDI architecture with a composite subscription as shown in Listing 6.3. This subscription basically represents a mapping of the stated functional condition to instances of available filter types. While the `TypeFilter` (*line 3*) usually is a programming language-optimized filter, e.g., by exploiting polymorphic dispatch [BCH+96], the reliability checking (*line 4*) yields an example of content-based matching with a generic filter that is based on XPath expressions on the document that is contained in each event notification. If all filters match, in this example an instance of the matching face event notification is pushed into a type-safe queue for face events.

```
1 SynchronizedQueue<FaceEvent> faces = new FaceQueue();
2 Subscription s = new Subscription();
3 s.append(new TypeFilter(FaceEvent.class));
4 s.append(new XPathFilter(new XPath("//HYPOTHESIS/RATING/RELIABILITY[@value>=0.95]")));
5 s.append(new FrequencyFilter(10,1,TimeUnit.SECONDS));
6 // add subscription to router object
7 r.subscribe(s,new QueueAdapter<FaceEvent>(faces));
```

Listing 6.3: *Java example of a filter chain representing an event subscription for faces detected with a high reliability and frequency. If it matches, the event is dispatched to the registered callback handler, which is here a generic adapter that appends detected events to a queue.*

6.4.2. Transformation-based Event Filtering

In the spirit of this example, a subscription can more formally be specified as a *filter* function, which is defined here as a concatenation of multiple transforming functions operating on a single message

$$\text{filter}(X_s, m_1) = \text{transform}(X, m_1, \phi_1) \circ \ldots \circ \text{transform}(X, m_I, \phi_I) \tag{6.2}$$

where I denotes the number of MTFs that constitute the subscription s at component X. As for a single transformation functions, a filter returns an empty set if a message is either not matched by any of the registered functions or if it is intentionally "*consumed*", what is for instance actually done in the frequency filter implementation.

As in most event-based infrastructures, a router actually performs the matching of incoming notifications. It maintains the list of active subscriptions for an individual listener and needs to decide whether one of these matches a received notification. This is done in the IDI architecture for an individual subscription by an evaluation of the following function

$$\text{match}(X_s, D_X) = \text{filter}(X_s, n_1) \circ \ldots \circ \text{filter}(X_s, n_I) \,|\, \forall n \in D_X \tag{6.3}$$

where D_X is the set of notifications that are received by component X in an event-based system in the given time interval. If this function evaluates to an empty set, this indicates that the notifications were discarded and the conditions expressed through the subscription were not fulfilled during the observation interval.

If a subscription matches a received notification an optionally transformed message is returned. In turn, the IDI architecture dispatches the represented event to the registered local callback handler as shown in Figure 6.6. These handlers bind application logic of an individual component either directly to the integration architecture or insert events in synchronized queues, cf. Listing 6.3, that allow for an subsequent retrieval of information by the component, effectively inverting the local event notification semantics.

Name	*Description*	*Optimization*
XPath	Content-based matching with XPath expressions.	Content
XSLT	Generic message transformation using XSLT scripts.	Content
Reliability	Evaluates notifications against a certain probability threshold.	Content
Compacting	Compares novelty of received notifications against previous ones.	Content
Identity	Matches on unique identity information.	Content
Scope	Reduces the visibility of events by introducing scopes.	Transport
Type	Matches on event types and sub-types defined in the event model.	Language
Frequency	Filter that outputs only every n-th received notification.	Language

Table 6.1.: *MTFs supported in the IDI architecture and their layer of optimized execution, e.g. whether they can be evaluated on the content level or on the network level.*

These two different styles of interaction between software modules are often refereed to as *push* and *pull* communication [Fai06]. While in the former model, the software framework calls the component (in conformance with the Hollywood[2]-principle as known in software engineering), which couples the temporal behavior of informer and listener, the latter style allows the component to perform an asynchronous event processing in accordance with its internal information processing architecture, actually pulling events from corresponding queues.

The independence of the individual MTFs allows to optimize the transformation and matching process within the router component. In the IDI architecture, the current realization of the matching strategy is based on a direct acyclic graph structure (contained in the *MTFTree* as shown in Figure 6.7 whose current implementation is explained in greater detail in Excerpts 6.4).

It currently supports grouping of identical filters as long as those are equal or covering the same subset of notifications, which implies that the filter instances must produce exactly the same matchings as well as the same transformations results. If stateful transforming functions want to be grouped, this is only possible if they feature the same state at the time of merging. Covering or merging is supported only for a subset of these filters and is possible future work.

[2] *Don't call us, we call you.*

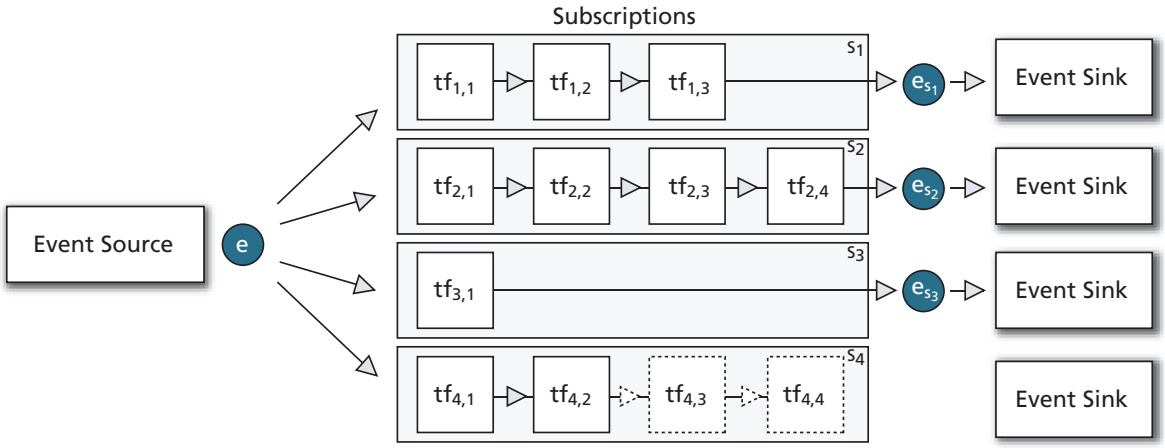


Figure 6.6.: *Simple event notification matching and transformation. If a notification message is successfully matched, the event is dispatched to an event sink, e.g., to the registered local callback handler. MTFs $tf_{4,3}$ and $tf_{4,4}$ are not executed as $tf_{4,2}$ did not match the event notification.*

Excerpt 6.3: The Compacting Filter

To reduce the burden of redundant and/or bad results in a system architecture, the aim is to filter elements based on their level of similarity. Firstly, the amount of new information present will be estimated utilizing the XML kernel as introduced in Excerpt 6.2 and only if a change is *big enough* elements will be forwarded for further processing. Secondly, elements are *clustered*. When a close group is found, it is updated, otherwise a new group will be created. This is called *compacting*.
Compacting at the level of the integration architecture allows to take advantage of global information, e.g. when two redundant recognizers are present. For the developer, it is beneficial to have a dedicated component for relevancy detection that can be changed to adapt to new challenges. Last, but not least, our approach allows components to selectively bypass compaction to receive all elements.

Detecting Relevant Elements Detection of relevant elements requires an indication of the amount of new information contained, relative to the elements already present in the memory. We use the violation of the present clustering to determine significance: New clusters are considered relevant. To determine this, we observe the minimum distance between a new element and the existing clusters over time and estimate the change using a moving average for the parameters of a normal distribution $p_I \sim \mathcal{N}_{\mu,\sigma}$. Let I be the current number of elements, and d_i the minimum distance observed at element number i, then the sample mean is $\bar{\mu}_I = \frac{1}{k}\sum_{i=I-k}^{I} d_i$ and sample variance analogous. A new cluster is created if $p_I(d_{i+1}) < t$. The parameter k allows for adaption to the result rate of the system, in our experiments it is based on frame rate. t has been chosen constant (0.05), with the variability in the system captured by the density p_I.

Online Clustering E.g., in the VAMPIRE system events about detected objects arrive one-by-one, not batched and due to user interaction stationarity can only be assumed short-term. The relevancy detection determines creation of new clusters but aims at fast reaction time more than at clustering quality and it has to, because of the limited amount of information and the strictly limited processing time. Fortunately, over time good clusters will acquire more support while outliers won't and this can be used to achieve good clustering quality in an online setting by determining cluster size and removing unreasonably small ones. The exact cut-off to choose depends on the variance in the input. In experiments the mean has proven a good choice.

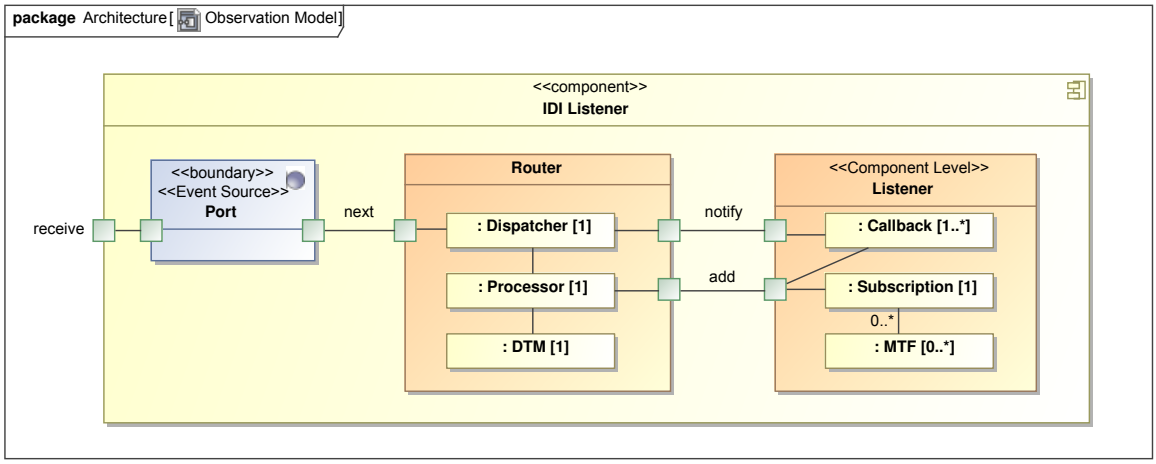


Figure 6.7.: *The conceptual architecture of the event observation model. The names of the composite classifiers were chosen in accordance with the framework of Barret et al. [BCTW96].*

Although the matching strategy can be treated as a black-box component that can be realized in a number of different ways, its interface must at least permit for dynamic (de-)registration of subscriptions and callbacks at runtime as this is necessary for the required level of dynamism that is needed in the IDI architecture for adaptation and orchestration of system components. A further benefit of this IDI concept is that MTFs can externally be injected in the router components of participants, which allows for the ex-post adaption of integrated components without the necessity to change the underlying source code.

The local callbacks that allow a particpant to actually connect its application code to the IDI architecture represent the *event sinks* in the observation model. These sinks finally receive the notification that matches a specific subscription.

On the contrary, the *event source* as shown in Figure 6.7 is the source of notifications. Thus, a single subscription is solely a logical connection from an event source to an event sink. Viewing an individual matching process as a transformation of a notification from an event source to an event sink is the basis for abstraction from the concrete *connector* types for sources (and sinks as we will see in the next chapter), e.g., network communication, shared memory or in-process communication.

The IDI architecture defines a number of filters that are directly reusable by system integrators. Table 6.1 gives an overview of these MTFs and summarizes their functionality. Some of these will be explained in more detail in later sections as they are used within the architecture to permit further integration functions. Users of the framework can transparently enhance the observation model by providing additional event types as well as filter functions that operate on these.

Last but not least, the dispatcher component as shown in Figure 6.7 is responsible for multi-threaded invocation of the callback functions that are associated with matching subscriptions within listener components. As part of the technology mapping, the multiple dispatch-based design of the dispatcher will be described in the next chapter.

Excerpt 6.4: Dynamic Tree Matching For Efficient Event Notification Transformation

The current implementation of subscription matching organizes the transforming function chains of the registered event sinks, e.g. event listeners, into a tree structure. In the most basic case, where subscriptions do not have any filter elements in common, this tree corresponds to the organization depicted in Figure 6.6. Each subscription's transforming functions are attached to the tree's root node as a separate branch, the event sink forming the leaf. An overview of the elements involved in the implementation is given in the class diagram shown in Figure 6.8.

Transformation When an event notification becomes available at the `EventSource` connected to the observation chain, it is picked up by the `EventProcessor` which provides the necessary processing threads to perform the transformation. The processor passes the notification to the `TreeMatcher` that implements the tree-based matching algorithm. It traverses the `MTFTree` structure recursively, depth-first, handing the result of the last step to the next `MTF`. If a node returns an empty result, the recursion stops for the current branch. A subscription has matched when the recursion reaches a leaf node. The `EventSink` contained in the leaf is then stored together with the result of the transformation as returned by the corresponding branch into a list of `Match` objects. Once the `TreeMatcher` has traversed the whole `MTFTree`, the resulting list of sink-event pairs is returned to the `EventProcessor` using the *Iterator* pattern [GHJV95]. The final dispatching of the transformed notifications to the corresponding sinks is then done by the `EventDispatcher`, allowing to implement a different threading strategy to process user-implemented event handlers.

Isolation Since the incoming event notification may be transformed by any of the tree nodes, it has to be copied before applying changes in order to prevent side effects in sibling nodes. In order to reduce the complexity of implementing the `MTF`'s transform function, copying is done by the `TreeMatcher` before passing the event to the `MTF`. This step only has to be applied where the tree branches, nodes without siblings may be passed the event without copying it before. For reasons of efficiency binary data contained in the event notification is not copied, in case an `MTF` needs to transform attachments, it must therefore handle the copying of these by itself before applying any changes.

Optimization Organizing subscriptions into a tree structure opens room for performance optimizations by collapsing common parts of different subscriptions. Collapsed `MTF`s have to be processed only once. Such optimizations can only be applied to subscriptions (or a prefix of the transforming function lists) that apply exactly the same transformations to event notifications. Especially stateful `MTF`s have to be treated with great care. Analysis of a given tree in order to automatically find pairs of collapsible branches is not covered by this thesis and may be explored in future work.

It is nevertheless important to note that the evaluation of subscriptions and their contained message transformation functions is executed locally in the process of the listener components. How a router as presented here is connected to its informers, the event sources, will be explained in the following section that shall explain the notification model.

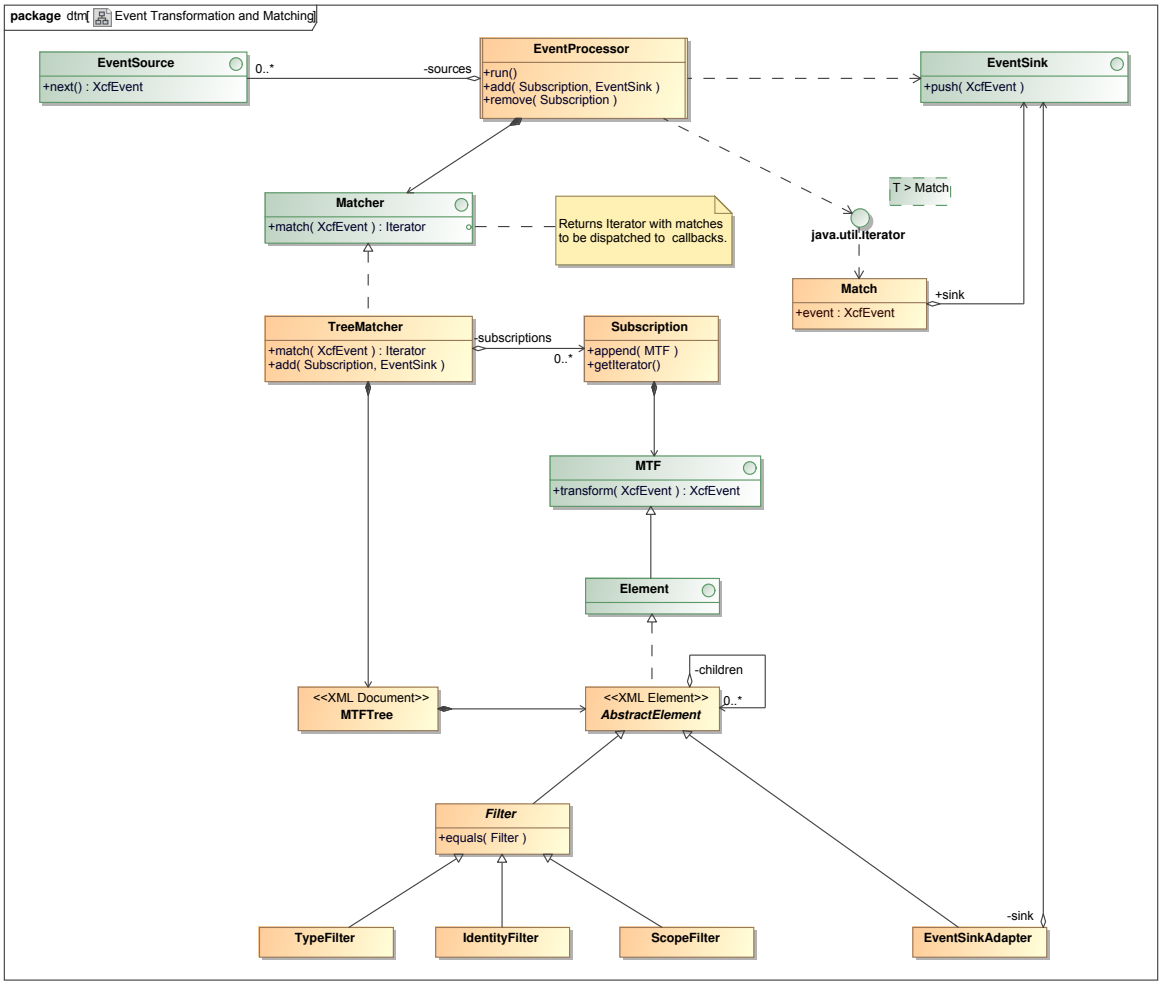


Figure 6.8.: *Class diagram for the tree-based event matching and transformation model.*

6.5. Notification Model

The fundamental paradigm for communication between participants in an IDI architecture is event-based interaction. Following up on the event and document models as well as the primarily content-based features for specifying event subscriptions, the critical piece missing to provide the basic functionality of an EDA are methods for distributing event notifications from informer to listener participants.

Thus, the aim of this section is to describe the abstractions necessary for communicating event notifications and to explain how events shall generally be dispatched to application code.

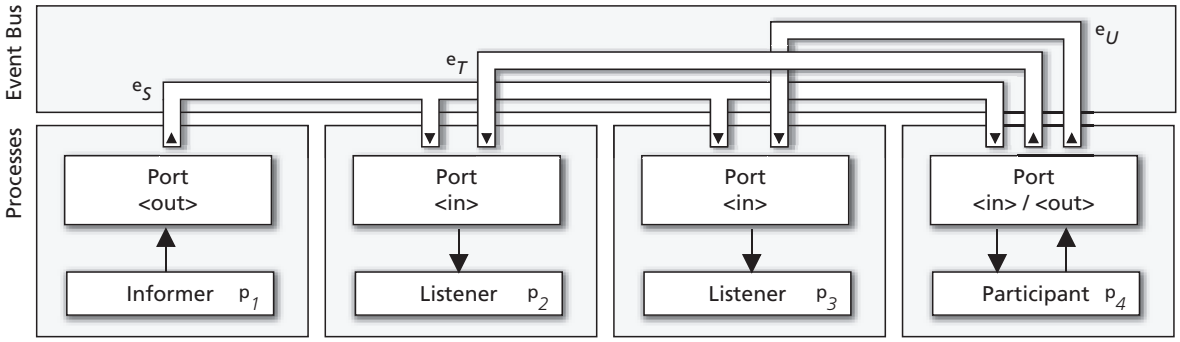


Figure 6.9.: *Participants connect via ports to a global, logically unified event bus that guarantees the delivery of event notifications to observing listeners. By adopting a* scope *concept, this bus is internally* channelized*, which is mapped to the capabilities of the underlying transport by port implementations for reasons of efficiency.*

In order to get started, let us recapitulate the requirements every event-based system architecture must comply with, concerning its delivery policy for notifications. Satisfying *liveliness* and *safety* conditions [MFP06], a listener participant shall receive:

- *Only event notifications it is subscribed to at a given point in time.*
- *Only notifications that were previously actually published by an informer.*
- *Each single event notification at most once.*
- *All further notifications that match one of its active subscriptions.*

It is in the responsibility of the observation and the notification models that the fulfillment of these requirements is guaranteed for each participant in an event-based system.

While the observation model already addresses the requirements that are necessary on the listening end of an interaction, it leaves open how these notifications are routed to their destination. In accordance with the event-based model of interaction, the style of communication between participants in the IDI architecture follows a *push*-based paradigm, which implies that informers in a system architecture initiate event transmission. Section 7.2 describes a number of communication patterns that allow to overcome this on the next higher level of abstraction if needed. In general, the fact that informers are not aware of their associated listeners contributes to the desired loose-coupling of participants.

6.5.1. Implicit Invocation

The push-based architecture and the decision to collocate the event observation model functions with each listener instance allows for realizing an *implicit invocation* architecture. Informers and listeners are connected via so-called *ports* to an event bus as shown in Figure 6.9.

The sole responsibility of this bus on a conceptual level is to distribute all event notifications to all connected ports. Models for propagating event notifications in event-based systems range from direct communication between participants, over centralized to distributed and broadcast architectures or mixtures thereof [MC05]. In order to decide on a particular method for event propagation, scalability

concerns must be considered, because they have a large impact on the design of the notification routing facility in an event-based system. If internet-level scalability is desired, this prevents the use of centralized (single broker) or broadcast (flooding) event propagation models.

However, as explained during the discussion of the technology perspective in Section 4.2, the level of geographic scalability needed for the integration task here does not exceed over the boundaries of a set of interconnected local area networks. While this prevents the application of broadcast models, the use of *IP multicast* groups [Bir05] is a favorable alternative for distributing event notifications over IP-based networks. In contrast to architectures that make use of intermediate components either distributed as it is done in Jedi [CNF01] or centralized as in many other approaches [MFP06], this multicast model has the advantage that subscriptions become immediately effective at the local observation models and that latency from event generation to notification retrieval is minimized.

The actual interaction between participants is carried out based on the port concept either over a networking layer or some other kind of serialized communication media as indicated in Figure 6.9. By utilizing *in-band signaling* [Fai06], control flow is not separated from the event notifications and thus no additional control channels are necessary. Ports connect the observation to the communication functions of the notification model and logically offer a bus interface to the higher-level components of the IDI architecture. For listeners, ports act as local event sources whereas they serve as event sinks for associated informer components as shown in Figure 6.10. Much of the functionality that deals with the networking and low-level infrastructure issues like efficient marshaling of event content or metadata is encapsulated in middleware or transport-specific port implementations.

In the current implementation of the IDI software architecture, the multicast-based event bus is realized on top of a group communication framework as explained in Excerpt 6.5 which allows for efficient *and* reliable process communication based on unicast *or* IP multicast. Ports serve as an abstraction layer decoupling higher level code of the integration architecture and thus also application code from the concrete low-level technology that is used for event transmission.

The router component of each individual listener participant thereby acts as a registrar connecting new subscriptions to its inbound port and as a local event dispatcher which forwards event notifications received from an associated port to the matching algorithms of its observation model. The locality of the notification model with regard to individual participants and the missing direct connection between informer and listener actually effects the claim for an implicit invocation architecture. A port in the IDI architecture shall thus be defined as follows:

Definition 6.2 (Port) *An IDI port is a bi-directional communication endpoint abstracting from a concrete transport infrastructure. Its responsibility is to realize a software bus for the exchange of event notifications. If possible, it maps concepts of the event, resource and observation models on its specific technology, thereby allowing for an orthogonal optimization of event processing.*

This definition highlights an additional benefit gained by the introduction of ports, which is that a port implementation may optimize event processing along different dimensions, e.g., in terms of notification marshaling.

Excerpt 6.5 contains a brief explanation how this is performed in the current software architecture. Another examples is the evaluation of event notifications before they are written on the network layer.

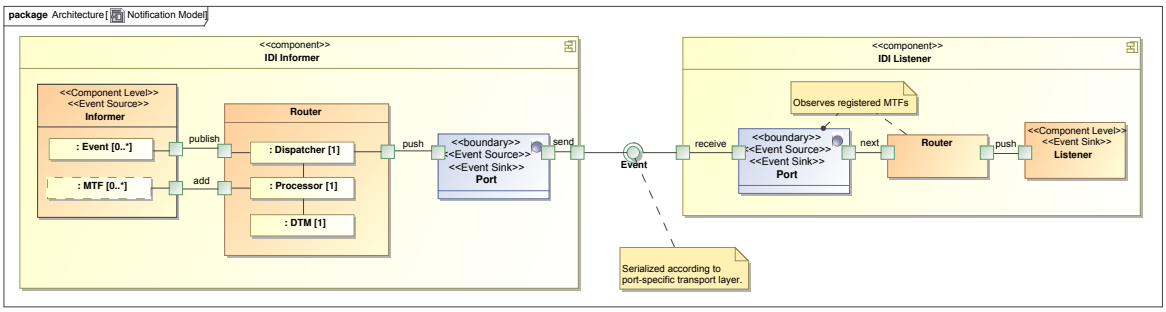


Figure 6.10.: *Architecture of the basic event-based communication model.*

Thereby, ports may send messages directly to specific participants, which is made possible by analyzing the sender / receiver metadata. Another example is to enforce specific delivery constraints as encoded in the metadata of an event notification with constructs of a specific underlying technology infrastructure.

Even more important with regard to port behavior than marshaling and delivery constraints is that ports can act as observers of MTFs that are registered for incoming and outgoing events in its associated router component. This allows for an optimized mapping of the software bus concept to the properties of the underlying technical infrastructure. If a subscription registers a new MTF that is known by a port implementation, it is registered with the port and dynamically reconfigures the behavior of the corresponding port instance, e.g., in terms of what events are actually received from the network. While the filters in the observation model are designed to work solely on the event content and its metadata, port implementations can dramatically increase the performance of filters if their semantics are natively supported.

If an event is received by a port that already satisfies a filter's matching rule, the router skips the subsequent content-based matching for this filter instance *and* event notification. The marking of individual events as being already filtered is termed *whitelisting* in the IDI architecture. Whitelisting is a generic functionality available for all IDI filters. Port-based optimizations must be dynamic as filters are usually dynamically (de-)registered at runtime. If a specific port type does not support an optimization, content-based matching is applied as a fallback. Obviously, port implementations must comply with the general requirements for the notification model as described in the beginning and must not change the semantics of a transformation function.

6.5.2. Visibility and Scopes

Operational middleware (see Chapter 5) does not concern visibility in the first place due to the fact that participants are explicitly addressed. Within an event-driven architecture that connects participants through a unified notification bus without explicit subscription routing, things are different as event notifications are logically accessible by every participant.

While the effected traffic overhead on a network link may be negligible if IP multicast is utilized as done in the proposed IDI architecture, the overhead in event processing can not simply be avoided. Thus, a concept of event-driven architectures for limiting the visibility of event notifications is adopted that decreases the overhead of the broadbast-style event notification model without breaking its semantics by utilizing the introduced port-based optimizations.

Excerpt 6.5: A Event-Bus Based on a Group Communication System

Notification routing in the IDI software architecture over standard IP network layers is based on the Spread group communication toolkit [AS98]. Spread is a high-performance (it allows for communication of over 8,000 1Kbytes messages a second in local area networks), fault-tolerant messaging service that provides a unified message bus for distributed applications based on network-level multicast and group communication support [Bir05]. The Spread toolkit is publicly available and is being used by several organizations in both research and production settings. The system consists of a per-host daemon architecture and a client library that is linked to applications. It supports cross-platform applications and has been ported to several Unix platforms as well as to Windows and Java environments. It features different language bindings, in particular for C and Java, which was essential for using it as a basis for the realization of the IDI software architecture.
Spread offers messaging guarantees ranging from reliable message passing to fully ordered messages with additional delivery guarantees. It supports multicast and unicast connections across the boundaries of local area networks. While in LANs (IP) multicast groups are used, unicast connections are established between Spread daemons for routing between LANs. Thereby it effects a distributed [Fai06] routing of event notifications up to the level of the IDI port structure while keeping up the unified messaging bus semantics. It features a simple but rather low-level API, that does not offer much functionality above pure group-based messaging. Thus, it provides almost orthogonal features compared to the high-level functions introduced by the IDI architecture yielding a good match for the reference implementation of a multicast-based IDI port.

Group Communication The central abstraction concept in Spread is a *group*, which is a logical representation of a set of processes that communicate via multicast in an asynchronous environment where failures can occur. Besides ordered message delivery, spread basically features a group membership service [Bir05]. This service provides all members of a group with information about the list of currently connected and alive group members and notifies group members about every group change either when members voluntarily join or leave the group or faults occur, e.g., if a process crashes. Spread offers a many-to-many communication paradigm where any group member can be both a sender and a receiver. Messages can be send by processes to groups even if the sender is not a member of the destination group.

Event Notification Marshaling Marshaling is a process that transfers data structures from one address space into another. As Spread ultimately is about networking over of a serialized connection, the marshaling process also serializes the data structure, while the unmarshaler deserializes them on the other end [Fai06]. However, Spread comes only with very limited support for marhaling. Actually, it solely accepts structures that are already broken down into a sequence of bytes.
Thus, the IDI software architecture comes with a simple marhaling algorithm, which translates event notifications into a vector of unsigned characters which can subsequently be processed by spread. Each serialized event notification message starts with of a header block that contains the event ID, the event type, its timestamp as well as sender and receiver URIs. This allows for the port-based optimization of certain filtering operations, e.g., identity filtering. The XML document is marshalled subsequently to this metadata block, followed by all binary attachments associated with this event. All binary data types are encoded utilizing a simple scheme that is inspired by the *basic encoding rules* as known from the ASN/1 standard [Dub08].

Scopes are abstract means of restricting the visibility of notifications to the participants in event-based system. While observation and notification principles are applied in the same way as described earlier, the interaction between the members of a scope with participants outside of the scope can be limited or completely prohibited [MFP06].

Scoping is complementary and connected upstream of the subscription-based filtering process[3]. If a notification is not visible to a participant, which means in the IDI architecture that it does not match any of its scope filters, it needs not to be further processed by the local observation model. Despite this obviously positive impact on the processing load of participants, the introduction of scopes promotes good software engineering principles like information hiding, abstraction and the specification of component interfaces in an event-based architecture. While subscriptions govern the local actions that are executed in a participant if its conditions match, scopes govern the system-level interactions between participants. As these interactions are defined externally in the participants configuration by developers or system architects, no source-level compile time dependencies are introduced. Scopes in the IDI architecture can thus be defined as follows:

Definition 6.3 (Scope) *A scope is an abstract concept that limits the visibility of event notifications in the global event space. Scopes logically bundle a set of participants and allow for optimizing the routing of event messages in the notification model.*

This definition highlights an additional benefit of scoping: scopes may serve as a structuring principle for grouping of physically or logically coupled participants. This provides an avenue for advanced optimization steps, e.g., the use of more efficient transport protocols within a single scope. Other advantages of introducing scopes will be explained during the discussion of the resource models and in Section 7.1.

Scoped Notification Dissemination

As each scope itself can recursively be a member of higher-level scopes, a hierarchical graph structure, the so-called *scope graph* can be constructed. A abstract example of a simple tree-like scope graph is shown in Figure 6.11.

The delivery policy for scoped notification dissemination in the IDI architecture is that event notifications sent by informers are delivered to the specified scope *and* to all children of this scope. This is exemplified in Figure 6.11. A notification that is published from an informer p_4 in scope $G.C.A$ to listeners in scope $E.B.A$ is visible in the target scopes and for participants belonging to a scope below, hence p_2 in scope $H.E.B.A$ and p_3 in scope $I.E.B.A$. However, the notification must not be visible anywhere else, particularly not for participant p_1 in the $D.B.A$ scope.

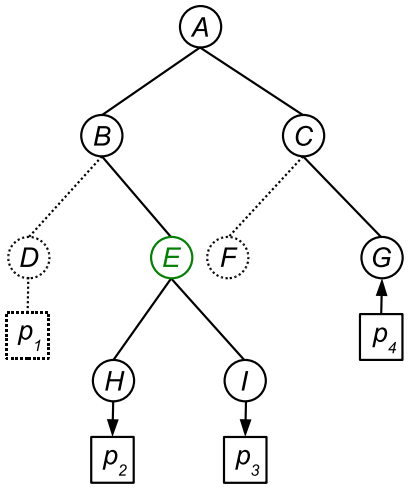


Figure 6.11.: *Scopes restrict the visibility of notifications.*

[3]While not visible in the external subscription interface, port implementations guarantee that scope filtering is conducted in preface of any notification matching.

The semantics of scope membership shall denote a *part-of* relationship between the participants in a scope. For instance, p_2 and p_4 are parts of the scopes $H.E.B.A$ as well as $I.E.B.A$ and thus also of $E.B.A$. While scopes are defined in the IDI architecture up to now only from a receiving perspective, scope control concepts can additionally be utilized for, e.g., defining intra-scope delivery or inter-scope transformation policies [MFP06]. However, as the benefits of doing so were less clear compared to the added complexity, the use of scopes in this architecture is primarily to elicit performance benefits by reducing the set of unfitting notifications that are to be evaluated by the local observation model of each participant and for structuring purposes.

Scope membership is specified individually for each participant and is evocative of a direct addressing scheme. Even so, the scope information is in this approach not encoded in the components but is configured externally. Thus, it delivers the benefits of visibility restriction and grouping on the basis of a much more loosely coupled communication model without any additional compile time references. Figure 7.3 that will be explained during the introduction to the resource model in Section 7.1 depicts an example of a scope tree for the components of the VAMPIRE assistance system, which is described in greater detail in Chapter 8. Based on this logical scope model, visibility control can be enforced in a system architecture.

Scope Architecture

While the previous paragraphs described the aims and semantics behind scopes, the following will explain how these concepts are mapped to the implicit invocation architecture of the IDI approach. Due to the lack of intermediate event brokering components and for performance reasons, the chosen strategy is based on an *implicit* but instantiated scope implementation in conjunction with a *collapsed filters* [MFP06] approach.

An implicit scope implementation shifts the responsibility of scope management and thus of visibility control into the individual participants that are connected to the event bus. While such an approach is infeasible in domains that require strong security policies[4], it is well suited for an architecture that aims at availability of information for all participating system components. Due to the fact that no explicit administrative scope components are used, these approaches often lack flexibility if the scope assignment for participants change. Following up on this, the IDI software architecture features an explicit instantiated scope implementation that maintains the scope attributes for all participants and allows for changing these at runtime if needed.

In general, the IDI scope architecture requires a bi-lateral cooperation between informer and listener participants with regard to their way of sending and receiving event notifications from the event bus.

Regarding receipt of event notification, a collapsed filter approach is followed where visibility constraints are enforced by merging these as an upstream filter, the so-called `ScopeFilter`, in the subscriptions of the listening participants. This leads to a “flat” notification model where enhanced filters implicitly enforce the visibility constraints. For the scenario in Figure 6.11 this means that the notification send by p_4 is physically transmitted directly via the event bus to the listening participants that are members of the $E.B.A$ scope. Scope filtering in the IDI architecture is based on the assumption that each fully qualified scope name, which is the concatenated list of scopes from bottom to top, e.g. $D.B.A$ in Figure 6.11, is unique in a single system.

[4] Participants not adhering to these conventions could easily compromise visibility constraints.

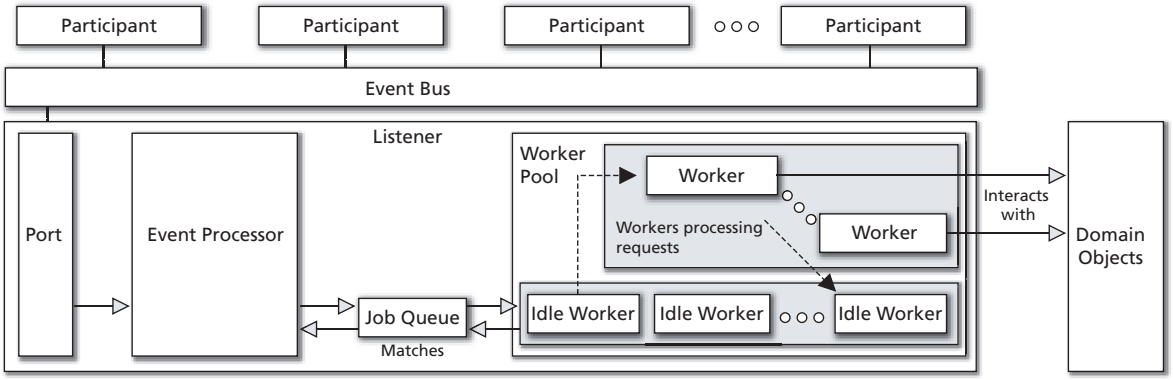


Figure 6.12.: *Multi-threaded event dispatch is realized in a variant of the job queue pattern [Pet05].*

Secondly, informers must annotate event notifications with additional scope information. As participants must be members of a scope, the receiver information is set to the root scope if not otherwise specified, which would make a notification visible for all participants in the system. This scope information is encoded as a fully qualified scope name in the receiver metadata in the form of an URI scheme that will be introduced in Section 7.1. Thus, informers control the visibility of information in the event-based system architecture.

If expressive content-based matching functions are available as provided by the previously introduced observation model, scope filtering is possible solely by evaluating the metadata of received event notifications. Scope filters are always evaluated before all other filters that are registered in a subscription. However, while this would already prevent the complete evaluation of subsequent filters in a subscription, still all received events would be fed into the local observation model of participants.

Fortunately, the abstraction of low-level communication through the port concept allows for shifting the scope evaluation functionality into the network layer. As explained previously, ports may observe the registration of filters. Thus, the addition of a scope filter to an observation model allows it to reconfigure itself and fetch only those notifications from the communication layer that match its visibility constraints. By whitelisting the received event identity in the scope filter instance, its content-based matching process is circumvented, which shall yield the expected performance gain.

A consequence of scoping is that the broadcast style event bus is logically and physically partitioned into many-to-many communication channels that emerge from sending a message from an informer to a number of listeners in a specific scope. If a port implementation supports a mapping to an underlying middleware or communication technology that natively facilitates these concepts, ports act as *channelizer* [Fai06] for the event notification. Excerpt 6.6 describes how the presented scope concept is mapped to the Spread-based port implementations which are used for network distribution at the time of writing this thesis.

The idea of using scopes is essential for large parts of the system management functionality that is available in the integration architecture as scopes directly support group addressing and anonymous requests based on the naming concept that is introduced in Section 6.5.

Excerpt 6.6: Mapping Scopes to a Group Communication System

Since all communication in the current realization of the IDI architecture is carried out on top of the Spread toolkit as described in Excerpt 6.5, a technology mapping for the scope concepts was developed that maps scopes to spread groups. By doing so, the Spread Toolkit already enforces in conjunction with the spread-based port implementation the visibility constraints in its low-level messaging subsystem. Only matching messages are delivered by a spread-daemon to the requesting process.

Dynamic Mapping To achieve this mapping, a scope name is translated to a spread group. The fully qualified name of the scope is used as a unique group name. Each message is sent only to the group of the receiving scope. Upon registration of a scope filter, the port of a listener participant joins its own group as encoded in the scope filter instance as well as the groups of all parent scopes of this specific scope. This way participants in sub-scopes will also be able to see messages that are send to its super-scopes as it is intended.
Let us consider again Figure 6.11 as an example. According to the shown configuration, the port of participant p_1 is configured through the registration of filters as a member of the $D.B.A$ scope. In addition to joining the corresponding spread groups for $D.B.A$, the port has joined groups for the scopes $B.A$ and A. Thus, it receives messages send to its own scopes as well as all of its super-scopes. This easily allows to address composite parts of a larger system by sending a message to a super-scope. The message send from p_4 that is targeted at $E.B.A$ is not visible to this port as it is not a member of the corresponding spread groups. Thus, the visibility constraints are already enforced through spread and need not be evaluated through the content-based matching in the observation model.

Whitelisting Even so, whitelisting of single events is necessary to conform to the safety condition of an event-based system as an individual port can be a member of many scope groups. Thus, the individual scope filter within a subscription must be notified about the fact that the received message has already been successfully evaluated in terms of its visibility constraints.

Additionally, using scopes for grouping and structuring a system, allows for re-configuration and re-use of individual participants or whole sub-scopes in differing application scenarios as the actual scope configuration can be configured externally.

Instead of hiding interactions between cooperating components in the source code, their collaboration is made explicit. Thus, scopes are not only a technical optimization within the observation model but additionally positively impact on the non-functional attributes of the IDI architecture.

6.5.3. Dynamic Dispatch of Event Notifications

While scoping is an abstract concept that has technological implications, the final missing link for the notification model to become effective is the question how local callback handlers of participants are invoked. While this is closely related to the observation model, it shall be briefly discussed now as the aforementioned functions of the notification model are required for it to perform its tasks.

The dispatching model in the IDI architecture is informed by the job queue and worker pool pattern [Pet05] as depicted in Figure 6.12. It realizes the multi-threaded dynamic dispatching of matching event notifications to locally callback handlers that are registered in the observation model.

If an event notification is received from a port it is matched in the observation model against its subscriptions by the event processor, cf. Figure 6.10. The matching pairs of event data and the callback handler that is attached to the corresponding subscriptions are inserted into the job queue, which is part of the dispatcher component. In turn, the next idle worker thread processes this pair and actually invokes the local callback handler with the given event.

The worker pool pattern allows to separate event matching from callback invocation. Thus, long-running event handlers do not stall the event matching process for pending events. The worker pool is externally configurable as the optimal parameterization of thread pool sizes can be application specific. In contrast to the standard job-queue and leader-follower pattern, an individual worker has no possibility to return job's to the queue and does not hold any network connections or locks for low-level devices. Instead, this is handled by the individual higher-level patterns as will be explained in Section 7.2. In the current realization of the IDI software architecture, the dispatcher component is based on generic synchronized queues and the dynamic worker thread pools.

6.5.4. Port-based Optimization

Adding up on scoping and port-based optimization, another important aspect of the notification model in the IDI architecture is that similar to the registration of message transformation functions through subscriptions in the inbound event processor of the router component in the observation model, an outward set of (optional) transformation functions can be registered that allows for filtering of outgoing events before they are broadcasted by associated ports. In fact, it is the same concept except that now informers are the event source and a transformation is performed to ports that act as event sinks.

Recalling Figure 6.10 the basic concepts of the notification model and its connection with the observation model can be summarized as follows: Events are published by an informer to its local router component that processes it in its outward event processor instance, which transforms or filters a published event still within the informer component. If the event notification is dispatched by the matching algorithms to a specific port, it is marshalled and transmitted according to the port-specific transport strategies that are optimized according to the current state of the outward set of MTFs in its event processor component and its event metadata information. From the perspective of the listener, the process is reversed. Ports dynamically enter communication with other ports according to the semantics of the underlying technology, ideally already filtering for scope or other selective information according to the current set of subscriptions registered in the inbound event processor of its associated router. If a subscription matches the notification as described in the previous section, the corresponding event handling methods of associated callback objects are invoked dynamically in a separate thread of control.

Many of the aforementioned mechanisms, e.g. scoping or native metadata serialization in conjunction with whitelisting explain the notion of *orthogonal optimization* that was mentioned in the port definition. While the layering in the overall architecture is maintained, ports introduce low-level optimizations that have effects at higher architectural levels without breaking encapsulation. By transparently introducing these optimizations, the overhead of the general bus architecture can be alleviated while keeping its benefits. Compatibility across different port implementations is ensured as all concepts can be mapped to content-based methods if necessary.

6.6. Summary

This chapter introduced the core architecture of the IDI approach, which is largely adopting models of event-based systems, that are typical choices for scalable and modifiable architectures [SG96]. Modules in an EDA are self-focused, thereby contributing massively to the strategic aims stated in the beginning of this chapter. Loose coupling is facilitated through a consistent application of document-orientation on the natural basis of the XML data model, which at the same time allows for the development of generic message transforming functions such as the introduced compacting filter.

However, extensions for efficient handling of binary large objects have been added to the introduced event model. The event types and the introduced expressive features of the observation model in conjunction with the optimizations introduced by the notification model such as the channelized event-bus yield pre-selection mechanisms that reduce the processing burden on components and allow to build scalable and efficient cognitive systems architectures.

While the core features of the IDI architecture described in this chapter are rather generic, the next chapter introduces domain-specific IDI models particularly facilitating the design and construction of experimental cognitive systems.

7. From Event-based to Cognitive Systems

A common prejudice about asynchronous event-based system architectures is that they are designed from a purely architectural viewpoint, which may lead to minimal coupling between participants but at the same time complicates their development due to the lack of pre-defined higher-level structures [Hoh07]. While the features of the observation model in conjunction with the document-oriented event model already allow for expressive logical connections between interacting services, this chapter describes explicit domain-specific functionality which shall aid researchers in constructing cognitive systems.

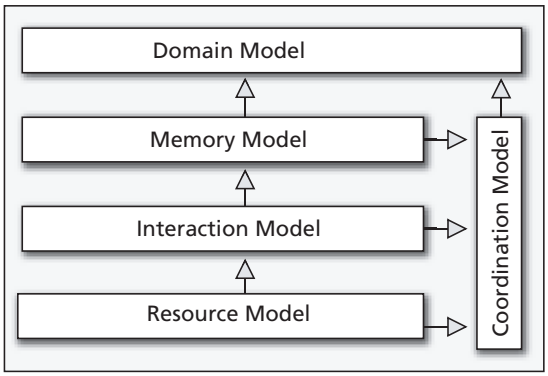


Figure 7.1.: *Domain-specific integration models.*

A number of higher-level models are introduced in the following sections, added on the IDI architecture's fundamental event-based integration layer, that have proven substantial for the domain. Some of the models, e.g., the resource model break to a certain extent with strict event-based principles, but they do this for good reason. They are focused at the fulfillment of real-world requirements, which in this case was to be able to structure a system into larger reusable services that represent coarse-grained named building blocks of functionality, permitting to assign functional responsibilities to clearly separated components in a cognitive system architecture.

In addition to functional concerns, modularization facilitates collaborative work through independent but parallel development, thus additionally assigning organizational responsibilities to individual collaborators for clearly defined parts of a software architecture.

These additional models provide still generic but at the same time already more specific functionality. The structural dependencies between these models and the corresponding organization of this chapter is shown in Figure 7.1, starting with the *resource* and *interaction* models, which introduce higher-level abstractions and extensions to the event-driven models for developing experimental cognitive systems. The *memory*, *coordination* and *domain* models explained subsequently, represent central functions in the IDI approach that were used as major building blocks in cognitive systems research projects.

7.1. Resource Model

The resource model describes concepts and terminology for structuring and referencing the participants in an event-driven architecture. However, in strictly event-based systems, direct referencing of resources is considered harmful as a source of coupling and participants are solely defined in terms of generated and observed events. Even so, the need for modularity and composition of resources needs

to be addressed in an event-based system. In the IDI architecture, functions for resource structuring are primarily defined from an abstract perspective that focusses on modularization of domain functions. In order to prevent source-code coupling based on specific identifiers, the overall approach is highly configurable. The resource model as presented in the following fuses ideas from two current methods in software integration of distributed systems: the *Service Component Architecture* [OAS07] (SCA) and the previously introduced scope concept. Both aspects are merged into a URI scheme that allows participants to refer to each other individually, to abstract services or to groups of services contained in a scope.

An additional benefit of re-introducing identity information in an event-based system is that this further increases the expressiveness of the observation model, which is particularly useful for, e.g., event correlation and other situations where the identity of a participant is critical for interaction. Furthermore, the need for the actual naming, management and monitoring of individual participants within an integrated cognitive system suggests the re-introduction of resource identifiers. Besides these motives, another important aspect originating from usability considerations is that identity information helps system developers to communicate about a system architecture and assign responsibilities for specific services or components. Therefore, a consistent naming terminology and resource model is crucial.

Necessary for the realization of a resource model are definitions for the structuring elements to be used, a coherent syntax and semantics for resource naming that respect the aforementioned aims and the introduction of a federated naming service. The following paragraphs introduce definitions that are used subsequently to refer to certain types of resources in the IDI architecture.

7.1.1. Services, Interfaces, and Components

While the lower-level layers in the IDI architecture are completely based on the adopted event-driven foundations, the provided higher-level functions are informed by high-level concepts of the *Service-Oriented Architecture* (SOA) paradigms. In particular, the logical modularization shall be guided by service-oriented concepts. Hence, the notion of a *service* in the information-driven integration approach is inspired by the corresponding OASIS service definition [OAS06]:

Definition 7.1 (Service) *A service represents a mechanism to effect access to one or more high-level, usually coarse-grained functionalities on the level of the functional architecture of a cognitive system. Access is provided using a prescribed information-driven interface and is exercised with the constraints and policies specified in a service description.*

The eventual consumers of a service may not be known to the service provider and may use the service beyond to what was originally conceived by the provider. As the IDI architecture was developed with loose-coupling in mind, so shall services consider this as an important prerequisite for getting reused and combined in new experimental system contexts.

The service definition is purely conceptual and must not depend on any implementation details. Services do not necessarily map one-to-one any component implementation. However, this definition promotes the use of interfaces for describing access to service functionality. While in the general SOA concept, the actual technology this access is based on is intentionally omitted, the IDI architecture defines it in terms of its event-based foundation:

Definition 7.2 (Interface) *The interface of a service is a named set of event types that is provided or requested by a participant in addition to a specification of its dynamic interaction behavior independent of any actual implementation.*

In order to describe the dynamics of interactions in an interface specification, the IDI architecture introduces a set of generic *interaction patterns*, such as request-reply, which will be explained in the Section 7.2. They actually determine - and realize - the expected dynamics in accessing the underlying capabilities of a service that is exposed using a pattern-based interface. As this set of interaction patterns is easily extensible, new interaction scenarios can be defined to be used in interface descriptions. However, the introduction of new patterns shall be limited as these interaction strategies must be available for the service consumer.

Interfaces represent the boundaries between functional and integration architecture, concept and implementation. While services are purely abstract bundles of functionality, their actual realization and exposition on a system level is realized with interfaces that ultimately map to pattern-specific instances of the event-based participants introduced in the previous chapter. While consumers of a service need to fully comply with individual interfaces, services may offer their functionality on different interfaces that may be used independently.

Nevertheless, interfaces are defined separately as they can be realized by different implementation artifacts that contribute to service functionality. To clarify this, the following definition of a *component* shall be used in the IDI architecture:

Definition 7.3 (Component) *A component represents a descriptor of a software artifact that realizes any number of interfaces, which can actually be executed in a defined system context. Execution of a component yields instantiation of participants that implement the specified interfaces, thereby contributing to or fully realizing services.*

Components are thus logical capsules for executables featuring an independent configuration. This allows to free their underlying source code from application specific properties and static dependencies, which again promotes to the goal of loose coupling. Their scenario-specific deployment may vary through different system instances.

A coherent understanding of service, interface and component concepts facilitate the collaboration in experimental cognitive systems research and promotes modularization. Furthermore, these up to now rather abstract definitions are the basis for the actual configuration management functions of the IDI architecture, which allow to orchestrate and demonstrate system instances.

Let us consider a person anchoring service as an example to demonstrate these concepts. Person anchoring is an important basis for stable human-robot-interaction as is realized in the robot companion developed in the COGNIRON EU project, which is described in Chapter 9. Such a service may be realized internally with many different strategies and may make use of different types of event information in a system if available. Even so, the interface of this service is quite abstract as it states that it solely publishes person hypotheses in the system and observes useful lower-level event perceptions, which may or may not be available. Such a high-level service, which makes minimal assumptions about its environment is easily reusable in a different application context with a similar requirement. In contrast, the component specification binds realizations of this service to a functional architecture and allows for their actual execution in a running prototype system.

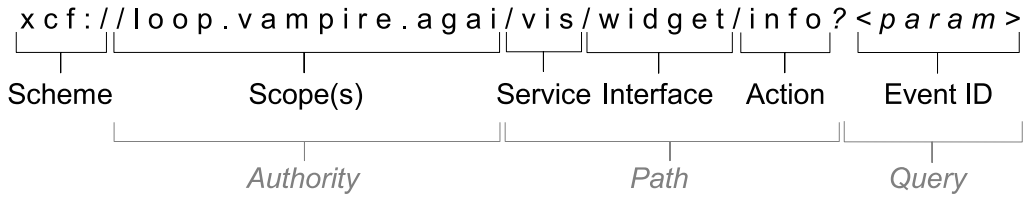


Figure 7.2.: *Exemplary resource reference as used in the IDI architecture and description of its constituting elements as specified in the introduced URI scheme.*

7.1.2. Naming Resources

A naming scheme for the envisioned integration architecture needs to support modularity, scalability, transparency and shall additionally promote understandability while not conflicting with the aim of loose coupling. In order to find a suitable compromise between these partly contradictory requirements, the resource model utilizes two concepts from different domains: the *Uniform Resource Identifier* (URI) and hierarchical scoping as introduced with the notification model in Section 6.5.

In contrast to other resource schemes abstracting solely from a specific physical location in a system architecture, e.g., a *Uniform Resource Locator* [BLFM05] (URL) referencing a web server available under a specific port on a given domain name address, the focus of the resource model in the IDI architecture is to permit a logical structuring of services according to their domain function. This is possible as most aspects of the required location transparency are inherently available using the broadcast communication style. Hence, the IDI functions introduce a higher abstraction level for resource modeling.

Compared to URLs, a *Uniform Resource Identifier* [BLFM05] (URI) as introduced by the *World Wide Web* (WWW) global information initiative in the 1990s is defined as "*a compact sequence of characters that identifies an abstract or physical resource*". This definition underlines the ability to address *abstract* resources and the featured declarative text representation, which is exactly why URIs are used to describe logical resources in the IDI architecture.

Figure 7.2 exemplifies the URI scheme developed for referencing resources in the IDI architecture following the recommendations for the general syntax of URIs. An IDI URI scheme starts with the name of the scheme (here `xcf`). Following up on the protocol specifier, the *authoritative* part of the URI describes in this URI scheme the logical location of the referenced resource through a hierarchical specification of its scope. As discussed in Section 6.5 scopes provide the basis for efficient communication in the event-based layer of the IDI architecture. The resource model takes up on the idea of scopes and utilizes these for the logical structuring of a system architecture on the functional level. While scopes are a very general concept, the URI scheme as shown in Figure 7.2 proposes at least three hierarchical scope levels, which were introduced from a practical standpoint in accordance to the integration context in cognitive systems research:

- *Organization*: Defines an institutional scope for participants.
- *Instance*: Defines a project or system scope for participating components.
- *Functions*: One or more scope representing logically or physically coupled functionality.

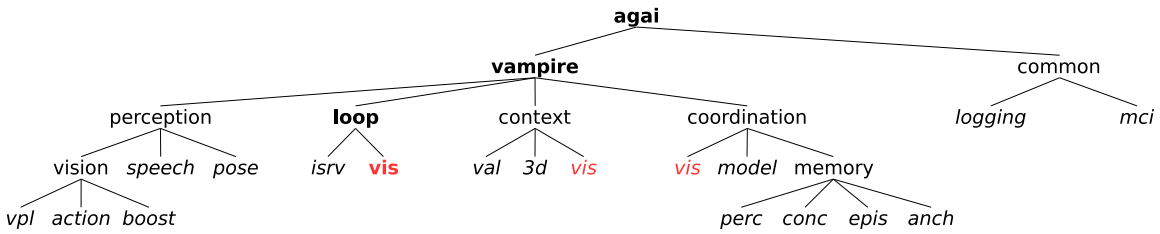


Figure 7.3.: *Utilizing the scope-based URIs, a tree can be constructed for an actual system architecture that quickly summarizes available services as shown here for the vampire system. Parent-nodes represent scopes while leafs are services. Although a* `vis` *service is used in three scopes, name clashes are avoided by the locality of identifiers.*

Even so, a service *must* be contained in at least one enclosing scope to allow for referencing with a valid URI. A service is encoded as the first element of an URI's *path* constituent by specifying its symbolic name, which is then local to the context defined by its enclosing scope. This path component of an IDI URI can be extended by a single additional name that specifies a single interface within a service.

In the depicted URI example, it is the `widget` interface that is responsible for distributing information about the augmentations displayed in the video see-through glasses to other system components. Further refinement of a URI is possible by appending optional action and query information to the interface or service identifier. While this can be useful to literally encode queries on participants, the introduced hierarchical scoping and the path element of the IDI URIs already allow for a transparent referencing of the scopes, services and service interfaces in a distributed system architecture.

Additionally, integrating the hierarchical scope in the URI concept facilitates the locality of service and interface identifiers. This not only prevents name clashes but also fosters re-use of application code if individual participants are designed to minimally rely on information about their enclosing scopes. This is enforced by the architecture as scope information is usually injected externally by a system designer and is not necessary for the development of component functionality. Summarizing, a URI in the IDI approach can be defined as follows:

Definition 7.4 (URI) *An IDI URI encodes an abstract reference to a logical entity such as a service or a service interface in a system. It uses hierarchical scopes for modularization in its authority part and thereby allows for referencing of functional sets of services or interfaces in a system architecture.*

To exemplify this, let us once again look at the URI depicted in Figure 7.2, which is taken from the software architecture of a VAMPIRE system. It references the `vis` service, which is located in the `loop` scope that contains the participants responsible for realizing the human-in-the-loop interaction exploiting the hardware and software of the AR-gear. The `loop` scope again is part of the `vampire` system and the `agai` organizational scopes. The consequent application of these concepts leads to clear and understandable service references, which can be visualized as a tree structure. Figure 7.3 shows an example of a resulting URI tree for the VAMPIRE assistance system that is described in greater detail in Chapter 8. The information contained in such a tree can in turn be used in system management tools that allow for configuring, deploying or introspecting a system at different levels of granularity, namely scopes, services or at an interface level.

7.1.3. The Federated Naming Service

While IDI URIs encode information *where* a component is logically placed in a system and how its services or interfaces can be referenced, an additional question that must be dealt with in the resource model is how URIs translate to physical resources and how unique names can be guaranteed. Both is usually realized in centralized or distributed naming services [Bir05], e.g., the Internet's *Domain Name System* [Net87] or Jini's trading service [Sun99].

The IDI architecture realizes naming in a federated approach. The realized naming service is itself largely based on the introduced event-based paradigms but exploits additional mechanisms.

The implementation of this service makes use of the advanced low-level features of the group communication framework that is currently used for the implementation of the notification model as introduced in Excerpt 6.5. For instance, guaranteed ordering and delivery as well as the group membership management functions are exploited for its efficient realization.

However, from a high-level perspective, the following properties of the nameservice realized in the IDI architecture are important for the realization of higher-level functionality that is introduced in subsequent sections:

- *Local uniqueness*: The name service guarantees that the chosen service identifiers are unique in their enclosing scopes.
- *Component mapping*: The name service resolves URIs to process identities that permit access to the actual component that implements a service interface.
- *System model*: The name service gives participants runtime access to a system model maintaining a list of all active participants and scopes of the overall system.
- *Failure detection*: Based on group membership, the name service can check whether a service is still available or has been disconnected for some reason, e.g., due to a crash.

The scoped URI model and the federated naming service as introduced here are inevitable cornerstones of the integration architecture to support developers in managing the complexities of setting up and running a complex distributed system. How these concepts contribute to the realization of higher level interaction patterns is in the focus of the following sections.

7.2. Interaction Model

In the same manner as the resource model proposes a hierarchical naming mechanism for logical structuring of service components, the *interaction model* specifies and enforces modes of interaction between participants. These modes and the accepted event types yield the specification of service interfaces. The aim of this model is to provide a set of broadly understood *interaction patterns* like synchronous *Request-Reply* communication between two system participants, which are realized on top of the asynchronous event-based core of the IDI architecture.

An interaction pattern (also commonly referred to as *communication* [Sch06a] or *message exchange pattern*) shall be defined here as follows:

Definition 7.5 (Interaction Pattern) *Interaction patterns provide reusable solutions for reoccurring types of communication that require the exchange of message sequences between software components in an asynchronous, event-based system architecture. They are defined on an abstract level and relieve developers from the error-prone details of interaction design in concurrent and distributed systems.*

These patterns come in particularly handy when realizing interactions more complex than publishing a single event, resembling to a *conversation* [Fai06]. Often, certain extra conditions shall be assured, e.g., the availability of an interaction partner, which needs supplementary coding in an asynchronous architecture. In addition to the aspect of reusing these generalizable functions and the higher abstraction level of interaction patterns compared to single event notifications, the available set of patterns defines a common vocabulary for software integration. It is by these patterns that developers may compose the overall architecture of a cognitive system out of services and components, which realize them by exposing pattern-based interfaces.

Recalling the integration context, interdisciplinary users often simply expect certain broadly understood interaction styles to be available in an integration architecture. For instance, it must be possible for them to interrogate a component for certain data by using synchronous request-reply - although extensive use of this pattern is not the central idea of event-based system integration. In addition to this, the available set of patterns shall in conjunction with the previously introduced models of the IDI architecture lead to modular, well understood semantics of event-based interactions between participants.

For these reasons, the IDI architecture itself provides an extensible set of interaction patterns that are utilizing all the concepts introduced so far to provide the needed abstractions, thereby hiding for instance threading, protocol, synchronization or lifecycle details of distributed and concurrent interaction. Excerpt 7.1 reports on some of these common design issues that are considered here for all pattern objects taking into account for instance their lifecycle.

7.2.1. Connectors and Service Interfaces

An interaction pattern usually consists of two complementary parts with distinct roles in an interaction that further describe their relationship, e.g. *Publisher* and *Subscriber* (with the exception of the event *channel* where there are no separate roles for the interaction partners). From the software architecture viewpoint, an interaction pattern provides an abstract *connector* between two or more components that is independent from a concrete transport infrastructure and that can be used to model and structure the software architecture of a complex system.

Mary Shaw defines a connector as the *loci of relations among components* [SG96]. Connectors shall decouple application from communication code as far as possible. The difference between the purely architectural viewpoint on connectors and interaction patterns is that the latter shall provide actual assistance for typical integration situations that may be domain-specific or introduce additional semantics in an interaction like the *active memory* pattern explained in the next section.

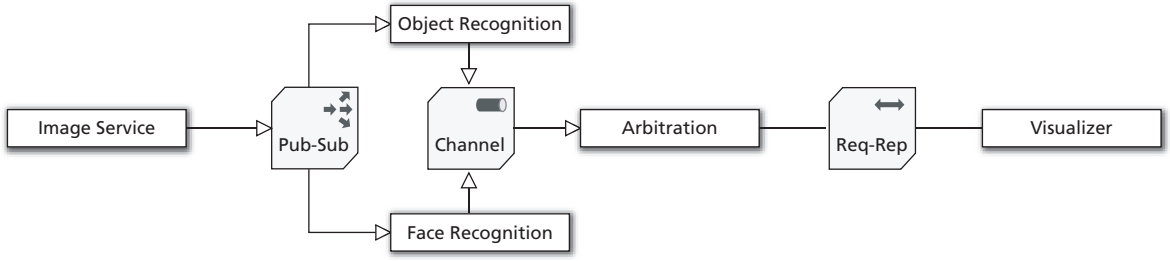


Figure 7.4.: *Architecture of a simple vision system exemplifying the use of the Channel, Request-Reply (Req-Rep) and Publish-Subscribe (Pub-Sub) interaction patterns.*

Instead of introducing new labels for already established concepts, the names for the patterns and supporting functions of the interaction model were chosen in accordance with well known terms from service-oriented and event-driven architectures. That said, the interaction patterns realized on the basis of the core models which are directly supported by the IDI architecture are:

- *Channel (N:M communication)*: Participants can act at the same time asynchronously as listeners and informers on the event bus that may be channelized. This is the basic communication pattern that is directly supported by the core IDI models.
- *Publish - Subscribe (1:N)*: Only publisher objects are allowed to send information on a virtually separated event bus they originally created. Subscribers act as listeners that are connected to the publisher's bus. This a restriction of the channel pattern.
- *Request - Reply (1:1)*: Classical client / server point-to-point remote-method invocation with at-most once semantics and support for asynchronous request objects. Excerpt 7.2 describes the event-based realization of this pattern in greater detail.
- *Anonymous Request - Reply (1:N)*: Compared to the request-reply pattern, a participant utilizing this pattern may send a request to an unknown server component. Any number of server particpants may reply to this optionally asynchronous request.

All of these assume that the interaction patterns are available at runtime, resembling to a kind of meeting-oriented communication [ASTMvS02]. While the first two patterns clearly origin from the domain of event-driven architecture, the latter are exemplary patterns that realize a command-and-control type of communication as attributed primarily to service-oriented architectures. The semantics of interaction that are usually associated with these patterns, can be easily described in analogy to natural language [Fai06]. While channel and publish-subscribe bear similarity to a declarative type of interaction, (anonymous) request-reply can be primarily interpreted as imperative or interrogative, which may give at the same time some initial advice when to apply which pattern.

Recalling the aim of loose coupling, the position taken up in this thesis is to use the EDA pattern to the extent possible, while the SOA patterns should be applied with caution and without making assumptions about the context of a request with regard to the called service, e.g. the state of the component or its specific environment.

Figure 7.4 depicts a simple vision application utilizing these patterns in an architecture diagram, restricting the model solely to the high level architectural aspects of a system in terms of components and connectors. The diagram exemplifies that image events are communicated via the Publish-Subscribe

```
1  try {
2      XcfManager xm = XcfManager.createXcfManager();
3      // scope is set externally
4      Server s = xm.createServer(new XcfUri("xcf://vis/widget"));
5      s.addListener(new RequestAdapter<FaceEvent>("create") {
6          @Override
7          public XcfEvent handleRequest(FaceEvent fe) {
8              // augment video and return status info
9              return vis.highlight(fe.getName(),fe.getRegion());;
10         }
11     });
12     s.run(false);
13 } catch (XcfException e) {
14     // ignored here for reasons of brevity
15 }
```

Listing 7.1: *Minimalist but complete example for server instantiation and callback registration using the request-reply pattern in Java. While request adapters can also be registered in Java with annotations, the example depicts regular callback registration as it is most similar to the programming model realized in C++.*

pattern from the image service to the pair of recognition processes, which in turn publish results and retrieve arbitration info via the general event channel pattern from a specific arbitration component. Finally, the visualization component is connected to this arbitration component via a Request-Reply pattern.

Please note that the connection between the arbitration and the visualization component is more closely coupled as it depends on mutual identity information. Besides their actual functionality, interaction patterns additionally contribute to the aim of improving communication between the collaborators in a research project due to their mapping to connectors in design-time, which allows to model their run-time system architectures.

Listing 7.1 gives an impression on how a (fully functional with regard to the IDI architecture) simple realization of a visualization service may look like from the developer's perspective. After the construction of a `Server` object that processes incoming requests (*line 4*) with a scope-independent URI, a callback is registered as a local listener at the router of this pattern object (*line 5–11*) that is automatically bound to a subscription matching this request and event type. Upon instantiation, this service would be available under the URI explained in the resource model that was shown in Figure 7.2 given that the scope is configured correctly to `loop.vampire.agai`.

Excerpt 7.1: State-Based Design of Interaction Patterns

All interaction patterns that are provided by the IDI architecture implement an object structure as shown in Figure 7.5 that combines a number of software design patterns and aggregates different framework objects that provide commonly used supporting functions. The essential parts responsible for the basic pattern functionality are the following:

- `ActiveObject`: This class is the common base of all pattern objects that feature their own thread of control. In general, these objects are said to be *active* as indicated in Figure 7.5 for the `Subscriber` class. Active objects must (de-)allocate used threads or other operating system resources cleanly upon activation or deactivation. To enforce this contract, it is specified in the general `XcfObject` interface that all active objects must implement. The `ActiveObject` class additionally provides access to the typical collaborators of a higher-level IDI object, which are:
  - `XcfManager`: The manager class utilizes a *Builder* pattern [GHJV95] and hides construction and implementation details of pattern objects from their usage. Client objects are thereby solely bound to the generic interfaces of specific pattern instances.
  - `XcfUri`: As patterns instantiate *resources* in the sense of the IDI architecture, this class encodes an URI according to the resource model introduced in Section 7.1.
  - `XcfConfigurator`: Configuration is an important requirement for the actual deployment and use of software artifacts in different contexts, thus a configuration strategy is needed as will be explained in the next chapter. This class provides access to the configuration object that is necessary to dynamically modify parameters of a participant.
  - `Finder`: This interface encapsulates access to the federated trading service and therefore to the current state of the resource model for a specific IDI system. It allows to register and retrieve meta-information about resources, e.g., about the state of a Publisher, which is needed for the correct realization of the different patterns.
- `Subscriber`: The interface of this class represents the actual functionality of a specific interaction pattern, here the subscriber role of the Publish/Subscribe pattern. All pattern objects are modelled according to the *State* pattern [GHJV95] where the main pattern class, here the `Subscriber` class, serves as the context object, which only implements bridge methods for the external interface and state-independent functionality. It delegates all other calls to the corresponding method of the currently instantiated state object, which is in this example one of `SubscriberStateActive`, `SubscriberStateCorrupted` or `SubscriberStatePassive`. The state pattern allows to cleanly encode a finite state machine in an object-oriented structure where the current state of the context objects changes dynamically at runtime.

Applying this pattern-based approach for the design of integration classes with their usually complex internal structure has important benefits for the software architecture in terms of clarity of realization, e.g. by encapsulating a finite set of capabilities that is state-dependent in an individual class and the ability to handle the whole lifecycle of active objects cleanly at runtime, contributing generally to the correctness of a software design. It provides a straightforward structural and functional basis for extension and introduction of additional patterns.

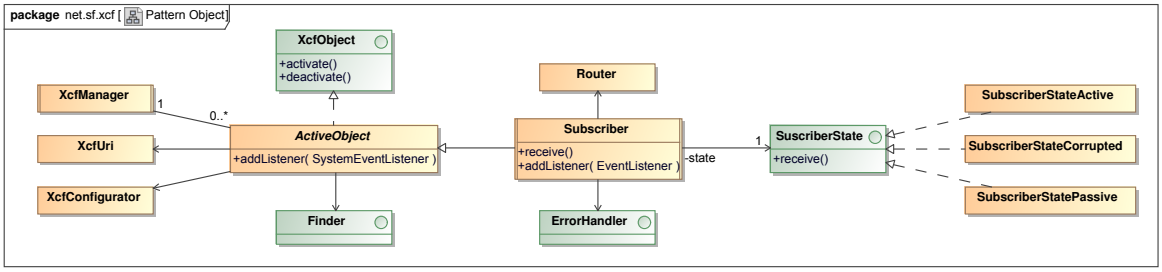


Figure 7.5.: *The general software design of interaction patterns in the IDI architecture exemplified using the classes responsible for the* `Subscriber` *realization. Besides collaborating with implementations for the IDI models, the pattern objects implement a State pattern [GHJV95].*

7.2.2. Event-based Realization

The described patterns realize their conversational strategies utilizing the event-based core of the IDI architecture. Thus, they use the features of the observation model, e.g. for dynamic registration of subscriptions in order to add guarantees to the communication between participants like the exchange of acknowledgements for received events or utilize subscriptions in conjunction with the functionality of the resource model to allow for patterns that re-introduce identity information.

Taking up on the previous examples, it is therefore possible for the IDI architecture to cancel the publishing of request events to a server and to throw an exception if it is not available at that time. This not only eases system development as failures are made explicit, but also simplifies their use from a developers perspective.

Not only event types and their serialization methods can be re-used by module developers across different patterns but at the same time it is for instance possible to combine filters with a method handler callback that is registered at a `Server` object in order to further constrain the set of events that is dispatched to this method handler. Similarly, it is possible to register the same event listener or a clone of a complex subscription at different interaction patterns to achieve code re-use and combine information from different interaction patterns in a central place. This underlines that the same concepts were consequently applied across the different parts of the integration architecture, which shall further increase usability.

If specific timing or quality-of-service constraints must be enforced, this is generally handled by adding corresponding MTFs to the inbound or outbound observation models of the patterns, cf. Chapter 6.4 and must not necessarily be part of the pattern implementation. In order to better assess the implementation aspects of the IDI patterns, Excerpt 7.2 describes the realization of the Request-Reply pattern in greater detail.

It is important to note that in contrast to middleware approaches that are based on static stubs and skeletons as outlined in Chapter 5, all patterns introduced here and their parts, e.g. the exposed methods of a `Server` interfaces, are per-se dynamically instantiated and registered at the architecture runtime without the involvement of a meta-compiler. This allows for the dynamicity necessary to adapt the orchestration of a system architecture as needed at runtime and reduces the complexity of the resulting toolchain for module developers thereby promoting to the overall usability.

Excerpt 7.2: An Event-based Request-Reply Pattern

Realization of the Request-Reply pattern using the event-based IDI core models requires to solve two tasks. On the one hand both participants need to directly address their partner, on the other hand delivery needs to be guaranteed, respectively errors detected to prevent the client side from blocking infinitely.

Identity Evaluation In order to allow for identity based message observation, an `IdentityFilter` is needed, which can be parameterized with either the sender's or the receiver's URI. In contrast to the concept of a scope filter, matching on identity is based on the whole URI and does not consider scoping mechanisms. The current implementation applies transport layer optimizations to the `IdentityFilter` by encoding the sender's and receiver's URIs into the binary message protocol, thus allowing for more efficient filtering without the need to deserialize or match message content.

Roles and Responsibilities The Request-Reply pattern defines two distinct roles realized in separate classes, the `Server` and the `RemoteServer`. The latter role represents the client side as it initiates the communication by sending a request and expects a reply from the server. As multiple requests may be sent in parallel, the client needs to match incoming replies to the corresponding requests. The event correlator pattern described in Section 7.2.3 is applied to achieve this. In Java, the `get` method of the `Future` interface, which is the basis of the asynchronous callback pattern is used to block the thread that sent the request until the reply becomes available or a timeout occurs.
On the server side, identity filtering is applied to incoming notifications as the server only answers requests directed to its method URIs. Incoming requests are dispatched to registered event listeners which implement the different methods. The generated replies are automatically augmented with the correlation id of their request, sender and receiver URI are transferred as well but swapped. Therefore the reply is directed to the scope of the client participant.
On the client side, identity filtering narrows notifications down to those sent by the server. Events are then handed to the event correlator which looks up the corresponding request and notifies the waiting requester thread of the reply.

7.2.3. Adaptation Patterns

Instead of the interaction patterns that deal mainly with message exchange protocols, the following list of patterns is generally applied locally. While not being first class interaction patterns, they represent auxiliary functions dealing with synchronization and adaptation of the programming model semantics. Due to the fact that they are very important from a component developers perspective and are even frequently used within the IDI architecture itself, they are additionally supported at this level of the integration architecture and explained in the following:

Active Queues Queues are versatile and generic communication adapters. They can be used to reverse API semantics with regard to handling of incoming event notifications. While usually a push model is realized by the IDI architecture, queues permit to process events according to the pull model. By attaching them to multiple informers using provided *queue adapters*, several subscriptions can be attached to one queue. As all queue implementations are thread-safe, they can be easily used as synchronization points in multi-threaded applications. Queues can also realize temporal event handling schemes. E.g., a queue type is provided that stores only the n most recent events, which allows to couple a listener to an informer with incompatible event production / consumption ratios.

Active queues additionally register request handlers in the IDI architecture, allowing to access them not only locally but also remotely via Request-Reply communication. Therefore, they can also be seen as a special type of connector from an architectural viewpoint and thus partially share the semantics of an interaction pattern.

Asynchronous Callbacks Synchronous calls to API functions have the obvious drawback that processing is suspended until the thread of control returns from the called function. However, a benefit of this synchronous model is that failures can be directly reported to the caller, usually in the form of an exception in object-oriented programming. In contrast, asynchronous invocation allows to directly continue processing instead of waiting for the result of a possibly long-lasting request to a service with the drawback that errors can not be easily fed back to the caller.

The IDI architecture provides an asynchronous callback pattern that can be used in conjunction with most of the functions of the pattern objects. Within the Java API, this class is based on the `Future` interface, thus allowing for synchronous as well as asynchronous retrieval of results. This object is also notified in case that a message has been discarded or another error occurred.

Thereby, the programming model allows developers to decide whether to make use of an asynchronous processing without losing the benefits of a synchronous operation.

Event Correlators Whenever bi-directional communication occurs, for example in the Request-Reply pattern, a mechanism is needed to find event notifications belonging to the same communicative act. Considering the Request-Reply pattern, a client may send a number of requests and expect a reply for each of them. A single communicative act consists of a request and the corresponding reply. But in event based systems notifications are independent of one another, therefore the client implementation is faced with the task to correlate the set of open requests to all incoming replies in order to match them up.

Hence, the IDI architecture offers an implementation of an *event correlation* pattern to handle this matching. The mechanism is based on adding correlation identifiers to any event which is meant to be answered by the communication partner. Each opponent answering the event must augment the reply with the identifier. This allows the sender of the initial event to filter for the correlation ids of open requests and thus find the corresponding reply.

As it implements the `EventSink` interface as introduced in Section 6.4, the event correlator can be linked into the observation model. This way it is possible to use the pattern with independent subscriptions and it is possible for other patterns to make use of this functionality. Access to correlated events is provided through the aforementioned asynchronous callback objects.

While the basic but fundamental interaction patterns help system designers to identify the core structure of applications in terms of integration, the adaptation patterns help developers to use the programming API in a way that suits their needs in a specific situation. The presented set of interaction patterns shall avoid mixing interaction and implementation issues and is easily extensible towards specific scenarios. An exemplary pattern that was adopted and added in the DESIRE [DES08] context is a *Task* pattern that permits monitoring of the completion state of an asynchronous request. The next section will introduce additional patterns at this abstraction level that realize a specific connector based on an active memory, already utilizing the patterns introduced in this section.

7.3. Memory Model

The aim of the visual active memory (VAM) as proposed in Section 2.2.2 is to provide an avenue for integrated systems to track the real-world context in terms of visually perceived episodes, events, and scenes. In previous chapters, concepts were introduced that allow for loosely-coupled but still efficient interaction between the functional components in a VAM system. However, even with these mechanisms in place, it is (apart from obvious computational capacity constraints) neither feasible nor meaningful to process all data immediately and then discard it. This brings us to the centerpiece of the information-driven integration approach and the architectural basis of the aforementioned visual active memory paradigm, the *memory model*.

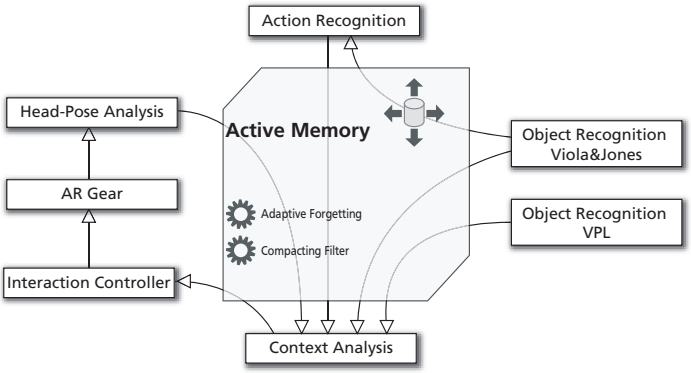


Figure 7.6.: *An exemplary cognitive vision system utilizing a single instance of an active memory.*

7.3.1. Concepts

A unifying property of many cognitive vision systems is that knowledge is generated, which is to be inspected later on, made available to other systems (or to humans) or re-examined when other relevant information becomes available. Learning processes need access to results generated earlier if conducted over longer time periods implying persistent storage of information, knowledge or data for later retrieval. Therefore, the function of memory is essential for cognitive systems.

Conceptually, the memory model introduces a *temporal* dimension in the otherwise transient communication patterns provided by the IDI architecture. It extends the referential (identities) and spatial (distribution) decoupling of participants as supported by the event-based integration functions introduced so far by a temporal decoupling.

In close analogy to the ideas presented during the discussion of the VAM concepts in Chapter 2, the memory model bases its terminology on that architectural sketch, which is used later on to discuss the characteristics of this rather complex integration pattern within the general IDI approach.

The memory model introduces the concept of an *Active Memory* that maintains dynamic representations of the world by correlating and consolidating events over time, e.g., to track the interaction state of cognitive system instances as will be exemplified in chapters 8 and 9. Information is stored dynamically, organized hierarchically and accessed by so-called *Memory Processes*. These processes perform the actual computations on the memory content including reasoning, fusion and learning. They also gather new knowledge from perceptions or allow interaction with the user.

Within this model, general events are mapped to *Memory Elements* as atomic information entities. The actual memory content at a given point in time is therefore defined as a *view* that contains the newest “generation” of correlated events. This virtual data space that is shared between otherwise

independent components is actively controlled by the memory itself and can be accessed by so-called *intrinsic* as well as *extrinsic memory processes*. While the latter are external processes, the former are executed synchronously within active memory instances. Figure 7.6 gives an informal example for a system that is integrated using the concept of an active memory.

The memory model features an architecture that is inspired by the concepts of tuple spaces, pioneered two decades ago by David Gelernter during research on the *Linda TupleSpaces* system [Gel85]. Linda [CG89] is the precursor of a generation of languages that aim at modeling and describing parallel algorithms without reference to any specific computer architecture. The basic idea is that different participants cooperate by reading or writing information through a virtual shared memory, which is termed a *tuple space*. Data exchanged via spaces is in general represented as tuples which are essentially ordered collections of primitive data types.

Apart from a large variety of prototypes, SUN Microsystems introduced a commercially supported software architecture called JavaSpaces [WA01] as part of the Jini [Sun08, Sun99] networking infrastructure utilizing tuple space concepts for integration and coordination in distributed systems. The term JavaSpaces already suggests that this implementation is targeted at the Java programming language and runtime environment. The JavaSpaces as well as the TSpaces [WMLF98] architecture by IBM enhance the Linda approach by adding a subscription and notification mechanism.

Functional Characteristics

Besides other aspects, the memory model introduced here extends those approaches by consequently applying the event-driven features introduced earlier for the design of a space-based active memory architecture. In the following description of the fundamental features of the memory model, both differences and similarities to the JavaSpaces concepts [FAH99] are outlined as a closely related approach to the memory model. The architecture of the ActiveMemory has the following characteristics:

- *Memory elements*: The *tuples* in the memory model are in fact event messages encoded according to an information driven representation as described in sections 6.2 and 6.3. Thus, each element is a tree-structured hierarchical document with optional binary attachments. This serves as a basis for dealing with versioning and extensibility of memory elements in very much the same way as described in the document model 6.2. Aiming to support potentially long-running cognitive vision systems, the latter is particularly important to enable visual memories for recalling of, e.g., memorized views of objects or actions. For an efficient use of binary attachments, it is possible to link multiple memory elements to a single copy of a reference-counted attachment.
- *Shared repository*: As within JavaSpaces, active memory instances allow distributed processes to interact with them concurrently. The active memory architecture deals with all details of distributedness, concurrent access and multi-threading permitting component developers to focus on the design of the abstract semantic interaction between processes.
- *Generative communication*: The communication model realized through the active memory is generative. As valid for general tuple spaces, events generated by attached memory processes feature an independent “existence” in the active memory instances. This is achieved by mapping transient events to memory elements. Any other memory process may remove generated elements as they are not bound to any individual process. Thus, locking of elements is not supported in the memory model.

- *Persistence*: An active memory instance provides reliable storage for its contained elements. Once a new memory element is generated, it will remain there indefinitely until it is removed. In the JavaSpaces architecture, a *lease* time can be specified [FAH99] for each tuple indicating how long an object shall be stored, whereas the active memory introduces the general concept of extensible *intrinsic processes*. Among other things, these intrinsic processes are capable of performing a garbage collection in the space of memory elements. This concept will be explained in more detail later in this section.

- *Atomic operations*: The active memory guarantees that any single operation on an element is atomic, which means that either the operation can be carried out successfully as a whole or not at all. In contrast to JavaSpaces that make use of Jini's transaction service [Sun99], the active memory intentionally not supports distributed transactions across multiple memory instances and operations of extrinsic memory processes. Therefore, the memory model shares the benefits of a state-less distributed system architecture [Bir05] as the memory server does not keep track of the state of its clients.

- *Associative lookup*: A key feature of most tuple spaces is that tuples are located by a kind of associative lookup, not by a concrete memory location or an identifier. In contrast to the JavaSpaces approach where templates with wildcards are used to locate tuples, the active memory concept applies content-based selection methods. Based on the document-oriented data model, XPath queries realize the associative lookup in the active memory. This allows a memory process to find required documents based on their content in very much the same way as described during the introduction of the observation model in Section 6.4, without having to know the type name of a memory element or the identity of the process who generated it.

- *Executable content*: As long as a memory element is contained in a shared repository, it is just passive data. It is not possible to modify it or remotely invoke any of its methods. However, when an element is retrieved from a memory server, a local programming language specific value object is created, whose fields and methods can be used as usual if their type is registered at the runtime of the IDI architecture. However, even if a specific type is unknown, the general event model as described in Section 6.3 permits accessing it via the generic event interface.

The modification of an active memory space is carried out by means of a small set of basic memory operations, which are *insert*, *replace*, *query* and *remove* as well as *take*. The methods behave as one can expect from their names and as illustrated in Figure 7.7. *Insert* stores a tuple, while *remove* deletes one or more of them. *Replace* exchanges an existing memory element with a new one. *Query* retrieves tuples from the memory. Finally, the *take* method deletes a tuple and, in contrast to remove, returns it. Both *query* and *remove* use a content-based selection statement to select tuples. The semantics of these five methods will be explained in greater detail later on in this section.

Observing the Dynamics of an Active Memory

In addition to these fundamental operations and the aforementioned general characteristics, which are - despite their different semantics - still comparable to state-of-the-art databases, the memory is not limited to these rather passive functions for managing the elements contained in a memory space. Instead, the active memory itself acts as an informer.

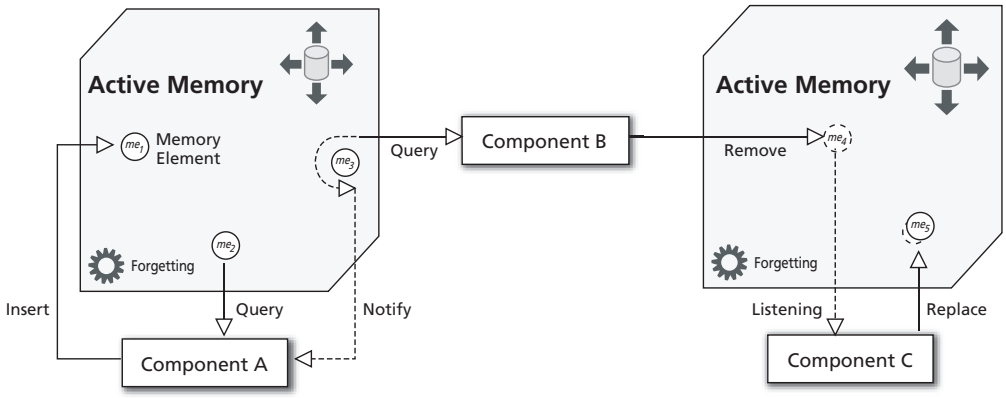


Figure 7.7.: *Interaction between memory processes is mediated via active memory instances.*

Upon any modification or assessment of memory elements through other system participants, the memory becomes active and publishes new events that contain either the original event notification or the modified memory element augmented with memory specific metadata information.

Figure 7.7 exemplifies this idea. All participants act concurrently on the elements that are contained in two active memory spaces. In this situation, participant A is notified if and only if element me_3 is queried and participant C is informed as soon as me_4 was removed from the second active memory shown. While these new events do not technically manifest themselves in a separate event type as introduced in the event model, the following definition of a memory event is developed from a conceptual point of view:

Definition 7.6 (Memory Event) *A memory event is generated upon the application of any of the basic operations provided by the active memory (insert, query, take, replace, remove) executed on a specific element within a memory container. It contains the involved element and additional metadata, e.g., about the type of operation carried out or the correlation identifier.*

In contrast to the template-based registration of listeners at a JavaSpace instance, listener participants interested in being notified about state changes in an active memory can use all the concepts that were introduced in the observation model. Thus, subscriptions are not limited to specifying all of the different types of memory actions (which was not possible, e.g., JavaSpaces only allowed notification upon insertion of new tuples matching a given template but not on their removal). Instead, interested listeners may use all the expressive power of the observation model in conjunction with these operations. For instance, it is easily possible to specify and register a subscription as shown in Listing 7.2 that states *"a notification shall be issued if a memory element of type FaceEvent with a recognition probability of 95% has been inserted or updated at least 10 times within a second"* in the observation model of an active memory listener.

There is another difference to the notification models of most tuple space architectures: the underlying event-based infrastructure guarantees that all participants which registered their interest in an information will eventually receive it. Within the IDI architecture, the notification and observation models ensure that all events are distributed to all subscribed listeners via the multicast event bus.

```
1 SynchronizedQueue<FaceEvent> faces = new FaceQueue();
2 Subscription s = new Subscription();
3 s.append(new TypeFilter(FaceEvent.class));
4 s.append(new MemoryFilter(MemoryAction.INSERT || MemoryAction.UPDATE));
5 s.append(new XPathFilter(new XPath("//HYPOTHESIS/RATING/RELIABILITY[@value>=0.95]")));
6 s.append(new FrequencyFilter(10,1,TimeUnit.SECONDS));
7 // add subscription to local am pattern object
8 am.subscribe(s,new QueueAdapter<FaceEvent>(faces));
```

Listing 7.2: *Extension of the plain Java subscription from Section 6.4 by a memory filter. This subscription, cf. Listing 6.3, only matches if face events were actually inserted in a memory.*

To achieve similar functionality in a space-based approach, the producer of an information would need to take care of this, e.g., by inserting a single tuple for all interested parties or checking against another global state information whether an element may be safely removed because all subscribers already received a copy. [CNF01]. Due to the use of the event-driven core layer of the IDI architecture and the fact that extrinsic memory processes register subscriptions for memory content at their local observation models, the server component of the active memory retains a stateless characteristic. Consequently, there is no need in the IDI architecture to attach lease times to subscriptions as it is done in the JavaSpaces model. If a listening memory process crashes, it is able to reconnect at any point in time to the event bus of the core architecture, re-registering its memory-related subscriptions at its local observation model.

Activating and Extending the Memory by Intrinsic Processes

In general, memory is a limited resource [Chr03], not only for computer systems but also for biological cognitive systems. Thus, some kind of garbage collection needs to be a basic quality of this memory model for cognitive systems. However, it is not clear that there is an optimal garbage collection strategy, as there might be other constraints on the data, most of which cannot be foreseen, in particular not by individual component developers. Thus, from a system-level engineering perspective it seems counterproductive to annotate every single information in a system with a specific time-to-live information.

In contrast to these lease-based approaches, the memory model allows users to extend its core functionality by introducing so-called *Intrinsic Memory Processes* that co-exist with the memory data in close coupling. As explained in Excerpt 7.3, *forgetting* is the prototypical example of an intrinsic process. It discards memory elements from the active memory repository based on metadata information. This metadata is available in almost every type of event exchanged within the IDI architecture, e.g., the time when an element has been updated last by a functional component of a system. As a consequence, other memory processes (either external or internal) can indirectly cause a hypothesis to be removed by changing relevant metadata. Due to the close coupling of this task to large parts of the memory content, processes such as forgetting can be realized as an IMP. The combination of forgetting and memory events is additionally useful for expiring requests in a space-based architecture because otherwise requests would remain in the system forever.

Another example of an IMP is the compacting filter that was introduced in Section 6.4, originally developed as an IMP that conditionally inserts new memory elements. The insertion only takes place if the calculated change to existing similar elements is significant enough, otherwise an already existing element will be updated.

Figure 7.8 provides an example of the interaction between extrinsic and intrinsic memory processes. In order to understand this example, let us shortly recall the hypothesis concept as introduced in Section 2.2.2, which is a domain-specific extension of a general memory element that turned out to be a fundamental concept for a visual active memory [HBS04]. The hypothesis type adds additional meta data like reliability or conflict values to each event exchanged and mapped to a memory element. This allows memory processes to transparently deal with any kind of hypothesis. One of these intrinsic memory processes is the aforementioned forgetting process that is supported through consistency validation by evaluating the added reliability information as explained in Excerpt 7.3.

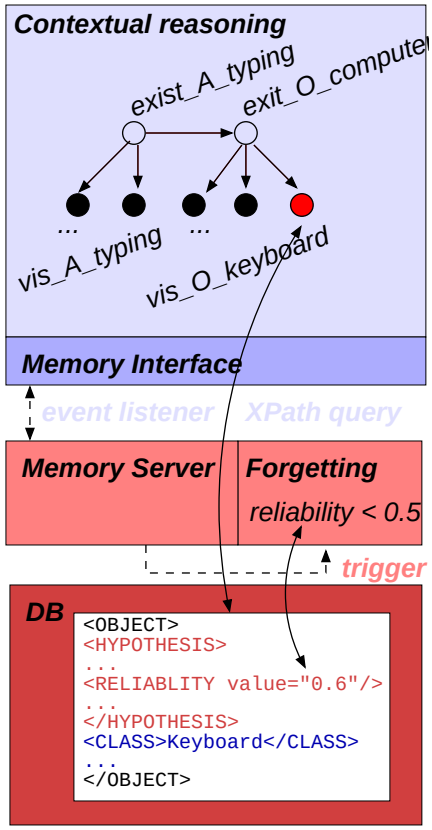


Figure 7.8.: *EMP/IMP coupling (from [WWHB05]).*

On the basis of the hypothesis concept, contextual reasoning is an important example for the utility of the memory model. It provides functionalities like scenery classification or consistency validation. Instead of being specialized to certain contexts and tasks, the realization of this component applies Bayesian networks [SSP00] for the interpretation and validation of memory content as well as model learning. It interacts with other perception and maintenance processes that feed new hypotheses into the memory or adapt the memory content, respectively, by observing the status of the corresponding memory elements. In order to actually perform the consistency validation, so-called *functional dependency concepts* (FDCs) are defined by the structure and parameterization of a Bayesian network, which can be learned from relevant memory content.

For example, in Figure 7.8 the FDC expects the user to be located in front of a computer and occasionally performing the action ”typing”. It is rather improbable to perform the action ”typing” without having a computer in the scene. If this situation occurs, the involved hypotheses have to be doubted, since they are not expected by the underlying model in the given context. Action and object hypotheses are interrelated by FDCs so that actions define a functional context for objects and vice versa. Conflicts between hypotheses are detected by calculating a conflict value as defined in [HBS04].

In active memory systems built according to this hypothesis concept, consistency validation modules provide important cues for higher-level processes even if the original event source is not able to rate the reliability or contextual suitability of its generated information. The added reliability values in this scenario are used to guide the removal of conflicting hypotheses through a forgetting process, which is illustrated in Figure 7.8.

Excerpt 7.3: An IMP Example: The Forgetting Process

Intrinsic memory processes can be declared using so called IMP specifications that consist of an initialization block, one or more memory event listeners and the corresponding algorithms expressed in a scripting language, which is Python in the current prototypical implementation of the virtual machine. A trigger specification is constituted by a content-based XPath condition that selects among the set of incoming events, e.g., hypothesis elements, as well as the memory action type.

Specification An example of such a declarative IMP specification for the forgetting process is shown in Listing 7.3. It consists of three trigger listeners and some initialization code, which are defined for the insert and replace actions within the memory as well as for a special *timer* event that is generated by the runtime environment in the given interval. The provided conditions match every hypothesis element inserted or updated by the memory instance. The respective algorithms of the IMP listeners are thus executed upon creation or modification of any hypothesis memory elements.

The Actual Forgetting In the process shown, forgetting is performed based on reliability information and a timestamp indicating when the element was updated last. Upon insertion of a new hypothesis, its reliability and updated timestamps are stored in an IMP internal dictionary utilizing the memory element identifier as a key attribute (*lines 6-13*). If the reliability value of a hypothesis is modified, the stored value in the internal dictionary is updated as well (*lines 14-20*). Correspondingly, a dictionary entry is updated if its timestamp information has changed or if the observed memory element has simply been removed (not shown here). The main task of the forgetting IMP is performed by the time-based listener (*lines 21-33*). It compares the updated timestamp with the current time, removes a memory element if it is older than 5 seconds and if its reliability is below a certain threshold (here 0.5).

Through the IMP concept, extension and adaptation processes like forgetting can easily be defined in a declarative way. The IMPs themselves can be reconfigured and stored as an element in the memory model. Through the execution of IMPs in the virtual machine architecture of the active memory, processes that are dependant on vast amounts of data can efficiently be realized.

At the time of this writing, contextual reasoning was realized as an extrinsic memory process. The active memory server triggers a single consistency validation step through implicit invocation of event listener while the evidential nodes of the prototypical Bayesian network are instantiated by memory hypotheses retrieved by using associative lookup during a bootstrapping phase. After this single consistency validation step, the reliability value of the memory hypothesis is updated. This in turn may trigger the intrinsic forgetting process to clean up the memory by removing the potentially doubted memory element.

From the perspectives of long-running cognitive systems, a persistent memory is important for temporal decoupling of producers (e.g., a process that acquires views of objects for training) from receivers of information (e.g., a long-running object classifier training process). From the viewpoint of information-driven integration a memory is important, because the responsibility for keeping state is shifted in event-driven architectures from informer to listener participants [Hoh06].

The principle that informers must not maintain state for their listeners promotes loose coupling and scalability again. Even though listeners may know about their informer's identity in the IDI architecture, they shall not make use of this knowledge, e.g., for retrieval of state information through a request-reply interaction as this would re-introduce the unwanted unnecessary coupling.

```
<imp name="forgetting" lang="python">
  <init>
    objects = []
    reliabilities = {}
  </init>
  <trigger type="insert" xpath="/HYPOTHESIS">
    <code>
      ts  = int(get_xpath('//TIMESTAMPS/UPDATED/@value'))
      rel = float(get_xpath('//RATING/RELIABILITY/@value'))
      reliabilities[vamid] = rel
      objects.append((ts, vamid))
    </code>
  </trigger>
  <trigger type="update"
           xpath="/HYPOTHESIS/RATING/RELIABILITY">
    <code>
      reliabilities[vamid] =
        float(get_xpath('//RATING/RELIABILITY/@value'))
    </code>
  </trigger>
  <trigger type="timer" interval="2">
    <code>
      current_time = time.time()
      for o in objects:
        timestamp, vamid = o
        if timestamp > (current_time - 5000):
           break
        if reliabilities[vamid] < 0.5:
           mi.remove(vamid)
           del reliabilities[vamid]
           objects.remove(o)
    </code>
  </trigger>
</imp>
```

Listing 7.3: *An IMP specification for a reliability-based forgetting process.*

Nevertheless, it is often important for a participant to *recall* past events in order to adapt their local state to the overall system state, e.g. for bootstrapping purposes during component initialization. Thus, a memory instance must have all data available, even if not all of it is stored. In such cases, the active memory is able to guarantee that all event-generating data is actually stored for later reference. In contrast to request-reply interaction between different components in a system to retain state information, the memory approach reduces coupling to a small number of identities, which represent the memory instances themselves, if a boostrapping phase is needed.

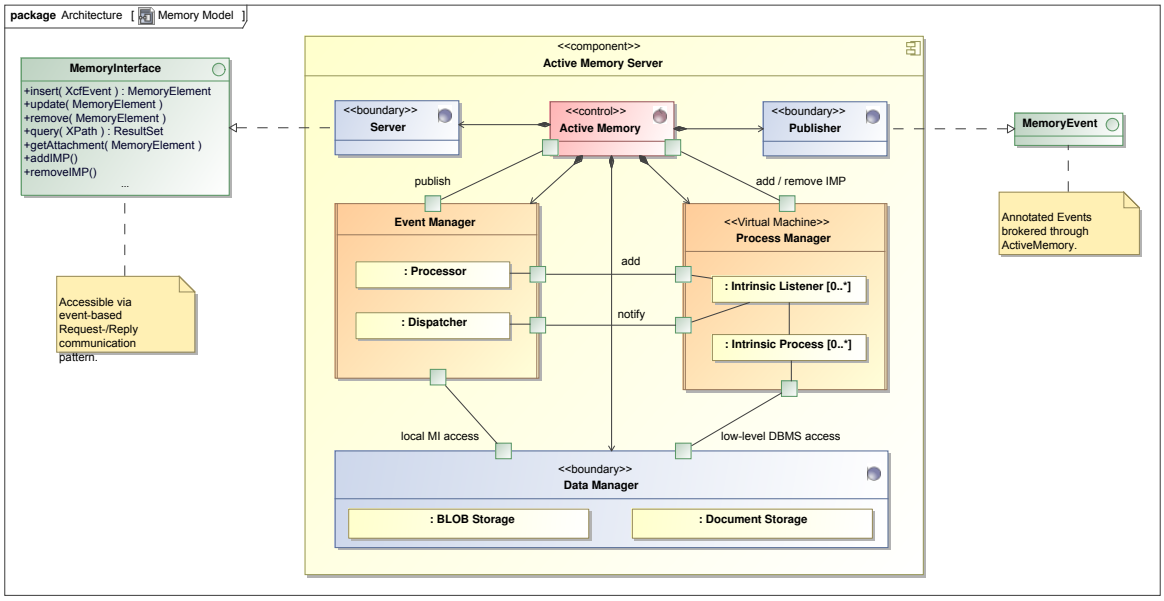


Figure 7.9.: *Conceptual architecture of the event-based memory model. Shown here is the server part of the active memory pattern.*

7.3.2. The ActiveMemory Architecture

The conceptual basis for the software architecture realizing the tuple space-inspired memory model follows data-centered opposed to task-centered paradigms. As a consequence, the architecture of the active memory is largely based on recent database technology that realizes the overall functionality of the memory model in conjunction with the functionality of the IDI architecture. The following paragraphs describe the conceptual software architecture of the memory server and its client processes in terms of general architectural styles, including the patterns and concepts introduced in the previous models.

The system architecture of the memory model is comprised of the aforementioned extrinsic memory processes, a database management system and an event processing subsystem, which shares commonality with the *router* component of the observation model as introduced in Section 6.4 as well as a runtime environment for intrinsic processes. Usually, there is exactly one database per memory, which is accessed by an arbitrary number of memory processes. While these may also communicate directly via the communication patterns introduced in the previous section, this is usually due to reasons that are conceptually independent of the memory. Therefore, the memory and its processes form a hub-and-spoke topology as indicated in Figure 7.6. From a software engineering viewpoint, however, participants are still loosely coupled as the memory makes extensive use of the event-driven patterns introduced so far and does not keep track of the state of its external clients.

In accordance with the other patterns presented in the previous section, the memory model is based on the principles and services of the resource model. Thereby, it offers location transparency through its client interface used by extrinsic memory processes. The client part of the active memory pattern provides an implementation of the `MemoryInterface` and encapsulates the communication logic

between client and server, currently based on a Request-Reply pattern, which is non-trivial, e.g., for query processing or the retrieval of binary attachments of queried elements. Additionally, it provides shortcuts for subscribing client-side listeners at the observation model that match specific memory events. As most of the used functions have already been explained in the previous section, we now shall concentrate on the server part of the pattern as it accounts for the advanced features of the memory model. These functions are located and realized in a single composite component, the so-called *active memory server*. First of all, it must be noted that multiple instances of this component can be executed as regular operating system processes that actually represent instances of active memories in order to partition the overall event space for scalability reasons. Each memory may additionally feature specific semantics as exemplified in the VAMPIRE systems explained in Chapter 8.

The Communication Layer - An Event-based Service Interface

Figure 7.9 shows the composite structure of the active memory service component featuring a three-layer architecture. Event and process manager constitute the core layer of the memory. They are sandwiched between the database management below and the communications management layer above. The latter is composed of two boundary components that connect the memory service with external IDI participants and the main control class for the overall active memory component.

In order to invoke the fundamental operations that can be applied on the data space of an active memory as exemplified in Figure 7.7, a request-reply pattern is used here. The remotely callable event request handlers in this server interface are:

- *Insert*: The insert request handler is equivalent to the *put* method in the tuplespace concept. It allows to store any type of IDI event persistently in an active memory as long as it is encoded in accordance with the document model explained in Section 6.2.
- *Replace*: The replace handler allows to exchange the content of a memory element with the contents of a new event. Since events represent a single observation, and thus a single point in time, the elements in a memory instance are not correlated on the basis of their event identity but on an additional memory-specific correlation identifier. Once an element in the memory has successfully been correlated, its predecessor is usually removed for resource reasons.
- *Take:* Similar to tuplespaces, the take request handler reads an element and removes it from an active memory within a single transaction. This request blocks if no matching element is available, but can also be combined with an asynchronous callback object.
- *Query:* The query request handler realizes the associative lookup of memory elements. It returns a list of elements that conform to a given content-based query statement and thereby corresponds to the read operation in the tuplespace concept.
- *Remove:* The sole purpose of the remove operation is to delete one or more elements from the active memory that conform to a content-based selection statement.

In addition to these fundamental methods, further request handlers for intrinsic memory process and database management are exposed in this remote API that realizes the external *MemoryInterface*. Requests accepted by the corresponding server pattern instance are delegated to the main control component. This internal *ActiveMemory* controller further dispatches them. Besides exposing the memory interface, a *publisher* pattern is used in this layer to realize the informer part of this service.

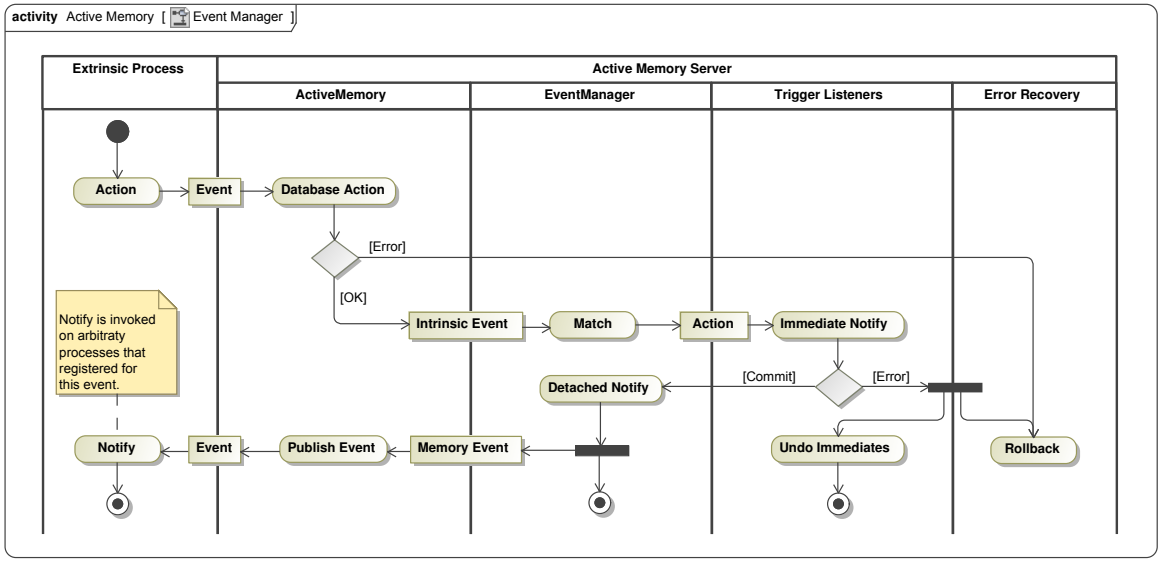


Figure 7.10.: *Activity diagram showing the fundamental processing steps in the memory model.*

Applying a publisher instance ensures that a unique event channel is created for each single memory instance. Each memory event can be multicasted by the publisher on the event bus of the IDI architecture, which in turn allows listeners to subscribe to individual events as exemplified earlier.

The Core Layer - Event Manager and Intrinsic Processes

The internal architecture of the active memory was largely influenced by performance and usability concerns. A usual issue with regard to distributed architectures and in particular with tuplespace approaches is inter-process communication delay. Large amounts of data, as they routinely occur in vision systems, aggravate this problem. The answer in the memory architecture is two-fold: first, *detached notification* and second, *intrinsic processes*.

The core layer of the active memory as shown in Figure 7.9 features an *EventManager* component that is fundamental for the realization of these two concepts. The interaction between event manager, communication layer and database management layer is performed via so-called *composite actions* and *intrinsic memory events* [Lüt04]. Composite actions encode an arbitrary sequence of operations to be carried out on the memory content. For instance, a memory interface operation such as *replace*, constituted by a query operation is followed by an update of the retrieved target element. Each action runs within its own transactional context and is executed by the active memory controller in its own thread of control. Figure 7.10 shows the processing steps carried out for handling a received request. Upon successful completion of an action performed on the database, the actions themselves create intrinsic memory events about the operation and the affected information. This allows the memory to publish fine-grained events as for instance in the case of an associative *remove* that affects a larger number of elements. In this situation, a single notification is sent out after each deletion of an individual memory element.

As actions should not be aware of higher-level functions of the active memory, the actual responsibility for forwarding events to other architectural layers is contained in the event manager component. The event manager augments and forwards each intrinsic event upon completion of its transaction to the publisher instance of the active memory where it leaves the active memory component and is placed on the general event bus. The published memory event in turn leads to an event notification in the extrinsic memory processes, subscribed to this specific event at their local observation model.

From the perspective of the active memory server, the *detached notification* realized by its publisher is vital. It allows processing in the memory component to continue without having to wait for event receipt, thus minimizing delay. Conversely, the same applies for the processing in the extrinsic listeners as they can already begin to react even if a complex memory modification may not have been finished yet. Taking up the deletion example again, it might be more important for an extrinsic listener to get notified about the removal of a specific memory element as fast as possible than about the fact that the overall removal operation was completed successfully. It has to be stated that its in the responsibility of the IDI publisher pattern to perform an asynchronous notification, which actually detaches event distribution in the active memory from its listeners. Moreover, the IDI publisher pattern guarantees that each subscribed process will eventually be notified about the occurrence of the event given with a short delay after the successful completion of the operation.

If a transaction fails, memory events are not sent via the publisher instance. Otherwise, the overall system state could quickly become corrupted due to partial failure. Furthermore, detached notification provides an avenue for a stateless implementation of the memory server component, because it does not need to track the subscription status of its clients. The concept of detached notification is crucial for successful operation of active memory systems. It enables participants to react to changes in the observed memory content with low latency and exchange time-critical information via the active memory patterns.

In addition to detached notification, the second mechanism to be introduced to increase the performance of the memory model are the so-called *intrinsic memory processes* (IMP). These are extensible programs that can be registered in an active memory at runtime and executed under certain constraints directly in the memory component. The specification of constraints and the matching of events against the resulting subscriptions is similar to what is done in the router component of the observation model. In this case, the active memory uses the same content-based matching strategy that takes the given document model as a basis; supporting equal expressiveness of (stateless) conditions as provided by the general observation model. As the name suggests, the event manager assumes the role of the observation model in the active memory. In order to achieve this in conjunction with a database backend, the design of the event manager follows the Event-Condition-Action (ECA) model [DBM88], which is a common pattern in active databases, realizing conditional event processing. Local action callbacks, which are intrinsic listeners attached to IMPs as shown in Figure 7.9 can be registered at the event manager and will be invoked if their condition matches an intrinsic memory event. The ECA pattern is applied here to allow for a flexible execution binding of the intrinsic memory processes and for supporting undo and rollback of intrinsic notifications. This pattern coordinates access to a database in a transaction-oriented way, which has a different focus than that of an observation and notification component in a distributed event-driven architecture.

Summarized in Table 7.1, intrinsic and extrinsic processes differ in several ways. Considering the data flow within the active memory and the semantics of IMPs, the first important difference is that IMPs are executed by *immediate* notification. In contrast to detached notification, intrinsic processes are executed within the transaction context of the enclosing memory operation that triggered an intrinsic

Attribute	*Intrinsic Memory Processes*	*Extrinsic Memory Processes*
Transaction	Inside	Outside
Access	Full DBMS backend	Client API
Coupling	Strong	Low
Execution	Synchronous	Asynchronous
Environment	Virtual machine	Any (with client API)
Examples	Forgetting, Statistics	Object recognition, Information fusion

Table 7.1.: *Characteristics of intrinsic (internal) and extrinsic (external) memory processes.*

listener of the IMP. A second distinguishing feature is that the callbacks of IMPs are invoked synchronously. When an immediate notification is performed, the memory passes control to the intrinsic listener that is to be notified and waits for it to return before proceeding. This ensures that an event was received and, if necessary, processed.

It is for these reasons that IMPs permit to change the behavior of the memory by, e.g., updating statistics, updating referenced documents or perform memory optimizations. For external processes, this constitutes a transparent change in the behavior of the memory model in a specific system context. Thus, intrinsic processes are a generic mechanism to change memory behavior and to provide modifiability.

The basis for intrinsic processes is a virtual machine architecture [SG96] running inside a memory operating system context as indicated in Figure 7.9. This in-process execution of IMPs reduces notification delay and enables access to the full functionality of the database backend. The close coupling between the database and the fact that no network or inter-process communication is necessary makes it additionally practical to work on large amounts of binary data that can be stored in an active memory. Additionally, the virtual machine architecture serves to isolate data structures, thus protecting the memory from malicious behavior of erroneous intrinsic processes. Besides being possibly erroneous, a much more common problem of IMPs can be that certain database actions, which are executed by an IMP within the provided transactional context may fail so that the enclosing operation fails as well. In this situation, the state of the underlying database is guaranteed to be consistent even if modifications through intrinsic processes were already effected due to the complete rollback performed by the database layer.

However, the same does not apply for the internal state of intrinsic processes. If IMPs received a number of notifications within a complex transaction, their internal state might may have become inconsistent. In order to handle this situtation, the memory will notify all previously involved IMP listeners about this exceptional occurrence. All memory events within a transaction are therefore recorded by the event manager, which allows to dispatch them in reverse order to the undo operation of subscribed intrinsic listeners. It is then the responsibility of the intrinsic processes to react accordingly to these situations and adjust their internal process state. Further details about the realization of a Python-based Virtual Machine for IMP's can be found in Excerpt 7.4.

Excerpt 7.4: A Virtual Machine Architecture for Python-based Intrinsic Processes

The memory model supports the execution of intrinsic memory processes. Listing 7.3 shows an exemplary forgetting process based on Python. Python is a dynamic, object-oriented, high-level language that provides both a compiler and a virtual machine to run compiled bytecode [vR04]. The virtual machine architecture of the active memory actually embeds Python and connects it to the event manager component, performs lifecycle management for IMPs, binds internal objects to IMPs and schedules their execution. As shown in Listing 7.3, IMP specifications are submitted to the active memory at runtime by extrinsic processes in the form of an XML document.

Activation The *process activator* of the virtual machine component features a built-in listener subscribed for the insertion of new process specifications in the active memory by matching `imp` on the root tag of the process specification messages. Upon notification it evaluates the specification in order to create a python object from the provided python scriptlets. The `init` block becomes a Python constructor, all other actions are translated into member functions. The process activator will then proceed to compile that class and instantiate an object. The intrinsic listeners are actually bound with their subscription to the generated member functions and wrapped in a local C++ callback object registered at the event manager of the memory. This wrapping is carried out using the Python C/API [vR08] and Boost.Python [AGK08]. The process is commonly denominated as *embedding* Python [vR04]. Wrapping and registration of intrinsic callbacks complete the instantiation of an immediately dispatchable IMP listener from Python code.

Binding The primary usecase executed within IMP listeners is memory modification. It is the only meaningful way of changing the externally visible behavior of a memory system. In order to facilitate this, IMPs can use the same basic memory operations as are exposed to extrinsic processes via a python-based memory interface that wraps a corresponding C++ class. In contrast to embedding python into C++, this is the reverse situation. It is commonly referred to as extending Python and is integrated through a loadable module (shared object), dynamically loaded at runtime by the Python interpreter.

Execution For reasons of transactional control and usability, a reference to the instantiated C++ memory interface is bound to a predefined variable at each invocation of an IMP listener. Please note that the `self` pointer can regularly be used as internal object reference in order to share variables between the various actions. The C++ `Event` class has been wrapped for Python, and is made available as another predefined variable in order to reference the event that just happened from within a python listener method. While an action has no return value, it may raise an error, which is passed through to the calling C++ code in the event manager. Like other event listeners, intrinsic processes are called when an event matches a specified subscription. Due to the introduced wrapping procedure, this becomes a simple method call into the Python virtual machine. When all registered listeners have successfully completed their computations, the XML information is written to the repository backend.
For more information on the virtual machine architecture of the active memory and more details on extending/embedding and the rationale behind the choice of *Boost.Python* as a wrapper generator, please see the corresponding technical report [Lüt04].

The Database Management Layer

The lowest layer of the active memory is a persistent storage engine for multi-modal information. The data management subsystem wraps two individual databases, one for XML-encoded textual data and another one for binary-encoded attachments such as cropped image patches which are stored for long-running subsequent classifier training. Besides realizing a reference counting and linking strategy

for binary attachments of memory elements, the database manager serves as a wrapper facade that encapsulates database management functions like opening and closing of data containers, deadlock exception handling etc. from higher-level layers. Most importantly, it is responsible for creating transactional contexts provided to the aforementioned database actions, which access the underlying database natively and support potential database rollbacks.

Coherently with the other models in the IDI architecture, the document-oriented data model was adopted in the memory model, too. Integrability and modifiability were a deciding factor that also had to be taken into account for the technical design and evaluation of the data management component of the active memory. It quickly became clear that database support was needed for the sake of re-use, e.g., with regard to the aforementioned support of transactions, and the available resources for the project in order to be able to realize the concepts of the memory model.

A native XML database concept was chosen as the technical basis of the memory model. This has been beneficial for several reasons:

- *Document model*: Native XML databases directly supported the XML-based data model as XML documents are fundamental units of (logical) storage, which means that there is no complex and potentially resource consuming mapping between hierarchical and, e.g., relational data models [MK02] necessary.

- *Schema independence*: Given the free data model as described in Section 6.2, which aims at general applicability for cognitive systems, it would be counterproductive if the information mediated via an active memory was required to conform to a certain predefined structure in order to get persisted. Luckily, native XML databases usually support schema independence, which frees developers from the burden to provide database schemata for every type of information exchanged via the memory model. This greatly improves the actual usability of the active memory by minimizing knowledge and effort needed for database management.

- *Query languages*: Taking up the general concept used in the observation model, memory elements that match a particular set of content-based conditions can be retrieved in native XML databases either by XPath or XQuery statements. This support for content-based declarative queries already realizes the required associative lookup of memory elements within an active memory space.

As the Berkeley DBXML supports all of these requirements and features high performance, this database was selected as backend storage for the active memory. Using this database had the additional benefit that the handling of binary attachments was facilitated by the Berkeley DB non-relational data store, which is itself a dependency of the Berkely DBXML and therefore represents no additional dependency for the active memory. The active memory realizes a linking mechanism for binary attachments that is based on RDF descriptions [KC04]. As an embedded database, the Berkeley database libraries link directly into an application, but is in contrast no complete database server. Its therefore small footprint permits an efficient implementation of the active memory server, based on the introduced event-driven networking infrastructure without any additional unwanted client/server communication. Excerpt 7.5 details on some of the advanced aspects of the Berkeley DBXML chosen as repository backend.

Excerpt 7.5: An Embedded XML Database as a Basis for Active Memory Spaces

The Sleepycat [a] Berkeley DBXML [Sle06] (BDBXML) is an open-source database that provides the technological backend for the memory part of the IDI architecture. It is packaged as an embedded C++ library and supports transactions as well as multi-threaded environments. In contrast to classical database servers it offers only core features and is not a complete server application. The Berkeley DBXML stores XML documents directly, without mapping them to another data model. Queries are made using XPath 1.0, an additionally included query optimizer utilizes user-defined indices. Indices are specified using the names of tags, without reference to a schema and can be added, removed or modified dynamically at runtime. While for the implementation of the memory model the Berkeley DBXML was used in version 1.2, recent versions support the full XQuery recommendation as query language on XML data - besides other improvements.

Although a number of different database approaches were tested during the initial evaluation phase, which was performed in early 2003, e.g., also relational databases such as MySQL, it turned out that the BDBXML provided an excellent match for the realization of the persistence features of the active memory. Besides taking credit for the active memory concept by the makers of the DBXML database[b], further usability experiments were carried out, which we additionally reported to the broader public [LW04] that underlined the suitability of this approach.

However, utilizing an embedded database comes with certain costs. For instance, with the Berkeley DBXML, the developer is responsible for reacting to deadlocks. These can occur when two concurrent actions each hold a lock on a database page the other action needs next. When encountering such a dead-lock, DBXML will throw an exception and the typical reaction is to retry the action immediately, giving up after a certain number of retries [Ber04]. This is the approach taken by the active memory. It is transparent to the client developer, who will only see a `DatabaseException` after all retries have been failed.

[a] Sleepycat Inc. was acquired by Oracle in 2007.
[b] The active memory concept won Sleepycat's DBXML innovation award in 2004.

The memory model extends the set of interaction patterns presented in the previous section in two directions: on the one hand, a set of interaction operations inspired by tuplespaces permit distributed coordination through generative communication. On the other hand, this paradigm is coherently integrated with the so far presented models of the IDI approach. In addition to the regular event-based communication, it provides *mediated* event-driven communication, resulting in a new type of events that extending regular ones by additional semantics based on the state of the corresponding memory elements. Therefore, distributed interactions can be flexibly composed out of the set of memory actions and the corresponding events. While this is still an example of implicit coordination, the following section introduces a method for external coordination of participants.

7.4. Coordination Model

In addition to providing patterns for interactions between individual components on an architectural level and utilizing functions of the observation model for developing implicit coordination strategies, an explicit coordination model for event-based components is needed on a system level. The availability of a system-level coordination mechanism that permits modeling of complex component or service interactions yields functional component implementations that are easier to develop, integrate, test and reuse.

The coordination model thus not only contributes to the functional requirement of modeling and synchronizing high-level system interactions but also supports the fulfillment of some non-functional requirements originating from the software engineering aspect, cf. Chapter 5 such as modularity, declarativity and testability.

Additionally, explicit coordination facilitates the integration of modular event-based cognitive systems for the following reasons:

- *Control of Asynchronous Interaction*: While in general, control flow is highly asynchronous and decoupled, arbitration and hardware access demand for sequenced and coordinated execution. Realization of such arbitration processes is eased by a generic reusable coordination model. In the VAMPIRE project, an exemplary use cases is multi-model interaction control as will be exemplified in Chapter 8.

- *Complex Event Processing*: While this term denotes a complete field of research in its own right, the proposed coordination model yields a simple but yet general variant for evaluating complex event sequences. The coordination model excels in this regard beyond what is possible with the introduced observation model due to its ability to combine individual subscriptions exploiting the full expressiveness of Petri net semantics. Thus, complex stateful event sequences can be matched and new higher-level events generated.

- *Component Adaptation*: Instead of developing components which are very specific due to their coupling to expected system states, often mirroring highly complex system state models, components can be further decomposed if their application specific adaptation is done externally. Hence, it is possible for component implementations to make fewer assumptions about their operational execution context as the responsibility for keeping track of complex stateful interactions is separated. While a flexible configuration system may permit this with regard to static properties, the coordination model achieves this in terms of their runtime dynamics.

Therefore, this section introduces a simple, yet expressive approach based on Petri nets [Pet81] for the modeling, analysis and execution of complex tasks or action strategies, which seamlessly integrates into the IDI architecture. Petri nets are well suited as formal underpinning of the coordination model.

They extend classic state machines by the ability to represent concurrency. Thus, they are well suited for modeling structure and behavior of parallel distributed systems such as discrete event-driven architectures. Within robotics research, petri nets were already widely used for different purposes, e.g. to model tasks and actions under temporal constraints [MPV00] or for behavior selection [KCkPK05]. Utilizing Petri nets, a formal model can be developed that can be independently tested, which permits a declarative specification improving changeability and that aims at high utility in terms of execution performance.

7.4.1. Formalizing Coordination with Petri Nets

The coordination model utilizes marked petri nets that are extended by guards, which couple state transitions in the petri net to external events that are observed utilizing the features of the IDI observation model.

In general, a petri net is a bi-partite, directed graph, which consists of *places* that may contain tokens, *transitions* and directed *arcs* that connect places with transitions. The so-called *marking* describes the current system state by the number of tokens which are present in the places at a given point in time. The current marking of a petri-net corresponds to the actual state of, e.g., a modeled action sequence, robot behavior or other dynamics aspects in a cognitive system. Computations are triggered by the firing of *transitions*.

In the subsequently introduced variant of Petri nets, the firing of a transition is made dependant on the evaluation of a *guard* function.

Definition 7.7 proposes a formal description for the type of high-level petri net developed for the coordination model that integrates ordinary petri nets with the $match$ function of the observation model through the aforementioned guard function.

Definition 7.7 (Guarded Petri Net) *A guarded petri net in the IDI coordination model is a six-tuple* $\mathrm{GPN} = (P, T, I, O, g, M_0)$ *where*

- $P = \{p_1, p_2, \ldots, p_n\}$ *is a finite, non-empty set of places;*
- $T = \{t_1, t_2, \ldots, t_m\}$ *is a finite, non-empty set of transitions;*
- $I = P \times T$ *represent arc connections from places to transitions with an assigned weight* w*; if an arc exists from* p_l *to* t_j*, then* $i_{lj} = w$*, otherwise* $i_{lj} = 0$*;*
- $O = T \times P$ *represent the arc connections from transitions to places with an assigned weight* w*; if an arc exists from* t_j *to* p_l*, then* $o_{jl} = w$*, otherwise* $o_{jl} = 0$*;*
- $g : I \rightarrow \{true, false\}$*, where* $g((p_l, t_j)) = \begin{cases} true, & \text{iff } i_{lj} = 0 \\ true, & \text{iff } i_{lj} > 0 \wedge match(s_{lj}, D_X) \neq \varnothing \\ false, & \text{iff } i_{lj} > 0 \wedge match(s_{lj}, D_X) = \varnothing \end{cases}$ *and* $p_l \in P, t_j \in T$*;*
- $M_c = (m_{c_1}, m_{c_2}, \ldots, m_{c_n})$ *represents the marking of a petri net.* M_c *is a vector in non-negative integer* $|P|$*-space. The* i*-th element of a marking,* m_{c_i}*, specifies the number of markers in place* p_i *at time instant* c*.* M_0 *is the initial marking of the net;*

and $P \cap T = \varnothing$ *while* $match(s_{lj}, D_X)$ *represents the matching of a trace of received event notifications* D_X *against a subscription* s_{lj} *registered at the observation model. The flow relation* F *can be defined as* $F \subseteq I \cup O$*.*

In contrast to colored petri nets [Jen91], which feature similar guard functions that are defined on data available within an associated place, the approach here couples petri net execution to external information through the binding of input arcs $(p_l, t_j) \in I$ to regular IDI event subscriptions.

Input arcs, $(p_l, t_j) \in I$, of transitions are *satisfied* if the marking of the input place corresponds to the *weight* i_{lj} of the arc which specifies the number of tokens that this arc consumes from its input place. In ordinary petri nets, the fulfillment of this condition for all input arcs of a transition would *enable* it, eventually yielding its firing.

However, the IDI petri net model features a slightly more complex firing rule. It changes semantics of enabling a transition and adds a precondition to this enablement. This additional step is termed *activation* and is described in Definition 7.8.

Definition 7.8 (Extended Firing Rule) *IDI guards add an additional condition to the process that decides whether a transition $t_j \in T$ is firable or not:*

1. *An individual arc $(p_l, t_j) \in I$ with $p_l \in P$ is activated by some marking M_c, denoted $M_c \rhd (p_l, t_j)$ iff $i_{lj} <= m_{c_l}$.*
2. *A individual transition t_j may then subsequently be enabled by some marking M_c, denoted $M_c \blacktriangleright t_j$ iff $i_{lj} <= m_{c_l} \wedge g((p_l, t_j)) = true \,|\, p_l \in P, 1 \leq l \leq n$.*
3. *Iff $M_c \blacktriangleright t_j$ then t_j may fire.*

This definition formally describes that once a sufficient marking is available in a place that is attached to an input arc of a transition, this arc is activated. *Activation* in this context means that the guard function for this arc is constantly evaluated in each processing cycle. Thus, the subscription that is attached to this guard is itself dependant on the marking in the Petri net.

Only the set of notifications D_X that is retrieved since arc activation and as long as the corresponding place is satisfied is matched for this listener. This is particularly important as not all listeners are context-free and often only provide meaningful semantics in the context of a modeled state.

Thus, the corresponding transition can only be enabled once the place's condition is fulfilled and the specified subscription has matched one of the events that were received in the observation model since its activation, fulfilling the additional guard condition. If a transition is enabled and subsequently fired, it executes a list of registered action callbacks, which may include the generation of new events that are sent to other system components. Thus, the effected changes in system behavior is bound to the observation of external events, which are evaluated in the context of the current system state.

This is the fundamental concept that couples the execution of the specified high-level petri-net model to the overall state of a system. If all input arcs of a transition are satisfied, the attached transition is enabled and the marking can be propagated to following state by applying the equation described in Definition 7.9.

Definition 7.9 (State Transition) *If a transition $t_j \in T$ fires, then the marking M_c is changed to M_{c+1} by the iterative application of the state transition equation:*

$$m_{(c+1)_l} = m_{c_l} - i_{lj} + o_{jl} \,|\, 1 \leq l \leq n$$

7.4.2. Development, Analysis, and Execution

Figure 7.11 depicts how a petri net-based coordination model process is coupled to the overall system. In terms of the IDI architecture, an instantiated petri net model represents a complex service component that observes events by subscriptions which are registered dynamically upon evaluation of guard specifications and that generates new events as soon as transitions fire.

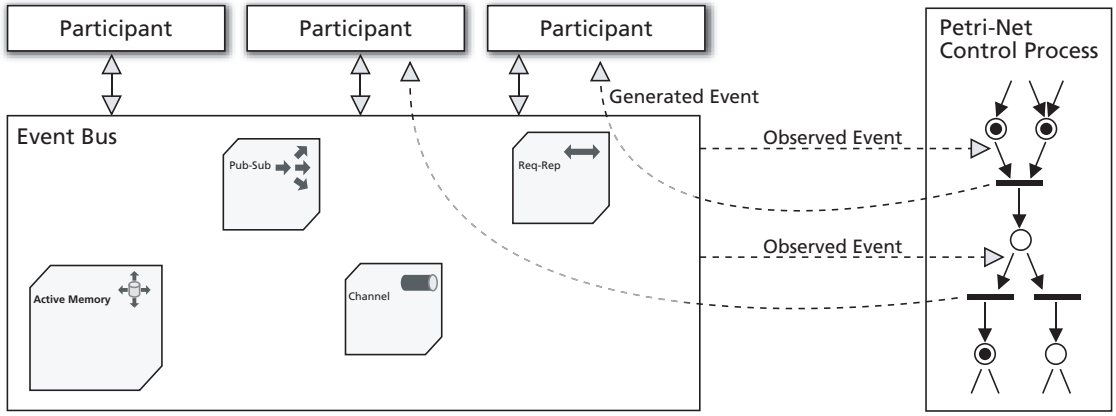


Figure 7.11.: *Observed events yield in firing transitions, which in turn may generate new events that modify system behavior. By utilizing a petri-net based approach, an event-based control loop from perception to action can be modeled and directly instantiated.*

These events result from the aforementioned action sequences that can be attached to any transition. The library that realizes the coordination model provides a number of reusable basic actions based on the patterns provided in the interaction and memory models such as (a-)synchronous request-reply communication, or active memory access.

Additionally, it is possible to exchange generic data items encoded according to the document model through a pre-defined local blackboard or to perform any action if their implementing classes are derived from a basic `Action` interface.

An example of reusable actions generally applicable are invocations of the service management functionality available in the IDI architecture. This allows for instance to restart or reconfigure a component, which realizes these management event handlers.

Figure 7.12 shows a screenshot depicting a fragment of the petri net model that handles user interaction in the VAMPIRE assistance system, which will be described in more detail in Chapter 8. The screenshot shows a session in the *Workflow Petri Net Designer* [Tho05] (WoPed). A tool like Woped facilitates the interactive development and testing of petri nets, even without other system components running. Besides invoking transitions manually or through replaying of observed events, petri nets can be analyzed for their logical soundness, e.g., with regard to reachability or liveliness [MFP06].

The realization of the coordination model permits a formal and declarative specification of net structure and guards as well as the attached actions in an application of the XML-based PNML document format [WK03].

This provides an avenue for extending petri-net coordination models at runtime by new places and transitions. The memorization of PNML models in an active memory space permits a dynamically reconfiguration of instantiated petri-net execution engines.

Further examples for the utility of modeling system behavior with petri-nets will be given in Chapter 8 along the explanation of the VAMPIRE system architecture.

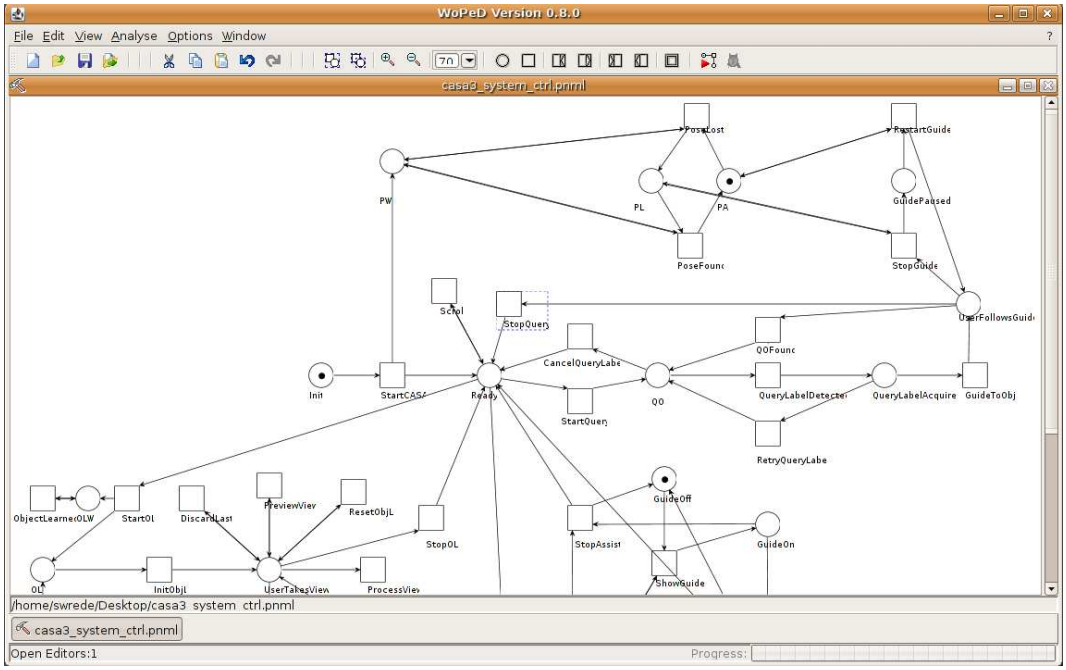


Figure 7.12.: *Interactive modelling and simulation of Petri nets with Woped [Tho05]*

7.5. Domain Model

Whereas the memory and coordination models provide essential functions geared at cognitive systems developed according to the visual active memory paradigm, they are still rather generic and domain independent as the other IDI models, too. Due to the fact that the IDI architecture shall be usable across the broad range of scenarios in cognitive interaction technology systems, a one-size-fits-all approach is neither desirable nor feasible. Thus, the overaching aim of the integration architecture is to easily support the development of different domain specific models.

Even so, the functional services already integrated with the IDI architecture, supporting specific functions in the domain of cognitive systems, e.g., a face recognition service [Lan07], represent a type of domain support available for reuse in other system instances. The aforementioned face recognition component can be easily integrated into both a robotics as well as cognitive vision system, which further eases the development and prototyping of novel types of composite services.

However, domain-specific functions from the perspective of an integration architecture and as explained in Chapter 5 must rather provide features on a technological level, e.g., reusable datatypes and algorithms or support for computer vision toolkits. While it is non-entertaining to describe all functionality that has been integrated in a reusable way on the basis of the IDI Architecture, the following sections shall highlight some prototypical examples for available domain support in this sense, each of which can coarsely be classified in one of three types. Of particular interest in the context of the VAMPIRE project was to support different computer vision toolkits, which is the reason that the *adapter* plugins developed for the IceWing [LWHF06] prototyping environment are briefly described as an example for this class of domain support, while *type libraries* and other *domain specific libaries* shall only be mentioned shortly.

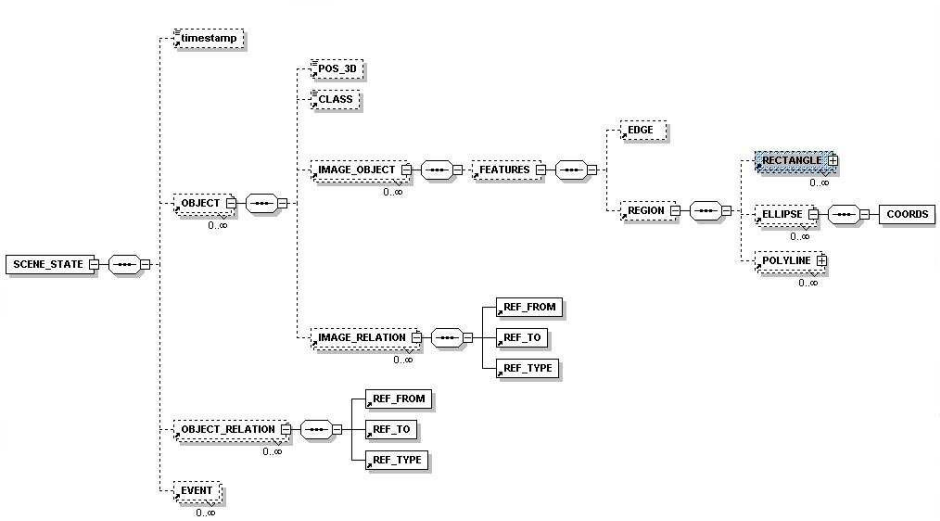


Figure 7.13.: *Graphical visualization of an XML schema used in the VAMPIRE project. It specifies the syntax of valid XML documents which describe memorized scene states.*

7.5.1. XML Type Libraries

As introduced in the document model, cf. Section 6.2, XML document types are the basis to describe event content transmitted, stored, and processed by the various services in an IDI-based cognitive system. In order to facilitate the integration process, a first step is thus to develop XML vocabularies such as the Computer Vision Markup Language [LF04] (CVML), which represents a similar approach developed in the context of Psyclone, cf. Section 5.3.2.

A suitable formalism - among other approaches like RelaxNG - for specifying an XML language such as CVML are *XML Schema* [BDG01] specifications. Figure 7.13 visualizes an XML schema for the description of a scene state at a given point in time as developed early within the VAMPIRE project. Schemas were defined for events generated by individual services such as head pose tracking, action recognition and many others. Similarly, in the COGNIRON EU project a large number of different schemas for symbolic robotic data (e.g., states, events, objects, etc.) were developed [FKH+05][1] yielding a formal data model and permitting runtime validation.

While XML schemas are useful to specify and validate implementation-independent interfaces with regard to certain document and thus event types, different language-specific libraries are available that contain these event types, thus their XML marshaling and possibly additional utility functions. These libraries often translate events with binary attachments into a domain specific API that is accessible for developers and beyond what is generalizable in the IDI architecture. A typical examples for an event type that is commonly provided by XML type libraries are image events, which encapsulate one or more images grabbed by an image service and translate the raw data into the specific representation needed in a particular technical environment.

[1] See also: `http://www.cogniron.org/wiki/DataStructures`

These libraries are usually shared across single organizations or between a smaller number of collaborators in a more closely controlled integration context, cf. Chapter 3. Even so, in contrast to the mandatory use of IDL libraries with stubs and skeletons as known from operational middleware, developers are not obliged to use the XML type libraries in order to access event information. Thus, a different process may interpret an event notification in a different, possibly more suitable way, e.g. by evaluating only specific fragments of the document contained in a event notification, which allows for developing very generic and loosely coupled processes like coordination, forgetting or anchoring processes, which do feature any compile-time dependency to specific event types and corresponding libraries.

Due to the fact that the exchanged information between functional services in a system architecture is often application specific, no extra effort was made to wrap all functionality that is available in a complex domain specific library such as OpenCV. However, as the effort to encode a necessary type according to the IDI document model is rather low, necessary types were added on a case-by-case basis to the domain model of a particular application when needed.

7.5.2. Application Adapters for Computer Vision Tools

Application adapters retrofit otherwise monolithic standalone applications into a larger system architecture. Due to the anarchical or at best oligarchical integration context in experimental cognitive systems research, cf. Chapter 3, the development of adapters is a common task in collaborative research projects. Adapters permit the integration of already existing legacy applications that are not based on the IDI architecture by means of, e.g., source-code modification, buddy processes, library replacements and other approaches [Bir05]. Most of these mechanisms are tied to certain applications, sometimes even to particular component configurations, increasing coupling and leading to poor reusability in other scenarios. In contrast, the IDI architecture supports the development of reusable adapters.

In order to extend the ideas of information-driven integration and provide reusable adapters in the domain of computer vision and pattern recognition, a previously monolithic computer vision toolkit was refactored. This toolkit, IceWing [Löm08], was originally developed as an infrastructural contribution of a dissertation on object learning [Lö04]. Thus, while other modular toolkits like Neo [Rit], which has its roots in neural networks research, are well integrated with the IDI concepts, too, the following paragraphs sketch the approach we realized for the modular development of computer vision services using a plugin-based architecture.

The Graphical Plugin Environment IceWing

IceWing is a graphical plugin shell optimized for the special needs arising in the field of vision system development. In contrast to the desired level of loose coupling on the system level, vision components may often need to run in a closely coupled execution context. Correspondingly, encapsulation and communication overhead should only have a negligible impact on the overall execution speed. Largely the same as for the overall architecture, a vision toolkit should help to develop and optimize the single components as well as integrated subsystems. Therefore, easy and fast introspection of any intermediate results as well as easy and flexible modification of algorithm parameters at any time are key points. As will be explained subsequently, these requirements are well covered by IceWing.

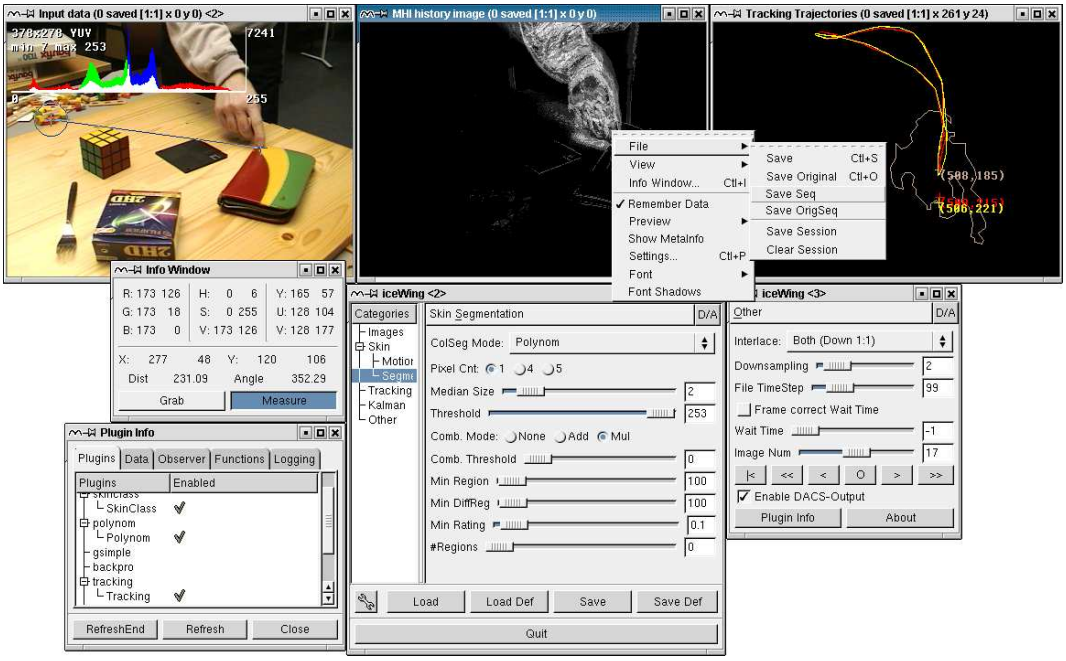


Figure 7.14.: *Exemplary screenshot of the IceWing image processing toolkit as typical representative for a domain specific software development environment. IceWing allows for flexible local coupling of image processing plugins and interactive prototyping. Generic infrastructure plugins are provided by the IDI architecture, yielding versatile and reusable application adapters for computer vision subsystems.*

Plugin Interaction Figure 7.14 shows a typical runtime session with the graphical plugin shell. IceWing itself provides solely an administrative core, an initially minimal user interface and a variety of support functions for tasks like user interface creation, communication and introspection. The real functionality for the task the user wants to solve with IceWing is provided by dynamically loaded plugins, which are realized as standard shared libraries. For the component designer, plugin development is as easy as deriving from an abstract base class and implementing a *process* method that carries out computations on data passed to the method by IceWing. During these computations plugins can take advantage of other external libraries, for example OpenCV or RAVL for enhanced image processing functionality. Plugin development can be done in C and C++. Additionally, bindings for the scripting languages Python and MATLAB are available.

For interaction between plugins two distinct communication patterns are provided, which actually feature similar semantics as the patterns introduced in Section 7.2, but operate on an in-process level instead of the system level [SSRB00]. On the one hand, a realization of the Observer pattern [GHJV95] implements a process local variant of the Publish-Subscribe pattern while on the other hand a function storage and retrieval interface for procedural communication realizes a form of local request-reply pattern.

Data items communicated via the observer pattern are represented in IceWing by a reference to the actual data and a stringified identifier. This allows to exchange any data without any restrictions and without the need of any preprocessing. However, *cooperative plugins* [LWHF06] are developed according to the information-driven paradigms introduced with the document model in Section 6.2 and

thus technically process XML documents with binary attachments. Plugins observe the storage of any number of such data items. If an observed data item is stored by a different plugin, the observing plugin gets called with the new data after the storing plugin has finished its work. The data itself is not copied during the complete process, but managed by reference counting, allowing a fast communication between plugins even for large images. The now running plugin may again store any number of new data items with equal or different identifiers and may thereby itself invoke other observing plugins. In contrast to the IDI system-level concepts the IceWing architecture does not provide a sophisticated observation model due to the stated goal of maximizing performance. However, filtering and transformation of data is easily possible as plugins may insert a transformed data item in the processing loop.

Besides the data driven observer pattern plugins can provide any number of C functions under different identifiers. After registration other plugins can retrieve and freely call these functions from within their processing steps.

While plugins are executed sequentially, they are free to start new threads and thus perform any calculations in parallel. At the same time more advanced and more dynamical distributed interaction patterns are possible utilizing the generic IDI infrastructure plugins.

Transparent Distribution and Integration with IDI Plugins A library of cooperative plugins not only pays off in reusability of plugin implementations but also facilitates smooth collaboration between vision researchers on the one hand and system integrators on the other hand.

This is achieved through a small set of generic infrastructure plugins that extend IceWing transparently by the different interaction patterns described in Section 7.2. With those plugins events can, e.g., be observed from or published to several different IceWing instances running on an arbitrary number of network nodes or any other service utilizing the IDI architecture.

Listener plugins such as the subscriber plugin only need be configured with the subscription that matches the corresponding informer and the identifier under which the imported data shall be made available to other locally registered IceWing plugins. Vice versa, informer plugins like the publisher plugin must be configured with the specific internal identifier for the data to be exported. Neither a special meta-compiler or data-description is necessary nor any implementation change of existing plugins is needed to make use of those plugins if the data to be exported conforms to the IDI document model.

Following this concept, vision subsystems can be quickly integrated into larger loosely coupled systems but locally executed in a closely coupled processing environment as well as transparently distributed for boosting overall performance if available processing resources are an issue.

Mosaicing as Exemplary Application A wide variety of plugins have been developed in the course of the different applications, ranging from fundamental plugins that allow reading of image streams from sources like multiple disk files, movie files, and various grabbers and cameras to filter plugins for, e.g., image smoothing, image cropping, or color conversion, etc. Besides these, a number of higher-level plugins for object- and action recognition as well as visual tracking were provided by partners in different research projects.

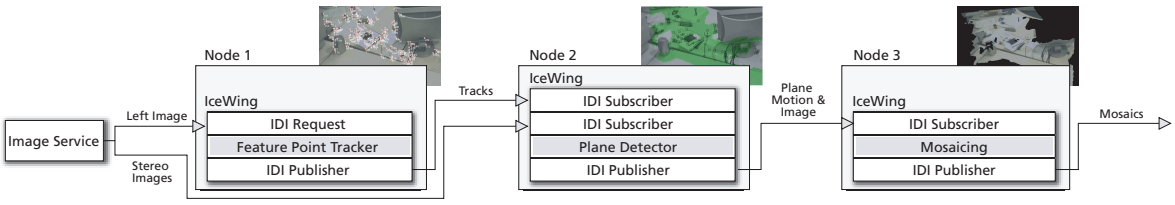


Figure 7.15.: *Exemplary IceWing application for generating mosaics from arbitrary stereo video camera sequences [GHC+04] using the presented IDI plugins. Besides providing the resulting mosaics to other components, the IDI architecture is used here for parallelizing processing to increase the overall performance [LWHF06] of this service.*

Figure 7.15 depicts how IceWing was used for a vision subsystem in the VAMPIRE project that generates mosaics from an arbitrarily moving stereo camera pair. This use case underlined the benefits of distributed processing and comprised individual algorithms developed by different partners in the project, which were integrated as cooperative IceWing plugins. Details of the underlying algorithms can be found in [GHC+04]. The architectural sketch of the three stage system is shown along with some results of each stage. The image service publishes captured stereo images from two cameras mounted to the VAMPIRE augmented reality device [SP04], which will be described in more detail in Chapter 8.

The *Feature Point Tracker* plugin detects feature points in the left image of the stereo camera pair request from the image service on demand and tracks these at about 15Hz frame rate. The *Plane Detector* plugin computes stereo matches from the tracked feature points and the corresponding right images. In terms of IceWing interactions, the plane detection observes tracking result data (via a corresponding subscription for tracking events) and referenced binary stereo images to compute new planes whenever the required data is pending.

As this correspondence matching can not be done at frame rate, parallel processing of tracking and plane detection is essential. Once planes are detected by stereo matching, tracking feature points indeed allows to track the individual planes. Finally, individual mosaics are asynchronously computed and published whenever new plane information becomes available.

Summarizing, IceWing itself is a development tool for a non-distributed low-level image processing which is highly extensible and already widely used within several research projects such as the VAMPIRE project. As up-to-date documentation and the software is freely available, IceWing facilitates real-time image processing on several platforms. In conjunction with the IDI infrastructure plugins, cooperative plugins are integrated transparently in larger system architectures utilizing the presented infrastructure plugins.

7.5.3. Application Specific Libraries

In contrast to modular and domain specific application adapters or type libraries, application specific libraries encode a complete domain model in a target language aimed at a particular scenario. Thereby it not only encodes the used data types but additionally encapsulates services behind a usually application specific programming interface.

An example of this class of available domain support is the *BonSAI (BirON Sensor Actuator Interface)* which is a high-high level Java API[2] encapsulating complex services running on the BIRON mobile robotics platform for training and teaching purposes. While all of these services are natively using the various functions of the presented integration architecture, BonSAI makes them available in domain specific primitives such as sensor and actuator abstractions on different competence levels. It provides a very robust and easy to use layer comprised by out of-the-box usable implementations of COGNIRON functions (CF) for simple implementation of robot behaviors, picking up concepts of traditional behavior-based robotics or behavior-oriented design. It thereby eases the implementation of sophisticated robot behaviors by young researchers and scientists even if they are unexperienced in robotics systems. The goal is to allow easy access to COGNIRON functions that enables these researchers to also more easily contribute their interdisciplinary experience and knowledge and quickly pour it into prototype systems that can actually be used for evaluation.

The BonSAI release has been successfully used and positively evaluated by interdisciplinary students of a winter school on human robot interaction[3].

To commence this section on domain support, let us recall the beginning. As no single domain model up to now exists for cognitive (interaction) systems, the main contribution towards domain functions is *usability* of the core models yielding in different models that are provided by users of the architecture. The infrastructure plugins for the Neo toolkit may serve as an example here. As these toolkits can then easily be integrated by others, the available domain support shall be steadily increasing.

7.6. Summary

While the previous chapter presented the core layer of the information-driven integration approach, this chapter introduced five additional models. Together they define the information-driven integration architecture that supports the actual development and software integration of experimental cognitive systems.

The resource model provides a vocabulary for conducting integration from a higher-level of abstraction. This is on the one hand beneficial for a logical problem decomposition; the introduced URI scheme also permits to assign responsibilities for clearly definable parts of a system architecture on different abstraction levels. Thereby, it implicitly supports a feature-driven integration approach as presented in Chapter 3. For the integration architecture itself, the naming model provides essential features for realizing higher-level interaction patterns.

The interaction model maps well known message exchange patterns from service-oriented and event-driven architectures into a consistent programming model that is solely based on the introduced information-driven core layer. Thus, it allows architects to use both functional and event-based decomposition for system design. The introduced adaptation patterns harmonize the expected interaction profile from the viewpoint of an individual component, e.g., in terms of synchronous or asynchronous programming models, thus promoting usability and flexibility of the resulting programming interface.

[2] see `https://code.ai.techfak.uni-bielefeld.de/bonsai` for documentation regarding the BonSAI API

[3] the COGNIRON winter school on human robot interaction (CWSHRI) took place at Lausanne, Jan. 21st to 25th, 2008

The memory model is a core feature of the information-driven integration architecture. Besides reusing the patterns of the event-based integration model, it extends the basic interaction patterns in many respects. It provides a simple but powerful model to coordinate distributed processes of a cognitive system inspired by the tuplespace concept. However, the semantics of memory operations are less strict and therefore more flexible. The active memory is a modular, event-driven service with clear communication and synchronization semantics. It permits subscribers to rely on the additional guarantees associated with memory events. For instance, an event is forwarded only after successful memorization, which in turn allows subsequent synchronized access to this element by other participants. In conjunction with the concept of synchronously executed intrinsic memory processes this provides an extensible and effective integration service. While in an event-based architecture participants are typically not synchronized, in the memory, immediate notification provides an additional form of synchronization for intrinsic processes.

From the perspective of event-based architectures, the active memory realizes a distributed event infrastructure using a separated multiple middleware approach [MC05]. The server part follows a multiple intermediate broker pattern, yielding potentially a number of memory servers that partition the overall event space into smaller fractions for which they act as central message brokers. Events are only forwarded to subscribers of the memory model if they are brokered via an instance of the active memory service.

Although state-of-the-art database technology is fundamental for the realization of this memory model, important conceptual differences to pure database systems can be stated. On the one hand, databases deal with events primarily internally instead of focussing on communication and coordination of loosely coupled distributed processes. Thus, the methods used in the database domain differ greatly from the concepts applied for distributed scenarios [CNF01]. On the other hand, the semantics attached with the externally visible interfaces differ from the operations defined on tuplespaces.

The persistence mechanism of the memory also bears some resemblance to a data-centered architecture like the blackboard. However, the memory itself does not need a central control component, as is usually assumed with blackboards [BMRS96]. Other differences are more important: Where in classical blackboards the topology is rather fixed, it is dynamic in active memory systems. Also, the memory uses the more scalable *signal/query* mechanism to pass data instead of direct access, which was one of the critiques of the naive implementation of the visual active memory concept. In blackboard architectures, data sharing for controlled components is emphasized, whereas in the memory the emphasis is put on notification between *independent* components and on persistent storage. The envisioned linear scalability of the active memory model by event space partitioning, detached notifications and the IMP virtual machine architecture provide an avenue for a scalable implementation. If state-of-the-art database technology and the principles of the event-based IDI approach are also considered, the objective of an efficient virtual shared memory architecture for cognitive systems is achieved.

Furthermore, within the different research projects where the active memory has been applied so far for integration and coordination of cognitive systems, some standard patterns emerged on the basis of the introduced memory model and its fundamental operations. Although many of those might be specific, a current research hypothesis [HS08a] is that some of those are applicable to a broader range of systems and scenarios.

However, many complex event-based architectures suffer from the fact that component coordination is implicit. To compensate for this, the coordination model introduces a standard method for dealing with the asynchronicity inherent to the information exchange in the IDI architecture, permitting the modeling of controlled arbitration in cognitive systems. The aim of this model is to permit a declarative and testable specification of the coordination strategies. Hence, an extension of Petri nets has been presented that integrates coherently with the IDI models. It not only permits to develop expressive coordination strategies but also yields a form of complex event processing component. Last but not least, an external coordination service eases the implementation of components, which permits the development of reusable generic services that are orchestrated in a scenario-specific manner by coordination models.

As the final contribution of the IDI architecture to the requirements identified in the first part of this thesis, the available types of domain support were explained. While XML type libraries are useful but not necessarily needed for integration in collaborative projects, the main contribution of this thesis in terms of domain support was to provide generic infrastructure plugins for an existing computer vision toolkit. While this toolkit can be used for the development of high-performance, real-time computer vision services, the infrastructure plugins resembles versatile application adapters that allow to integrate developed applications into larger systems.

To commence this section, let me briefly refer to the required aspects as introduced in Chapter 5. The IDI architecture largely fulfills most of the requirements, with a particular focus on functions for information management, distributedness and coordination. How these functionalities have been applied to research systems is the main focus of the following part of this dissertation.

Part III.

Experimental Evaluation

The main contribution that is presented in the third part of this thesis is the augmented reality assistance system that the author cooperatively developed in the VAMPIRE EU project.

The various other VAMPIRE systems developed prior to this served as an iterative testbed for the presented information-driven approach. Besides explaining the utility of taking “the human in the loop”, this chapter further argues for the usefulness of the presented approach for coordination and integration of experimental, distributed cognitive systems.

In order to underline the claim that information-driven integration is a more general concept and can be applied to different application scenarios, the second chapter in this part will briefly report on the application of the framework and the design concept in other research projects, particularly in the domain of cognitive robotics.

8. The VAMPIRE System

The vision of the VAMPIRE research project has been to develop systems capable of understanding what they see based on what they have previously memorized following the concept of a visual active memory as introduced in chapter 2. Throughout the project, a number of prototype systems were cooperatively constructed involving several partners in a geographically distributed and scenario-driven research process (cf., Chapter 3). This integration process contributed at the same time to the iterative improvement of the integration approach described in this thesis. All demonstration systems, e.g. the office assistant [BHWS04], combined several perceptual processes in a coherent and usable system. In addition to perceptual features, techniques for the retrieval and interactive learning of new visually perceivable artifacts have been prototyped in the VAMPIRE demonstration systems.

Those systems were primarily build along the lines of two distinct scenarios: sports video annotation [KCK07] and wearable AR-based assistance [SHWP07]. The focus of the following sections is on the assistance scenario as it served as the fundamental testcase for the introduced IDI approach. At the same time, the software design and integration of the services used in the final demonstration system within the VAMPIRE project is one of the main contributions of this dissertation (cf. [WHWS06, WWH06]). The developed assistance system and previous prototypes were demonstrated at various occasions, e.g., the EU IST Event 2004 in The Hague, NL as well as on other international research workshops and conferences such as the ICVS 2006 in New York City, USA.

The software architecture of the assistance system was constructed according to the concepts of the introduced information-driven integration approach and is based on the initial implementation of the corresponding software framework [WFBS04]. It combines the perceptual processes that were cooperatively developed by the various project partners as well as the ones that were created at Bielefeld University into a coherent architecture based on the principles of the different IDI models introduced in the foregoing chapters. The IDI approach and the underlying software framework provided an avenue for the efficient collaborative construction of a real-world visual active memory instance.

This chapter first introduces the scenario of the augmented-reality based context-aware assistance system. In order to manifest the concepts introduced so far, the subsequent section explains the implemented instance of a visual active memory focusing on the use of the information-driven integration models. Last but not least, some results of a user study will be discussed that was conducted based on the available integrated system which also allows to draw conclusions on the utility of the software architecture when applied in a systemic context.

8.1. Augmented-Reality for Context-Aware Assistance

The aim of mobile assistance technologies is to support users in performing complex tasks or provide them with additional information either previously learned by the system or dynamically retrieved from external knowledge sources.

← Hybrid tracking unit consisting of an inertial sensor (XSens MT9) and a custom CMOS camera for pose tracking.

← Custom stereo video see-through set combining two Fire-i firewire webcams and a head-mounted display (I-visor 4400VPD HMD). The webcams are also used for scene analysis as well as object and action recognition.

Figure 8.1.: *The assistance system: Hardware setup of the AR gear. It has been designed and assembled by Graz University of Technology [SP04].*

Information relevant for a particular situation must be made available to the user in a context dependent and unobtrusive manner. Future real world applications might include industrial assembly, remote teaching, multi-user collaboration and prosthetic memory devices for personal assistance. Prototypical questions answered by such assistants are for instance "*Where have I put my keys?*" or "*How do I construct this assembly?*". In the VAMPIRE mobile assistant scenario the user wears a mobile device that - by means of Augmented Reality (AR) - integrates him in the processing loop to close the perception-action cycle as explained in Section 2.2.1. Thereby, human-computer interaction and visual processing is tightly coupled and it is beneficial that system and user share the same view. By this means, the AR device's sensory equipment enables the system to take the perspective of an acting human as well as to provide feedback.

In order to exemplify these concepts and to demonstrate as well as evaluate the ideas of a visual active memory and the presented integration architecture, we considered an interactive scenario that is easily explained: *the VAMPIRE cocktail assistant*. Prior to the presentation of the resulting information-driven software architecture, the utilized AR platform serving as the hardware basis for integrating the human-in-the-loop and the desired functional properties of the cocktail assistant system will be introduced.

8.1.1. An Augmented-Reality Interface for Human-Machine Interaction

The first requirement for a mobile augmented reality (AR) device useful within the VAMPIRE application contexts is to *provide information* about its environment to the active memory components. As the system shall be able to assume the perspective of the human user, video images from his or her perspective need to be recorded and forwarded to other architectural components. Furthermore, as the system is not stationary, information about the direction of the user's view for (self-) localization and head pose recovery need to be made available, which can be achieved, e.g., by applying vision-based tracking methods or by using inertial tracking devices [SHWP07].

Last but not least, the system shall allow for natural language understanding, hence, a microphone is needed to facilitate speech recognition.

The second requirement for an interactive device is to *provide feedback* to the users of the system. This is usually done via a head mounted display (HMD) in mobile augmented reality. In such a scenario, the perceived or simulated environment together with additional textual or graphical information overlays is projected into the user's field of view. Additionally, auditive feedback can be provided to users via some sort of sound output in order to achieve multi-modal interaction. Through these functions, an AR-based human computer interface effects a rich bi-directional communication channel between man and machine.

Visualization is carried out on a laptop that features an OpenGL graphic chip (nVidia Quadro) with hardware supplied stereo graphics rendering for high-quality augmented reality. The custom stereo video see-through head mounted display (HMD) utilizes low cost, off-the-shelf components such as two Fire-i firewire webcams and an I-visor 4400VPD HMD. Additionally, attached earphones account for the desired non-distracting audio output.

Hybrid tracking is performed with a custom CMOS camera and an inertial tracker. This tracking subsystem is run on a custom mobile single board computer (SBC). Besides the SBC itself, the tracking unit includes a power supply (AC / DC) serving all the peripheral hardware of the mobile AR system such as HMD, firewire cameras, CMOS camera and inertial sensor. It utilizes a custom-made Fuga 1000 based CMOS camera ('i:nex') featuring an USB2 interface for extremely fast acquisition of small, arbitrarily positioned image portions typically used for tracking of corners or other local features with small support regions [SHWP07].

Figure 8.1 depicts the prototypical realization of such a hardware device as designed for the VAMPIRE project. It was made available to all project partners and has been usable even for unexperienced persons during the evaluation studies. This device as developed by TU Graz [SP04] consists of a visualization and a tracking subsystem.

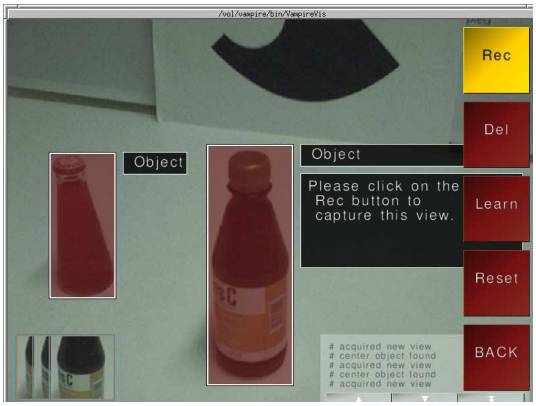
A wireless mouse may be used as an additional input device for controlling the system besides giving speech commands via the attached microphone. Laptop and SBC can be mounted on a backpack and connected via (W)LAN to the other parts of the system.

8.1.2. The Assistance Scenario

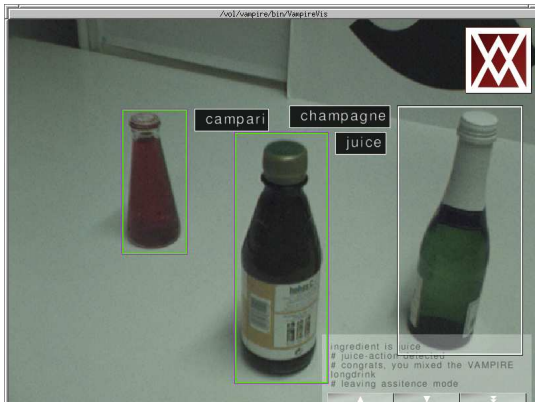
In the course of the project, we considered a scenario that aimed at the realization of an interaction loop providing context aware assistance to users carrying out everyday tasks in real-world environments.

In this scenario a user is sitting in front of a table and is wearing the AR-gear as introduced above. The user inspects or manipulates objects, e.g. beverages, cups or other rigid objects, which are placed on the table. Based on the multimodal interface of the AR-gear as outlined above, different capabilities were integrated that allowed us to design a number of assistance use cases.

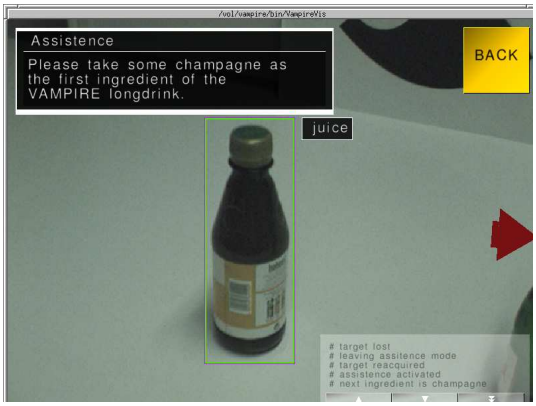
Exemplary high-level capabilities are object recognition and learning, visual tracking and action recognition as well as task models for the supervision of action sequences. Visualization capabilities were developed allowing the system not only to display instructions but also to highlight objects and even guide the user to referenced objects outside the current field of view through visual markers.



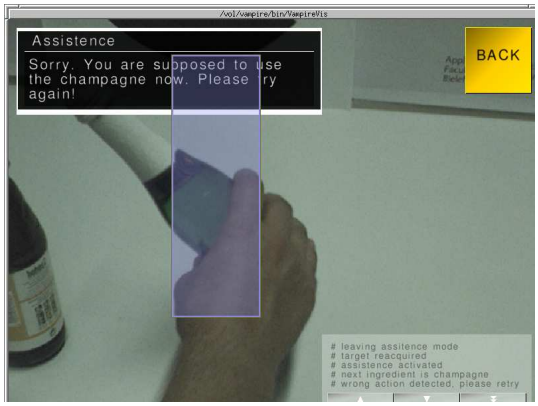
(a) Acquiring views for the training of new objects



(b) Displayed object recognition results



(c) Assistance: The system guides the user to the next ingredient (arrow on the right)



(d) Action recognition: The blue box indicates the tracked object.

Figure 8.2.: *Screenshots of the user's augmented view while performing prototypical use cases.*

The following use cases were realized that exemplify the utility of this visual active memory system:

- *Interactive Learning*: As shown in Figure 8.2(a) it is possible to teach the visual active memory new objects in order to achieve the desired flexibility of a cognitive vision system. To assist in the interactive learning procedure, snapshots of the recorded views subsequently used for training are displayed as augmentations that can be discarded by the user. Furthermore, only the object focussed by the user is considered in the capturing step.
- *Object Memorization*: The system is context- and situation-aware in its visualization and information presentation. Figure 8.2(b) shows a prototypical augmented view of a scene including detected objects on a table. Two objects were correctly classified and thus highlighted by a green box. Perceptions with a reliability below a certain threshold are not displayed to the user. If an object hypothesis is constantly reliable over a longer time interval it is considered as a stable hypothesis and memorized for later retrieval. Additionally, the displayed augmentations depend and vary according to the overall operational context of the system. For instance, no other object recognition results than the relevant ones are shown if step-wise instructions are presented to the user.

- *Object Retrieval*: Based on a computation of the three-dimensional position for recognized objects and the beneficial properties of taking the human-in-the-loop through augmented reality, the system is able to direct the user's attention in the scene and guide her or him to a referenced object that may reside outside the current field of view as done with the three-dimensional arrow-like augmentations in Figure 8.2(c). However, as a natural precondition the object that is searched for must have been previously memorized by the system. This serves as the basic interactive retrieval functionality of the visual active memory and allows to provide an answer to the questions raised in the beginning of this section.
- *Step-wise Assistance and Supervision*: The system provides context-aware assistance for simple object manipulation tasks, here the exemplary task is to mix beverages according to a given recipe. It guides the user to memorized ingredients, prompts for the necessary next step as shown in Figure 8.2(c) and observes and recognizes the actions carried out by the user, which is depicted in Figure 8.2(d). As a possibly complex recipe is broken down into a sequence of actions that are carried out with specific objects and as these are stored in corresponding task models, the system is able to dynamically check the correctness of an individual action and the involved objects.

On the one hand, the scenario highlights the purpose of a visual active memory. At the same time, it demonstrates the concepts of placing the human inside the processing loop of a vision system that would otherwise lack the necessary embodiment to modify the environment for its purposes, e.g. to acquire views from different perspectives in order to learn a new object. On the other hand, the immediate feedback on visual processing results is prevalent in this scenario making the ongoing system behavior transparent to the user and augmenting his or her reality with the results of an interactive retrieval process.

These are the basic functionalities our demonstration system needs to provide in order to qualify as an instance of a visual active memory. However, which components were used for the actual realization of this system and how all this is designed and integrated based on the concepts of the information-driven integration approach shall be explained in the following section.

8.2. An Information-Driven Software Architecture

The following sub-sections describe the architecture of the resulting context-aware assistance system from a functional viewpoint, a development perspective, a service oriented and a physical viewpoint. Subsequently, a number of interaction scenarios will be presented that combine different aspects of these views. This style of presentation is loosely inspired by the 4+1 model of software architecture introduced by Kruchten et. al [Kru95] but stays on a rather high level of abstraction in order to provide a good architectural overview.

The developed assistance system serves as a proof of concept for the integration architecture presented in this thesis. While the functional building blocks that allowed the realization of the different use cases for the context-aware assistance system were independently developed by the partners in the VAMPIRE project, they were incrementally integrated with the IDI approach. The overall system functionality results from the interplay of the different components managed by the coordination functions of the introduced architecture as will be explained in Section 8.2.5. Implicit and explicit coordination is thus an important raison d'être for the proposed information-driven integration approach.

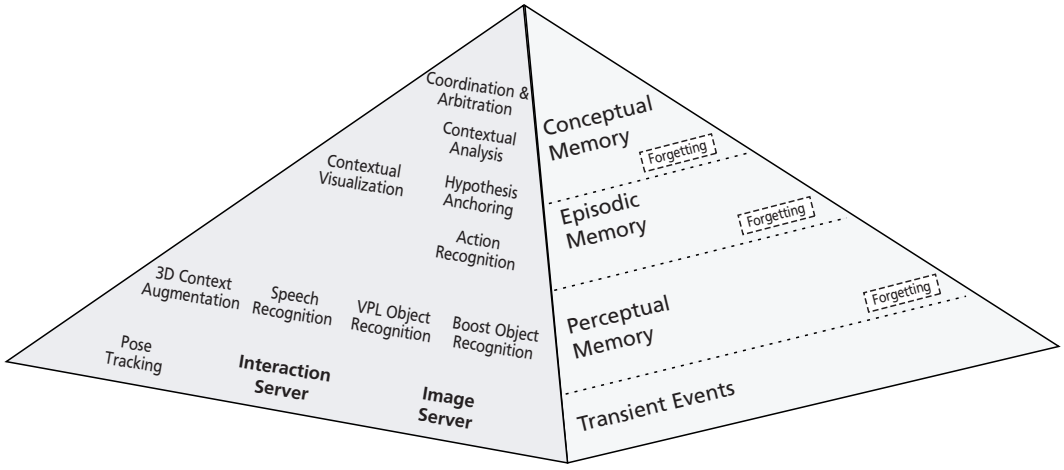


Figure 8.3.: *Illustration of the functional architecture of the VAMPIRE assistance system. Processes coarsely map to different layers of the visual active memory. Abstraction level of information is increasing from bottom to top layers and higher-level components potentially make use of lower-level information while this is not the case vice versa.*

However, before entering this discussion we shall have a closer look on the individual components that were integrated into the overall system.

8.2.1. Functional View

Figure 8.3 depicts the functional architecture of the assistance system. The depicted processes can be classified into object recognition and learning processes, the 3D vision and hybrid tracking subsystem, visualization and multimodal interaction processes, action recognition modules as well as hypothesis anchoring processes. All processes are making use of the functions of the IDI architecture, on this level in particular of the coordination and the memory models. While the former, e.g., manages the aforementioned task models used for supervision and assistance, the latter is the central method of integration as most of the processes interact via the active memory.

According to the conceptual layers of a visual active memory architecture as introduced in Section 2.2.2, the software architecture is structured similarly featuring three different active memory spaces and a transient event space for sensorial data as shown in Figure 8.3. Each of the memory spaces features an independent forgetting process with different parameterization. Hence, information is usually more durable in the conceptual and episodic layers than in the perceptual layer.

As the active memory and the intrinsic forgetting processes are fundamental features of the integration architecture and have been introduced in the previous chapters, they shall not be explained here once again. Contextual analysis is not explained here either as it has been discussed as an example for the coupling of IMPs and EMPs in Section 7.3. The functional architecture presented in Figure 8.3 permits the realization of all features important for the construction of a cognitive assistance system. Please note that each of these processes usually reflects an area of research for itself. However, in order to assess the capabilities of the integrated system, the most important building blocks for this scenario will be outlined briefly. For further information, the interested reader is referred to the corresponding publications that describe each approach in greater detail.

Object Recognition & Learning

Objects, e.g., ingredients for recipes, play a crucial role in the presented scenario of a cognitive assistant. The system needs to know which object is located where and which objects are manipulated by the user. The assistance system features two different methods with distinct characteristics for object recognition: one that is robust and domain-driven but needs to be trained in preface and an object recognition and learning component that allows fast online retraining of classifiers. The latter represents an appearance based object recognition [BBHR04] component that is motivated by biological information processing principles which are believed to underlie early visual processing in the brain. It is constituted by a two-step procedure consisting of segmentation and classification.

The first step is based on the integration of different saliency measures such as local entropy, symmetry and Harris' edge-corner-detection into an attention map. Based on this map, objects on the table are distinguished and segmented. Each segment is normalized in orientation and scale and a combination of vector quantization and local Principal Component Analysis (PCA) is applied to achieve a dimension reduction of the input data. The final classification decision is realized on the basis of Local Linear Maps.

These classifiers can be trained with only few (about five) different views of an object acquired interactively by the user. The training set is automatically extended by including rotated and scaled versions of the captured views. The VPL classification itself performs at real-time on recent computers, which makes the approach feasible in online reactive systems and particularly useful for the envisioned scenario [BBHR04].

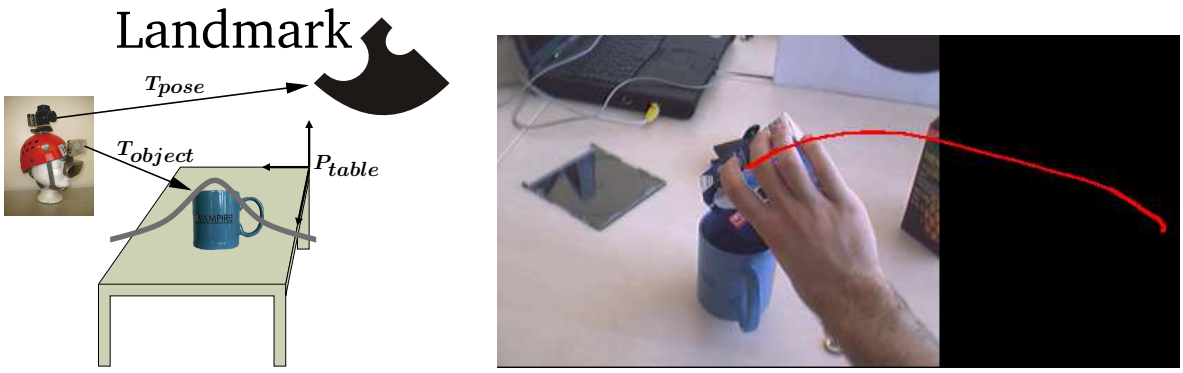
For robust domain-specific object detection, we integrated a well known cascaded weak classifier approach as introduced by Viola & Jones [VJ01]. In order to provide a basis for higher-level processes such as action recognition, several cascaded classifiers are pre-trained. Therefore, objects typically found in everyday environments would be recognizable upon starting the system. In long running cognitive systems featuring a visual active memory, cascaded classifiers could even be trained a posteriori, e.g., if the system is not being used and thus in an idle state.

As both object recognition components typically do not generate perfect recognition results, the visual active memory considers their outputs as hypothesis data as introduced earlier.

3D Vision Sub-System & Hybrid User Tracking

As explained in the scenario description the system needs to know the position of objects in the real world to guide the user and to be aware of the current situation. Since the environment is perceived only from (visual) sensors mounted to the AR gear their position with respect to the environment must be known. Accordingly, a component for user pose estimation and tracking is integrated into the system making use of the calibrated CMOS camera depicted in Figure 8.1.

The three-dimensional pose is computed from artificial landmarks [CSP03] as depicted in Figure 8.4(a). By knowing the precise location of at least four coplanar but not collinear reference points (or at least six arbitrary points) in the 3D environment and detecting their corresponding 2D image points, one can calculate the position and orientation of the camera. The applied targets provide excellent landmarks as they contain seven coplanar points defined by the corners of the target. To overcome deficits in visual tracking of these points, an inertial tracker aids the tracking process [RBP04].



(a) The position of objects on the table can be computed from the tracked 3D position of the user.

(b) Absolute trajectory of the manipulated object in a recognized "pouring" action. Image coordinates of previous frames are motion compensated.

Figure 8.4.: *Examples of 3D pose tracking and action recognition in the integrated system.*

By means of this pose estimation, the precise position and orientation of the user's T_{pose} is known (cf. 8.4(a)), allowing the system to determine where the user is looking. Hence, the 3D position T_{object} of objects located on the table (or any other known plane) can be computed by the intersection of the view ray determined by the object position in the image with the known table plane P_{table}, which allows the 3D context augmentation component to add estimated 3D locations to generated 2D object hypotheses.

Visualization & Multimodal Interaction

The AR gear realizes the interface between the user and the system. It can guide the user visually to certain places and provides feedback about the system's status and processing results by means of visualization, e.g., object recognition results as shown in Figure 8.2(b). Since the system and the user share the same view, the scene is really augmented by visual elements like semi-transparent rectangles, three dimensional arrows, etc. This allows for an efficient support of the user by the system, as for instance instructions can be directly displayed in the field of view and relevant places are visually referenced.

Furthermore, the system is able to receive input and requests from the user. The underlying software component is designed for multimodal interaction [Sch08] to navigate the GUI and control the system. It is decoupled through the active memory from arbitrary input and command sources. The scroll wheel of a wireless mouse can be used to choose buttons in menus and dialogs. Furthermore, speech input [Fin99] for labeling and more natural control, as well as head gestures [HBS05] to express confirm or decline in various communication situations are integrated. By this means, it enables interactive learning and labeling of objects, information retrieval and overall control of the system by the user.

An additional responsibility of the visualization and multimodal interaction layer is to connect the available sensors and feedback devices to other system components. In that sense, e.g., the image server is not a memory processes but a regular IDI process that sends image events as soon as a new frame has been captured from the AR gear's webcams to subscribed perceptual processes.

Hypothesis Anchoring

Components as, for instance, object recognition only provide instant percepts of the environment that describe the current visual appearance of the scene. These percepts are fed into the active memory and are transparently available to other components for further processing using information-driven integration principles. As indicated by Figure 8.5, the producer of information is not relevant for the anchoring implementation.

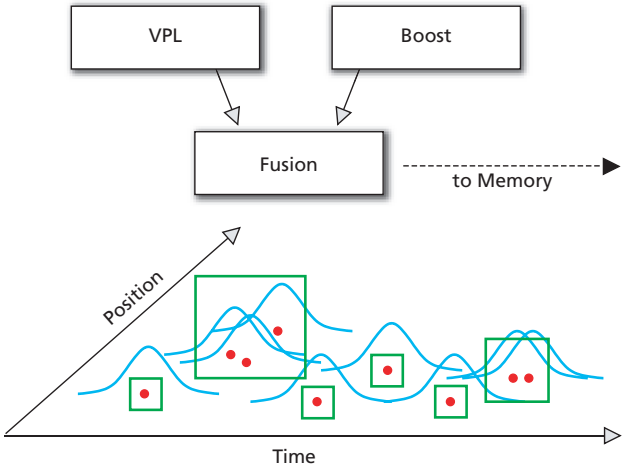


Figure 8.5.: *Anchoring maps percepts to reliable symbols in a memory space.*

Inspired by the work of Coradeschi and Saffiotti [CS01] a component called *hypothesis anchoring* has been developed which is described in [Han06] that maps these percepts to reliable symbols (anchors). This is essential for representing episodes over an extended period of time. For objects, anchoring compares the 2D or 3D position of a percept to assign it to existing anchored hypotheses. This position can be estimated based on a self-localization of the cameras as described previously. Object hypotheses are fused over time if the 3D positions are close enough to each other as illustrated in Figure 8.5. A Gaussian curve models the probability that two hypotheses refer to the same object (see the superimposed curve in Figure 8.4(a)). For the final classification result the labels provided by the object recognition component are integrated over a short period of time. If there is no anchored hypothesis that matches, a new one is created. Thus, hypotheses are anchored over time and a specific hypothesis gains increased reliability if many matching percepts support it. The reliability factor is included in the hypothesis representations in the active memory.

Action Recognition

As the system should not only guide the user but also supervise his actions, a component for action recognition is integrated. It has to answer the question whether the user has correctly performed the requested action or not. We utilize a classification approach based on the two dimensional trajectory of the manipulated object in the video sequence [FHS04]. It is trained with model trajectories of the respective actions and copes with variations of these by classifying them using a condensation algorithm.

Since objects cannot be reliably recognized by the object recognition component when being manipulated, visual object tracking [BGD05] is integrated to provide the trajectory of the object as input for the action recognition itself. Whenever an object is reliably recognized, visual tracking is initialized and follows the object. The robustness of the approach with regard to occlusion allows to track the object even when being manipulated. But since the scene is perceived by the head mounted cameras only, it is necessary to compensate for camera movement to compute the absolute trajectory of the manipulated object. Based on the fact that this object only covers a minor area of the field of view of every frame we compute the global motion of the image from the movement of tracked feature points in the background [ZGN04].

This visual background model permits to estimate the absolute trajectory by compensating the user's own movement. Figure 8.4(b) shows an estimated absolute trajectory of an object when performing a "*pouring*" action. Note, that the trajectory started on the right hand side in a part of the scene that is not visible anymore in the current frame.

Coordination and Arbitration

While the interplay of many of the presented component functionalities is based on the event-based implicit invocation patterns in the IDI architecture as explained later on, more complex behaviors of the system like those necessary for supervision and assistance need to be coordinated explicitly.

Due to this requirement, an arbitration component [WHWS06] was developed as part of this thesis that is based on the concepts of the coordination model as introduced in Section 7.4. Utilizing extended Petri nets [Pet81], the functions of the coordination model allowed to develop a component that controls the overall behavior of the context-aware assistance system. Its main responsibilities are:

- *Assistance models*: Representation of step-wise task models that consist of action sequences, parallel actions with or without involved objects and corresponding user interactions, which permits the system to guide and supervise user actions. For instance, it has to answer the question whether the user has performed the requested action in a specific task step with the correct object or not.
- *Interaction modeling*: Modeling of the interaction options that the system offers to its user in a particular situation. Petri-nets easily permit to set up context-dependent models that define the space of possible actions that may be invoked through any kind of event, with the additional possibility to transform low-level events sequences, e.g., into higher-level interaction events.
- *Exception handling*: Exceptions not only occur on a programming language level, but may also be raised due to hardware defects or unexpected user behavior. Thus, the system needs to react and adapt accordingly in order to continue its operation. For known exceptions, this component defines system level handling strategies to deal with these situations. Section 8.2.5 will give an example that models the recovery strategy for situations where the head pose of the user is not availabe.
- *Component control*: In order to embed components into a specific system context, they must be dynamically reconfigured according to the overall system state. This is an additional responsibility of the control component that achieves this either through the general service control interface or by specific means that are encoded in custom coordination model actions. An exemplary use of this feature is the reconfiguration of the visualization components in the assistance system, which provide differing visual feedback based on the interaction context, cf., Figures 8.2(a) and 8.2(b).

In order to execute the various system-level actions a number of additional generic and domain-specific actions were developed, e.g., for object learning, that perform necessary operations such as image transformations or the like and actually generate new events in the system. Extending the system behavior is easily possible with this approach as high-level petri-nets can make use of structured transactions that are itself executed in instances of control components. Concluding, the petri-net based control component allows for easy realization of different assistance scenarios.

8.2.2. Development View

The software development process for the context-aware demonstration system was carried out iteratively at different geographical locations distributed all over Europe with only a small number of intermediate technical meetings that were attended solely by researchers who were involved in the actual software development. Those meetings were used to discuss the overall architecture and find sustainable agreements, cf. Section 3.3. The declarativity and the high abstraction level of the service interfaces explained in the next section were quickly adopted by participants.

Besides using the AR-gear as a common hardware platform available to all participants, only the used operating system, which was Linux, and the IDI architecture itself were adopted as project-wide standards. With these exceptions, a number of different toolkits and libraries for signal processing introduced earlier such as IceWing, Matlab, Nessy or RAVL were used by the institutions. Due to this technical and organizational environment, the challenges described in chapter 3 had to be dealt with in the integration process in the VAMPIRE project.

In order to permit the use of the integration architecture on these platforms a number of specific application adapters - in addition to the IceWing infrastructure plugins [LWHF06] described in Section 7.5 - were developed as part of the integration effort. As other project participants interfaced these with further system components, the implementation and quick integration of the different services was facilitated, which will be explained next.

8.2.3. Service View

As introduced in chapter 7, the IDI architecture defines a service as a logical unit that represents a high-level functionality. Every service that can actually be used in a concrete system architecture consists of at least one component implementation that provides parts of or a complete functional service interface. In the assistance system, a component implementation features an external control interface handling system events like start, reconfigure, and stop requests. These services communicate via the introduced set of event-based communication mechanisms like publish-subscribe or memory-based interaction. Individual services must be designed such that they avoid assumptions about their collocation in the same process or processing node.

A service interface is characterized by consumed and provided event types, the patterns they use for interaction as provided by the interaction and memory models (cf. sections 7.2 and 7.3) and their explicit references to other system components. While a full specification of a service interface would need to state which schemas are used for event types and give exact descriptions of subscriptions, this information shall be omitted here for reasons of brevity. Instead, this section gives an overview of the different services which were instantiated in the assistance system's architecture as shown in Figure 8.6. However, some more detailed examples for certain use-cases will be given later on.

Naturally, a research system's software architecture must be broken down into smaller sets of independent services not only for functional but also for organizational reasons (cf. Chapter 3). During the development of the context-aware assistance system, a number of services were developed that can coarsely be classified in five groups according to their overall function. Each of these groups and their consituting processes will be shortly described subsequently.

Service	*Patterns*	*Roles*	*Provides*	*Observes*	*References*
Hybrid Tracking	PS	I	PoseEvents		None
Visualization and Interaction Service	RR, AM	I, L	VisEvents	PoseEvents, VisRequests	Hybrid Tracking Conceptual Memory
Image Services	PS, RR	I, L	ImageEvents	ImageRequests	Visualization and Interaction Service

Table 8.1.: *Service interfaces for the integrated visualization and sensing components. The abbreviations represent the different high-level interaction patterns and the component role as follows: PS = Publish-Subscribe, RR = Request-Reply, AM = Active Memory, I = Informer, L = Listener.*

Visualization and Sensing

Table 8.1 lists the three services that are directly related to sensing and visualization. Each of these services needs hardware access to the augmented reality device and its sensors. While this is obviously necessary, the three services thereby violate the stated rule that an individual service shall not make assumptions about its execution environment.

The hybrid tracking component sends information about the orientation of the user's head observing system components via a publish-subscribe pattern. It acts as an informer and does not reference any other system components. Due to the fact that the pose information is almost transient, it is not inserted in an active memory space but solely distributed as unreliable event notifications via multicast to other subscribed services like the 3D context service.

The visualization and interaction service (VIS) component realizes the multi-modal interaction as explained in the previous section. In order to receive events that need to be executed synchronously, it provides specific request handlers. Results of visualization requests are put in the conceptual memory and thus are persistently available for other system components, e.g., to analyze the interaction history of a user with the system. The VIS component references the conceptual memory but is otherwise not bound to the existence of the remaining system components. As an additional feature, it permits to grab regions of interest from the visualized video stream, which can be used for the asynchronous training of view-based object recognition services.

The most obvious component in a distributed cognitive vision system architecture is an image service that distributes grabbed images to other system processes. Utilizing the introduced publisher pattern, image events that solely contain metadata about the image itself are sent as unreliable messages via IP multicast to the perceptual services in the system. The image publisher is for efficiency and historical reasons co-located with the VIS component mainly because the latter performs the actual grabbing and places the current images in a shared memory buffer that is further used by both processes.

The image service features an additional request handler for on-demand retrieval of images in a short-term buffer as explained in the previous section returning preceding image events asynchronously to the calling component. If needed and if supported, the image service transforms grabbed images according to colorspace, resolution or region of interest specifications contained in the request events.

Service	*Patterns*	*Roles*	*Provides*	*Observes*	*References*
3D Context	PS, AM	I, L	ObjectEvents	PoseEvents, ObjectEvents	PerceptualMemory
Object Anchoring	AM	I, L	ObjectEvents	ObjectEvents (Anchored)	Perceptual Memory, Episodic Memory, Conceptual Memory
Contextual Analysis	AM	I, L	ObjectEvents, ActionEvents	ContextEvents, ObjectEvents	Perceptual Memory, Episodic Memory
Perceptual Memory	AM, PS	I, L	Perceptual MemoryEvents		None
Episodic Memory	AM, PS	I, L	Episodic MemoryEvents		None
Conceptual Memory	AM, PS	I, L	Conceptual MemoryEvents		None

Table 8.2.: *Service interfaces for information fusion and memory components.*

Information Fusion and Memory

The services in Table 8.2 represent actual instantions of the active memory components that realize the necessary partitioning of the overall space for memory elements and closely related functionality. The three active memory services are set up with individually parameterized memory processes reflecting the different semantic profile of the corresponding memory layers. For instance, *forgetting* discards in the perceptual memory all hypotheses older than 2 seconds without consideration of their reliability while in the conceptual memory only doubted hypotheses are discarded and no time-based forgetting at all is used. As for the active memory, the role of forgetting and intrinsic memory processes were already introduced in Section 7.3 as core concepts of the IDI approach and at the same time their interplay with the contextual analysis service was explained. Thus, I refer the interested reader to these pages for further information.

In addition to permit memorization, recalling of memory elements and the publishing of corresponding memory events through the active memory instances, a 3D context service is part of this group of services that is a rather scenario specific component. It enhances the object hypotheses inserted in the perceptual memory with their estimated 3D position based on the approach introduced in the previous section. From an interaction point of view, it subscribes to pose events as published by the hybrid tracking service in order to accomplish this task.

In contrast to the other services discussed so far, the hypothesis anchoring service is realized as an extrinsic memory processes that exclusively works on available memory spaces (and even as more general listener participant), not referencing any producers of the information that is to be anchored. It can thus be realized as a very generic process that only evaluates common metadata elements making no assumptions about specific event types, similar to the forgetting and contextual analysis processes, and can thus be applied in other memory-based systems, too.

The fact that this interface is not referencing any specific producer of events and just subscribes to the event types its implementation can process allows it to fuse information as sketched in Figure 8.5 without further configuration, regardless of whether, e.g., one or both object recognition services are available in the system or whether a different information source is providing information about objects.

Service	*Patterns*	*Roles*	*Provides*	*Observes*	*References*
Object Recognition (V&J Approach)	PS	I, L	ObjectEvents	ImageEvents	None
Object Recognition (VPL Classifier)	PS	I, L	ObjectEvents	ImageEvents	None
Online Learning of Objects (VPL)	RR	I,L	TrainEvents	TrainRequests	Object Recognition (VPL Classifier)
Action Recognition	AM, PS	I, L	ActionEvents, TrackingEvents	ImageEvents, ObjectEvents	Episodic Memory, Conceptual Memory
Speech Recognition	AM	I	PhraseEvents		Episodic Memory

Table 8.3.: *Service interfaces for recognition and learning components.*

In the present scenario it was mainly needed for tracking of otherwise independent percepts. In addition to that, it provides two other important functions: it improves the quality and stability of object recognition results and it judges whether an anchored object hypothesis shall be memorized in the conceptual space of the memory. The anchoring component subscribes for any modification or insertion of object events in the perceptual and episodic memories and assigns them to new or existing anchors that are managed in the episodic memory space. Anchored objects usually feature an improved reliability and are less transient than perceptual object events which is useful, e.g., to compensate scarce errors in an otherwise stable stream of object recognition results. If 3D information is available in an object hypothesis, its reliability is further increased.

This effects an improved visualization quality in the head-mounted display preventing, e.g., a flickering of augmentations. If an object hypothesis is highly reliable over a certain time interval and is thus considered to be correct, it is copied from the episodic memory and inserted into the conceptual memory for later interactive retrieval.

Recognition and Learning

For a cognitive vision system, services for recognition and learning as shown in Table 8.3 are essential to provide its functionality and consequently all of these service process incoming sensor data, either some kind of image or audio data. According to the image service's use of the publisher pattern for distribution of the captured live images, action and object recognition components subscribe to the corresponding image events like the object recognition service that was realized according to the weak-classifier concept.

In general, both object recognition components realize the same service interface, which allows other services to handle object events in a uniform manner. Regarding the VPL-based object recognition component, however, integration was slightly more complicated. In order to deal with the unnecessary high rate of new object hypotheses the underlying algorithm generates, which was much higher than the expected rate of change in the scene, we wanted to adapt its behavior concerning this matter.

As internal change would have resulted in a significant effort, we made use of the possibility to register message transforming functions in the outgoing router of the components object publisher as described in Section 6.5.

Service	Patterns	Roles	Provides	Observes	References
Context-Aware Visualization	AM, PS, RR	I, L	VisRequests	ObjectEvents, TrackingEvents	Interaction Service, Episodic Memory
CASA Control and Coordination	AM, RR	I, L	CmdRequests, VisRequests, TrainRequests	ObjectEvents, ActionEvents, VisEvents, TrainEvents	Episodic Memory, Conceptual Memory, Interaction Service, Object Learning

Table 8.4.: *Service interfaces for coordination and interaction components.*

This permitted us to modify its behavior without changing the underlying source code. Additionally, it was possible to register a transformation function that translated internally used coordinates into the globally used coordinate system. In addition to this, the learning capability of this component needed to be integrated by an additional wrapper as the necessary functionality was only available as a set of legacy shell scripts. Thus, a new high-level service interface was introduced that features an event handler for training requests. Observed events contain object metadata and the image patches to be used for subsequent training processes that are carried out asynchronously. Further request event handlers allow to control the operation of the legacy component via shared memory. The learning adapter references the VPL-based object recognition service not by means of the IDI architecture but is closely coupled via the filesystem and the shared memory regions with this component, yielding in a legacy service that violates the stated goals of not making assumptions about its execution environment.

The action recognition service interface is consituted by a publisher that provides events about tracked objects to other system components, which is, for instance, used to provide user feedback. It is based on active memory access as it subscribes to reliable object hypothesis available in the episodic memory space, which allows to trigger the actual action recognition process as will be explained later. Last but not least, the speech recognition service provides information about recognized phrases and submits these events to the episodic memory, which may, e.g., be observed by the interaction service that can be configured to scan, dependant on its own interaction context, the phrase events for matching commands allowing for verbal system control.

Interaction and Coordination

The services that belong to the final group of components used in the described instance of the assistance system are shown in Table 8.4. Their main responsibilities are to control the interaction with the user and the overall coordination of the system as well as to permit context-dependant augmentation and to provide visual feedback of the internal system state to the user.

The context-aware visualization service(s) observe object events in the episodic memory layer and tracking events generated by the publisher of the action recognition services. Events are visualized according to the overall system state. Visualization is carried out by asynchronously sending corresponding visualization requests to the interaction service component. While the components that implement this service interface are rather simple, they are essential for the usability of the overall system. Separating this functionality from other components allowed for parallel development and independent testing.

The CASA[1] control and coordination service as explained in the previous section is a component that coordinates and supervises the behavior of the system as well as the actions of the human user. Thus, it makes use of almost all of the available interaction patterns and observes many events that are exchanged via the conceptual and episodic memories. Controlling the interaction of the user with the system, it is closely coupled with the visualization service using its request-reply interface. It communicates changes in the Petri net encoded interaction state synchronously to this service and thus permits an online synchronization between the interaction service visualized state and the state of the corresponding part of the Petri net.

The control component also makes use of asynchronous requests for long-running actions such as object learning, which involves a query for recent snapshots in the episodic memory space and the triggering of the object learning adapter. This concurrent execution can directly be mapped to corresponding petri-net structures and may therefore be encoded in the coordination models.

Despite its semantic coupling to the higher-level layers of the VAMPIRE assistance system, it references only the two components explicitly that do not allow other interactions, which are interaction and learning. However, as requests to these components are sent using the IDI request-reply pattern, requests are dynamically constructed and therefore, there is no compile or sequence coupling that imposes specific startup ordering between these components for the control service to become operational.

Infrastructure Components

The service architecture of the assistance system features a number of additional services that permit an efficient development and seamless operation of the overall system such as distributed application logging, introspection and visualization of memory contents or the control of the process life-cycles. Despite their importance from a collaborative and technological perspective as motivated in Chapter 3, they are not going to be further explained here as they do not affect the core functionality of the assistance system. Thus, they are also not depicted in Figure 8.6 for reasons of brevity.

The introduced service interfaces provide an avenue for the realization of the functional architecture with regard to component interaction and coordination. The overall functionality of the resulting software architecture allows an efficient realization of the use cases to be handled by the assistance system. Not anticipating the conclusion, most of the described service interfaces based on the concepts of IDI approach yield loosely coupled services that could independently be developed and reused in different scenarios. In order to gain further insights about the software integration of the VAMPIRE system using the IDI architecture, the next section describes a chosen deployment situation to briefly underline the distribution capabilities of the developed integration architecture.

8.2.4. Physical View

Figure 8.6 depicts a deployment of our assistance system. It is running on four standard Linux PCs (Pentium 4, 2.4GHz, 512MB) connected via a switched Fast Ethernet network infrastructure, a visualization laptop and the tracking subsystem as explained in Section 8.1.1. Images are captured from the fire-I firewire cameras and distributed with a resolution of 320 x 200 pixels.

[1] CASA has been the working title for the assistance system.

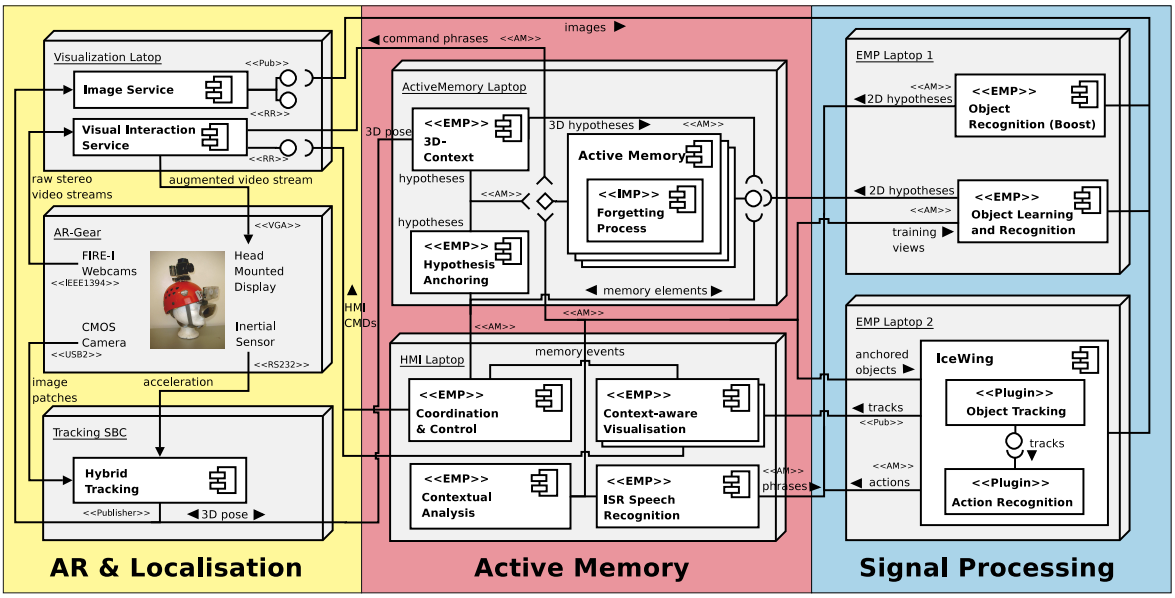


Figure 8.6.: *Architectural sketch of the cognitive assistant*

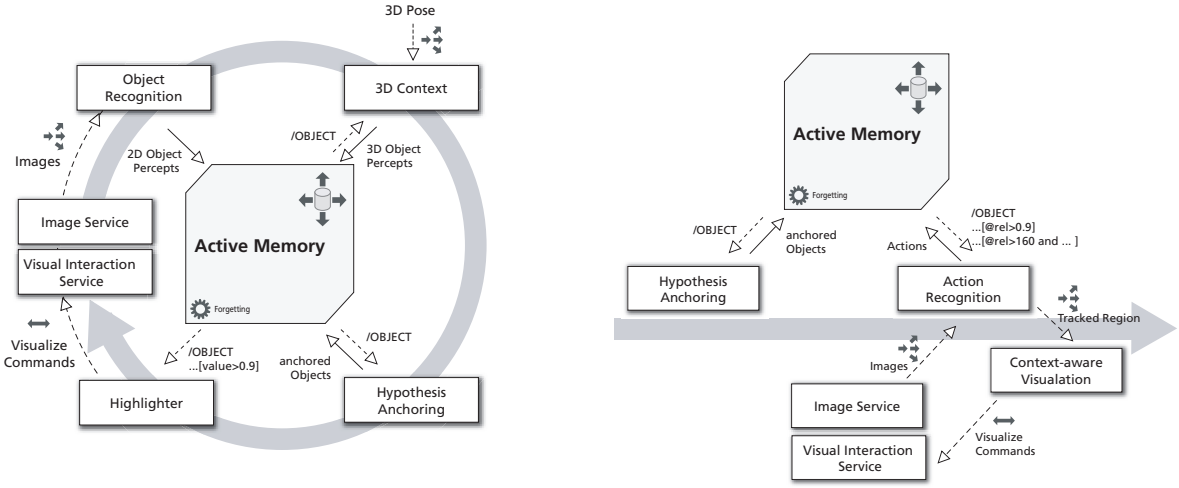
Services presented in the previous section that realize the distributed assistance system's software architecture run on each of these nodes. The components shown in the active memory and the signal processing partition can be distributed freely as they make no assumptions about co-located services and are not dependant on specific hardware.

However, this is naturally not true for all of the shown components. For instance, the hybrid tracking process runs on the single board computer due to the local access to the CMOS camera of the AR gear via USB2.0 and the RS232-based local interface to the inertial tracker. Similar to this component, the visual interaction service must be run on the visualization laptop for accessing the webcams via the local firewire interface and the availability of the specific graphics hardware, which in turn restricts the image service to be run co-located with this component as images are transferred between the two components via shared memory.

While this kind of hardware-induced execution coupling is sometimes inevitable, this already exemplifies how coupling limits flexibility, which for instance in this case lead to an unusable system if the visualization laptop was broken. Another example of such an execution coupling are the VPL-based object recognition and its learning adapter. If a system features many of these coupled components, its combined accidental complexity imposes problems on the development process, yields a brittle operation and will finally make it unusable and unattractive as a research platform over a longer period of time.

8.2.5. Interaction Scenarios

The functionality of the system certainly depends on the individual components, their correct composition as services and the physical deployment of the service components as well as their coherent development, but even more on their adequate and efficient interplay. In the presented assistance system, the implicit and explicit coordination features of the IDI approach are applied for the integra-



(a) Data loop from images to visualization.

(b) Triggering action recognition when a reliable object is in focus.

Figure 8.7.: *Information-driven integration in the cognitive assistant. The diagrams illustrate the logical flow of data in the specific cases. Much of the data flow is mediated through the active memory.*

tion of the different components. Three different rather simple usecases that are fundamental for the system shall exemplify how components are coordinated based on the concepts introduced with the information-driven integration approach with regard to this aspect in a distributed system architecture.

Augmenting the User's Perspective

Since a central idea of the information-driven integration architecture is to coordinate involved components by events, a usecase can be explained by analyzing the flow of these events in a system.

Figure 8.7(a) outlines the processing path of an object from perception to augmentation. It starts with image frames that are captured from the user's perspective and published as image events. As the object recognition component is subscribed to these events, their observation triggers its recognition algorithm. Detected objects itself yield new events and are inserted as 2D object hypotheses into an active memory space. The 3D context component is directly subscribed to the insertion of new 2D hypotheses which it extends with 3D information based on the current headpose of the user that is frequently updated by a corresponding listener registered for published pose events. The received 2D percept is updated and its memory element replaced.

Because the hypothesis anchoring component has subscribed itself on the insert or replace action carried out on such percepts it in turn gets triggered, matches the percept to anchored hypotheses and assigns a reliability factor. The hypothesis is then once more replaced in the active memory.

Continuing along its path, the hypothesis triggers the context-aware visualization component only if the hypothesis is reliable, since the user should not be bothered with unreliable information. In the memory concept this filtering is realized by registering the corresponding listener with a more restrictive XPath condition as in this example: `/OBJECT[RELIABILITY@value >= 0.9]`.

Thus, the data is already interpreted by the IDI architecture itself. Finally, the visualization sends an asynchronous request to the visualization service to display the anchored, reliable object hypothesis to the user.

By accepting and executing the visualization commands, the information is displayed to the user and by this means, it closes the interaction cycle. Note, that all of the described activities are carried out asynchronously, which, e.g., allows for continuous augmentations even if for a certain amount of time no new stable hypotheses are detected by the object recognition processes.

Outdated or unreliable hypotheses are discarded at regular intervals from the active memory through the forgetting processes as explained in Section 7.3.

Triggering Action Recognition

As a second case study, we consider the way action recognition is triggered. We follow the idea that a user usually focuses an object before starting to manipulate it. Therefore, the action recognition component registers itself on reliable (`/OBJECT[RELIABILITY@value >= 0.9]`) and centered (`...[@x>160 and @x<240]...`) hypotheses that are available in the active memory space.

Figure 8.7(b) illustrates the complete flow of data in this usecase. The action recognition component starts tracking the object in the video stream when its subscription gets triggered by its local observation model due to the availability (either insert or replace) of a suitable memory element. Publishing tracked regions to the context-aware visualization provides a visual feedback to the user and allows him to seize the system behavior. A recognized action is subsequently inserted into the memory and may trigger further processing steps.

Coordinating complex behaviors

Implicit notification for component coordination is often sufficient for control of individual components. To realize more complex context-dependent coordination of several components running in parallel, the CASA control and coordination component uses the features of the coordination model.

To exemplify this, Figure 8.8 shows a small module of our high-level petri-net that models an exemplary part of system behavior: The handling of self-localization errors of the 3D vision subsystem. When the user is mixing a beverage, the system guides him with arrows to the next ingredient as shown in Figure 8.2(c). For this task, a correct 3D-pose is necessary. If it gets lost, e.g., due to occlusion of the landmark, the system copes with this situation and reconfigures several system components, e.g., the 3D guide widget in the visualization server. In particular, it instructs the user explicitly to re-focus the target. When the pose is available again, the system resumes normal operation.

Figure 8.8(a) shows the system working when the pose is available and the 3D object guide is activated. The event listener associated with the guard of the transition's input arc TargetLost is triggered in this state, if the 3D context module has inserted information about an illegal pose in the specified memory instance. Thus, the transition fires, which leads to a reconfiguration of the system components and petri-net model state as shown in Figure 8.8(b). Consequently, the transition StopGuide is now fireable.

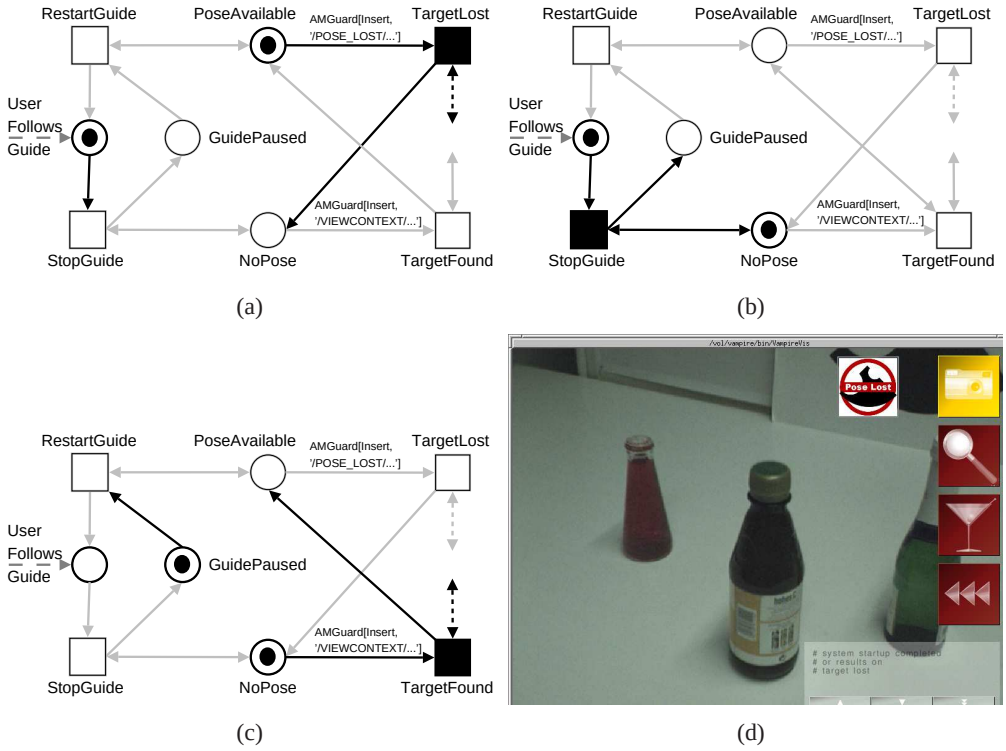


Figure 8.8.: *Active Petri net transitions when 3D pose is lost during object guidance and resulting system feedback. Rectangles depict transitions, circles places and filled circles tokens in places. Relevant model elements of each step are drawn in bold face. Existing guard specifications are annotated at corresponding input arcs.*

After this transition fires, the guide is paused, which is directly reflected in the model as illustrated in Figure 8.8(c). The system now waits for reacquisition of the 3D pose and in case one is inserted, TargetFound and RestartGuide would be fired and their set of actions be executed. This change would result in the original marking as shown in 8.8(a).

As described in Section 7.4, a sequence of actions is executed when a transition fires. To give an example, a dynamically constructed request event is attached to the TargetLost transition to deactivate the 3D object guide on the interaction service component.

While these usecases shall have underlined the suitability of the proposed models for coordination and integration of a distributed cognitive vision system, the following section considers the evaluation of the introduced system with user studies, which implicitly also evaluates the proposed system architecture.

8.3. System Evaluation

Evaluating the presented integrative system includes very different aspects. On the one hand, the applicability of the realized integration infrastructure in terms of performance has to be evaluated. As the IDI patterns play a central role in the assistance system's architecture, criteria such as access performance are fundamental to ensure the reactivity of the proposed system. On the other hand, these numbers are generally of little use as a meaningful evaluation needs to be carried out in system context. Thus, we shall put our emphasis on the latter type of evaluation. Even so, the subsequent section starts by discussing central product utility aspects with regard to typical performance considerations.

8.3.1. Performance Considerations

From a user's perspective, the main programming interface (besides the use of XML tools) of the IDI architecture are the different patterns that are provided by the interaction and the memory model. Besides ease of use, performance is another important factor for usability. Thus, the following paragraphs briefly report selected performance considerations to demonstrate that the chosen XML document model in conjunction with the interaction patterns and the memory model are fast enough to allow for the integration of a reactive cognitive vision systems such as the VAMPIRE assistance system.

Interaction Patterns

The performance of the basic interaction patterns provides a first rough estimate for the utility of the architecture in the given context. The version of the IDI architecture that has been used in the VAMPIRE project features a port implementation that is based on Ice [Zer06]. As this is an operational middleware with strong support for network-wide object references, patterns involving identity information are performing extremely well. For instance, the latency of a request-reply interaction carried out in C++, sending an object hypothesis as shown in Listing 6.1 on a 100MBit ethernet amounts approximately to 0.2$\sim$0.5ms [WFBS04]. Optional schema validation takes $\sim$1ms for typical object hypotheses as shown in Listing 6.1 in Section 6.3.

Evaluation of the native datatype transmission showed that performance is also sufficient to publish image data to a limited number of subscribers. However, experiments with this implementation yielded that if the number of subscribers is raised, the overall performance drops quickly. This is due to the fact that in 2004 the Ice-based solution did not support multicast and thus each subscriber received an individual copy of the event.

This was one of the technical reasons that an additional port implementation on the basis of the Spread Group Communication Toolkit was carried out, cf. section 6.5. As Spread supports network-level multicast, e.g., image data can be very efficiently communicated to a potentially large number of listeners.

On the basis of the Spread-port implementation, publish-subscribe using multicast communication can be very efficiently realized as shown in Figure 8.9. The screenshot shows the results of applying the Netbeans[2] profiler to a simple publisher application.

[2] see also `http://profiler.netbeans.org`

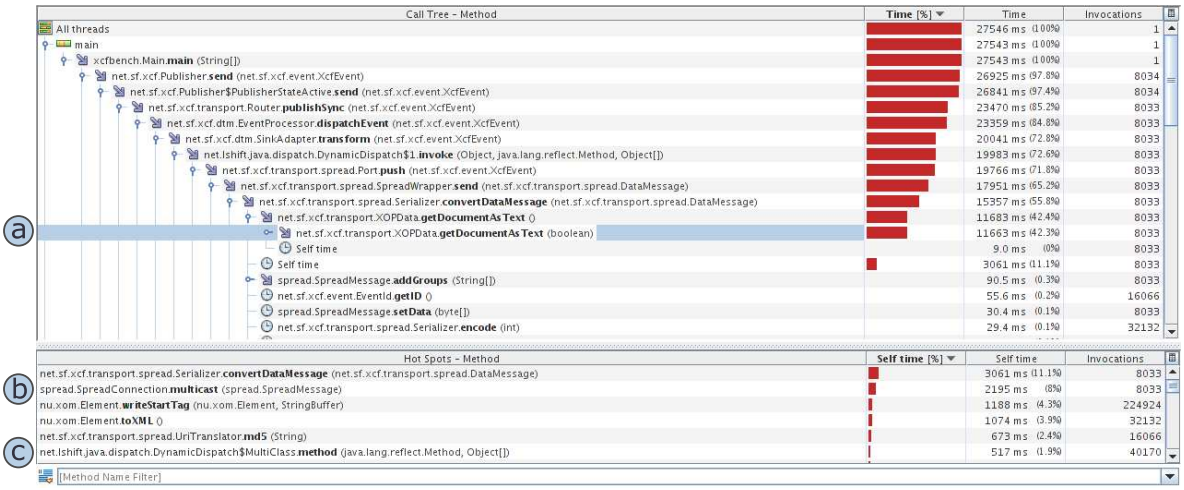


Figure 8.9.: *Runtime profile of the current Java implementation using a Spread-based Port implementation.*

While the quantitative results of the profiler obviously include measurement overhead and are thus quantitatively too high, some insights can be gained by analzing the shown behavior qualitatively:

- The XML serialization that converts the Document Object Model into a byte array consumes a significant amount of time as indicated in the call tree by line a in Figure 8.9.
- A single synchronous multicast of a reliable message with an already serialized XML document of 1KB size and a 10KB attachments takes only about 1ms to send (line b).
- The dynamic dispatch approach, cf. section 6.4, takes only a negligible amount of time (1.9%) that is very well acceptable given the increase in usability (line c).

Profiling the subscriber, it appeared that while each individual subscriber is able to receive messages in full speed with a cycle time of ~ 0.8ms, the limiting factors to perform load tests with an increased number of subscribers is supposed to be the local loopback interface of the receiving machine and the consumed CPU time. However, as it seems rather improbable that individual processes exchange messages at the rate of several kHz, this overhead is still acceptable. From a system-level perspective, the combined number of interactions is very well expected to exploit full network bandwith.

Memory Model

While features for XML processing are widely supported in recent databases, it still has to be ensured that information processing within the memory model allows for the necessary reactivity of a real-world cognitive system. Therefore, we conducted a performance analysis of the XML-based repository. Of foremost interest was the question how query performance scales with larger datasets which are to be expected in cognitive vision systems.

Our evaluation method resembles the application independent Michigan micro-benchmark procedure for XML databases [RPJ+03] but was adapted to our own dataset consisting of memory elements similar to the standard example as introduced in Section 6.2.

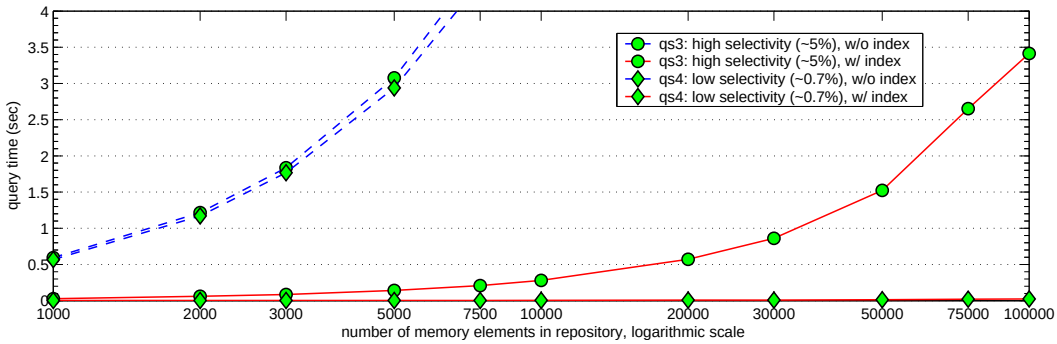


Figure 8.10.: *Query performance of the used DBXML database backend for memory elements.*

Figure 8.10 exemplarily depicts the mean performance of an attribute equality query such as `/OBJECT/REGION/RECTANGLE/COORDS[@w=225]/@w` (*qs3*) that will return a set of XML nodes matching the given condition, described by an XPath statement. Apart from the size of the dataset the size of the result set is also of interest. A typical query might return less than one percent of the whole dataset (*low selectivity*), no query is expected to exceed a result set size of five percent (*high selectivity*).

Looking at indexed and non-indexed queries, the latter ones are very expensive in terms of time. Also, in that case, selectivity of a query is irrelevant as disk I/O for a sequential scan of the repository seems to be the limiting factor. In contrast, indexed queries with low selectivity show almost constantly excellent performance regardless of repository size. Even better, the performance of the indexed *qs3* query with high selectivity is also sufficient for our application as it takes e.g. $\sim$0.57 seconds to retrieve about 1000 XML hypotheses from a repository with 20000 memory elements (see Figure 8.10).

However, as this are results collected in-process, the question remains how well this is supported on the system level. Table 8.5 suggests that these remote operations obviously come with a certain overhead due to network transmission of event payload. Fortunately, the fact that in the functional architectures of VAMPIRE and COGNIRON rather small documents are exchanged definitely facilitates the reactivity of this software architecture.

In addition to the performance of the memory interface operations, an evaluation of the latency between the initiation of a memory action such as an insert and the retrieval of the memory event has been carried out. For instance, in Java, this latency amounts for different memory events to ~ 1.5ms that must be added to the duration of the corresponding operation involved.

Document Size	*Java*	*C++*
1 KB	3.825 ms	1.322 ms
10 KB	16.798 ms	7.528 ms
100 KB	164.324 ms	64.66 ms

Table 8.5.: *Active memory insert performance on a 100MBit network (Ice).*

These and other results frequently gathered during practical integration underline that the use of XML-encoded memory elements with binary attachments, the multicast-based event distribution and the ability of indexing the underlying database provide a fast and reliable basis for the information processing in the IDI architecture. However, as such an evaluation is only partially useful and within VAMPIRE the user's are part of the processing loop of the system, they provide an implicit but more meaningful evaluation of the concepts, how well this approach works in a real-world context.

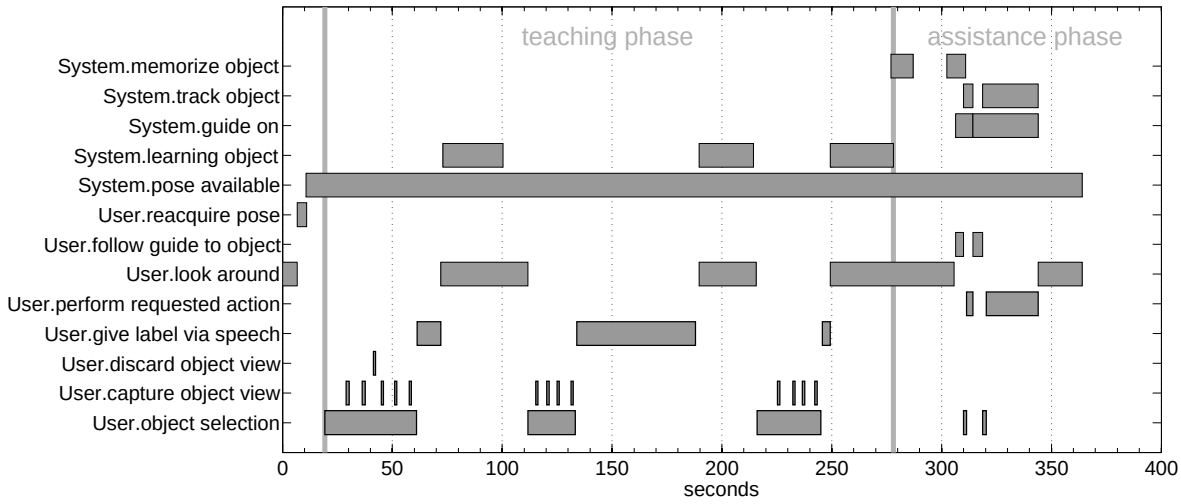


Figure 8.11.: *Annotation of user actions and system activities taken from one of the user study sessions conducted for evaluation.*

8.3.2. User Studies

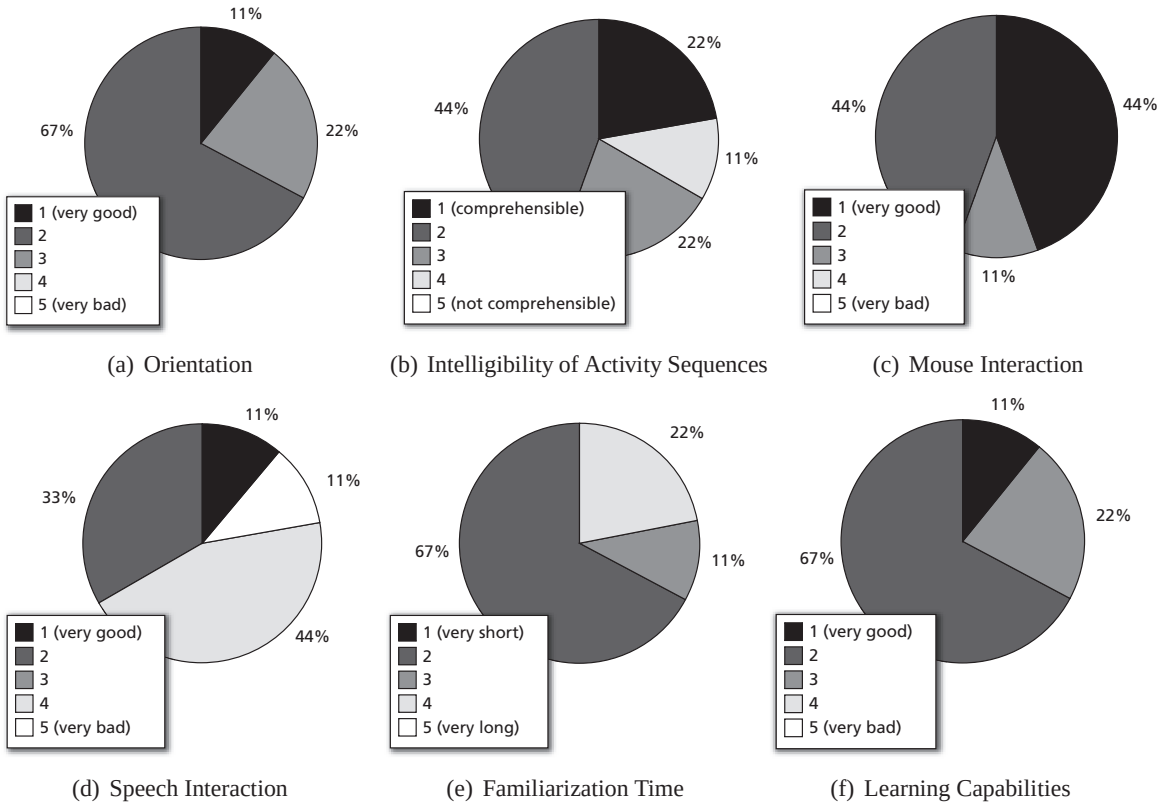
The availability of the integrated system allowed us to perform comprehensive user experiments to gather insights about how humans collaborate with such systems in the context of scenarios and dedicated tasks.

First of all, this permits to qualitatively assess the approach of a cognitive assistant and furthermore provides valuable hints for future development not only of the system itself but also of the underlying integration architecture as its utility is evaluated in a real-world systemic context. This is particularly important as the human user is part of the processing loop of the system and thus he or she can clearly judge the performance of the overall system. In the following, results of a user study are presented, which evaluates certain aspects of the described cognitive assistant system.

In contrast to performance figures about the framework or individual components, we focused this study on the question whether the human-in-the-loop paradigm is beneficial for users and which implications it induces on HCI and augmented reality assistance systems. Thus, our evaluation of the system in the assistance scenario covers important non-functional aspects like usability, comfortability, and reactivity of such a system as well as the provided functionality for user-assistance itself.

A total of eleven computer-literate subjects who had never before used an augmented reality system attended in this series of our evaluation study. To guarantee equivalent knowledge about usage of the system, a short instruction video was shown to each participant, which explained basic interaction primitives, e.g., how to present an object to the system for learning.

The task the users had to carry out was two-fold: Firstly, they had to train two previously unknown ingredients like orange juice or champagne to the system, which involved both labeling of the objects by speech and additional system interaction by using the mouse wheel. Secondly, they had to follow system commands in an assistance mode without prior instructions in order to mix a specific cocktail. This step has been carried out twice with different recipes to evaluate the familiarization of the user with the assistance system. During the experiments all user interaction has been recorded by video



(a) Orientation (b) Intelligibility of Activity Sequences (c) Mouse Interaction

(d) Speech Interaction (e) Familiarization Time (f) Learning Capabilities

Figure 8.12.: *Selected results of overall system evaluation*

cameras for later analysis. Afterwards, the subjects were asked to fill out a questionnaire yielding both quantitative and qualitative results regarding the presented system, which can be found in the appendix of this thesis. Faced with questions like “How would you rate the overall collaboration with the system?” they ranked their assessments on a scale from 1 (very good) to 5 (poor).

Our basic goal during this evaluation has been to prove whether the concept of the human-in-the-loop is suitable for users of the presented cognitive assistant system and whether the overall performance of the system is sufficient. As a basic pre-requisite to answer this question, we asked the subjects several questions about the AR gear prototype from an ergonomic perspective. Not surprisingly, most users experienced the bulky hardware device itself in its prototype state as heavy and uncomfortable but were able to complete the requested task with it. Besides that, the participants had a good overall orientation when looking through the HMD at the augmented video stream as shown in Figure 8.12(a). Most comments we received indicate that the chosen semi-transparent overlay with additional information is usually convenient for users of the system.

The hypothesis that quality and richness of human-computer interaction in the cognitive assistance scenario can significantly benefit from the concept of the human-in-the-loop has been verified with good results in our study. The evaluation underlined our personal experiences that system feedback is of highest importance to achieve this result, e.g., because the user needs to be informed about how he can collaborate with the system in order to accomplish a specific goal.

Within the evaluated scenario, an example for the assessment of system feedback are the ratings for the clarity of action sequences carried out by the subjects to perform a specific task, which are rather positive as shown in Figure 8.12(b). Furthermore, we received positive comments for the clear indication of error states, e.g., when the 3D pose is lost. This is shown in Figure 8.8(d) in the upper-right corner of the augmented image. A different example where we can improve the system feedback is e.g. for the tracking initialization. As shown in Figure 8.7(b) no additional information is given to the user that he or she might now start interaction with the object. An improved solution would be to indicate this clearly as it is done during object learning, see Figure 8.2(a).

An interesting finding of our study is the correlation of the type of system feedback, system reactivity and user adaptation. During the execution of a single step in the action sequence, e.g., pouring an ingredient into a cup, the visualized tracking highlight, see Figure 8.7(b), has not been able to follow the modified object in real-time. As a consequence, 77% of the subjects reduced the speed of their motions during the action sequences, thus adapting to the speed of the system, which in turn lead to a lower recognition rate due to the fact that the action recognition classifier was trained with faster motions. Nevertheless, comments from the subjects show that the overall speed of the system with regard to object and speech recognition as well as visualization was sufficient for a seamless interaction.

While the ability of the system to present its internal state to the human is important for productive modalities, the question of which perceptive modalities to use for interaction is equally relevant for the overall collaboration with the system. To that effect, we allowed the users to interact via the mouse wheel and a speaker independent speech recognition. Figures 8.12(c) and 8.12(d) show the individual interaction quality as reported by our subjects. The results for mouse-wheel interaction indicate that this interaction primitive seems convenient for most people. The familiarity of mouse-based interaction and the fact that the subjects were able to further concentrate on the given task by only using the scroll wheel for interaction have been reported as main reasons for this assessment. As all of the subjects were non-native English speakers, the results for speech recognition quality vary greatly since our speech recognizer has been trained on the American English Wall-Street Journal corpus.

The marks for the overall usability of the system have been rather good as 44% of the subjects rated it as moderate while the majority of 56% of the participants stated that the system has been easy to use. This is underlined by a steep learning curve as the familiarization time shown in Figure 8.12(e) indicates and additionally confirmed as all of the subjects needed significantly less time for the training of the second object and the preparation of the second cocktail.

Finally, all of the subjects involved in the study managed the given tasks and really liked to play around with the system. From subjects comments and answers, it can be concluded that the ability of the system to learn about its environment, see Figure 8.12(f), and the direct feedback resulted in a high motivation of the participants during the experiments. The ability to interact and collaborate with the system has been directly exploited by the subjects, e.g., to separate objects spatially in order to get better detection results. In our opinion all these observations underline the fact that the idea of the human-in-the-loop is suitable and useful for vision-based HCI and assistance systems. Furthermore, memorization and learning were appealing for users and the performance of the overall system was good and not distracting people when looking around in the scene or gazing at objects with the exception of the above mentioned action tracking visualization.

8.4. Conclusion

The aim of this chapter was to underline the applicability and utility of the introduced approach by explaining the software architecture of the context-aware assistance system that was developed collaboratively with the partners in the VAMPIRE EU project.

While the evaluation with naive users that was reported in the previous section showed that the system fulfilled the envisioned usecases, it showed on the other hand that the software architecture of the overall system was able to perform fast enough for running a reactive system for an augmented reality scenario, thereby proving its utility in a systemic context.

In addition to that, the explanations on the developed information-driven software architecture and the introduced service interfaces for the functional components within the VAMPIRE assistance system prove that the applied IDI approach yields - among others - the following beneficial characteristics when applied to research systems engineering:

- *Modularity*: The IDI approach clearly supports a modular development of software services for cognitive systems in heterogeneous research environments on the pattern, information and service level.
- *Understandability*: The resulting integrated software architectures are with regard to their coordination characteristics easy to understand and comprehensible as implicit invocation is applied for simpler situations whereas petri-nets allow for the modelling of complex integration scenarios.
- *Parallelism*: To allow for simple distributed, parallel operation was one of the basic motivations behind the development of this approach. Thus, the inherent parallelism in cognitive systems is well supported through the various integration patterns.
- *Asynchronous operation*: With parallelism inevitably comes asynchronicity – at least if one wants to exploit the benefits of the former. Thus, asynchronous communication is supported in all models of the information-driven integration architecture.
- *Low coupling*: The resulting software architecture and the underlying integration approach focus on promoting loose coupling between components whenever possible. This implies referential decoupling, distribution and temporal decoupling. To support this concept, services shall make no assumptions about their execution environment or bootstrapping sequence.
- *Improved testability*: Modular testing is possible based on the event profiles of service interfaces and the possibility to record and replay document-based event notifications.

While these benefits already suggest that many of the required aspects are met by the introduced approach, the next chapter shall shortly describe how the IDI approach has been applied in the domain of cognitive robotics.

9. Interactive Cognitive Robots - A New Domain

In recent years an increased interest arised on building personal robots that are capable of a human-like interaction. In addition to multi-modal interaction skills, robots must also be able to adapt to unknown environments as they are recently moving out of restricted lab environments into less constrained human environments. Therefore, a robot has to be capable of knowledge acquisition through embodied perception in a lifelong learning process. Furthermore, reactive control of the robot's hardware is important as humans are around.

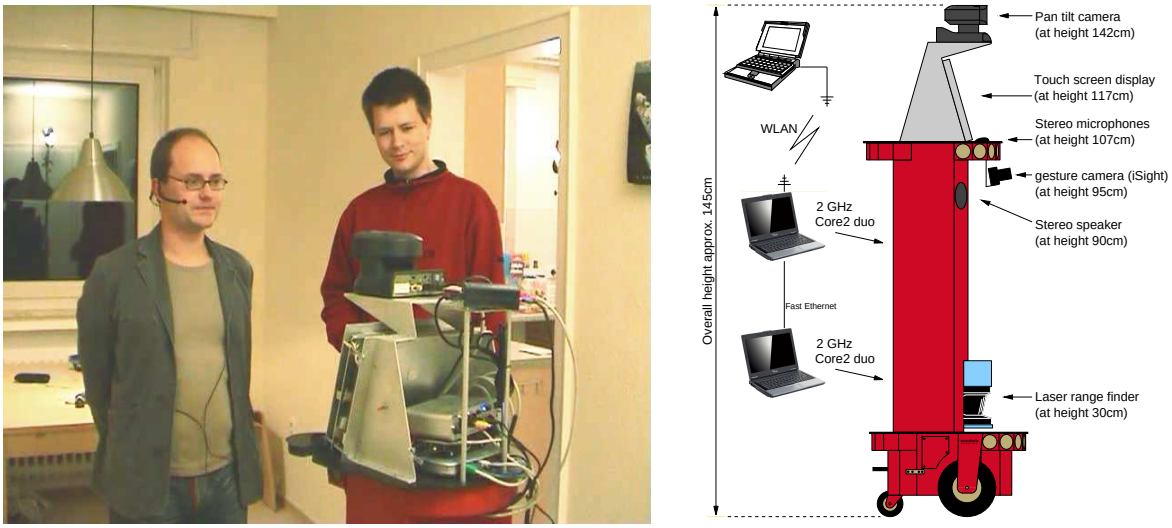
Consequently, researchers aiming to realize a personal robot have to integrate a variety of features considering aspects such as social rules, interaction design and usability factors. Hence, interactive robotics research is a truly interdisciplinary challenge and matches well with the stated aims for the introduced integration architecture, cf. Chapter 3. Naturally, the question arises whether the proposed approach can also be applied in this new domain.

Not anticipating the conclusion, this question has indeed been answered positively [FW07] as the integration architecture has been applied to a number of different research projects on robotics. In the following, the BIRON mobile robot companion developed cooperatively under the involvement of seven international research teams in the context of Key Experiment 1 of the COGNIRON EU project [Cog06] is shortly explained. It shall serve as the primary example for the utility of the integration architecture in this domain. As the BIRON robot and its software architecture are a truly collaborative system's project and its detailed description very well beyond the scope of this text, a particular focus will be set on a recent extension of the robot's capabilities.

Instead of discussing BIRON's interaction capabilities, involved algorithms, and the system architecture comprehensively, a case study of a face memory [HWLS08] for improved human-robot-interaction is presented as the main contribution of this chapter. It vividly illustrates how concepts of an active memory and the information-driven architecture are applied to collaborative robotics research. In addition to this, further applications of the presented approach in collaborative robotics research will be briefly presented with a short summary.

9.1. The Cognitive Robot Companion

The robot BIRON (BIelefeld Robot companiON, see Figure 9.1) is equipped with several sensors that allow an assessment of the current situation as a basis for interaction. Recent versions of BIRON feature already quite impressive interaction capabilities [LHW+05]. For instance, it is able to pay attention to different persons and engage in a one-to-one interaction with one user as illustrated in Figure 9.1(a) if this user greets the robot by saying "Hello Robot". From this point on, the robot focuses on this communication partner and engages in a dialog with him.



(a) Two persons interacting with BIRON

(b) Hardware

Figure 9.1.: *The BIRON robot companion engaged in social interaction in its hometour environment and sketched from a technological viewpoint. Its hardware platform is a Pioneer PeopleBot from Mobilerobots Inc. with two on-board laptops for control of actuators and on-board sensors as well as for sound and image processing. A third external laptop is used for speech processing and dialog control linked via WLAN. A pan-tilt-zoom color camera (Sony EVI-D31) is mounted on top of the robot at a height of 141 cm for acquiring images of the upper body part of humans interacting with the robot. Two AKG far-field microphones are mounted right below the touch screen display. A SICK laser range finder is attached to the front. As additional interactive device a 12" touch screen is provided on the robot.*

BIRON features extensive speech processing capabilities that allow it to understand instructions, questions, and statements in a flexible manner [LW07]. For example, the command "Follow me" results in the robot following the human around. The user can teach new objects to the robot by pointing at them while giving additional information. For example, giving an instruction like "This <gesture> is my blue cup" enables the robot to focus its attention on the referenced object and acquire an image of it for later recognition. Components for localization and navigation enable the user to teach the robot places and locations as well as to enable the robot to autonomously go to verbally specified locations (e.g., 'Go to the kitchen').

The development of these capabilities is framed by the so-called home-tour scenario which is driven by the vision of future household robots being introduced for the first time use after purchase. A robot needs to get to know its new working environment which cannot be pre-programmed, but which can be explored together with inexperienced users in an interactive manner. Hence, human-robot interaction about the spatial and functional environment is in the focus of research in such home-tour scenarios. Capabilities a home-tour robot must reveal for natural interaction comprise understanding of spoken utterances, co-verbal deictic reference, verbal output, referential feedback, as well as person attention and following. The sketched functionality has been achieved by integrating modules for robot control, person tracking, person attention, speech recognition, speech understanding, dialog, gesture recognition, object recognition and object attention.

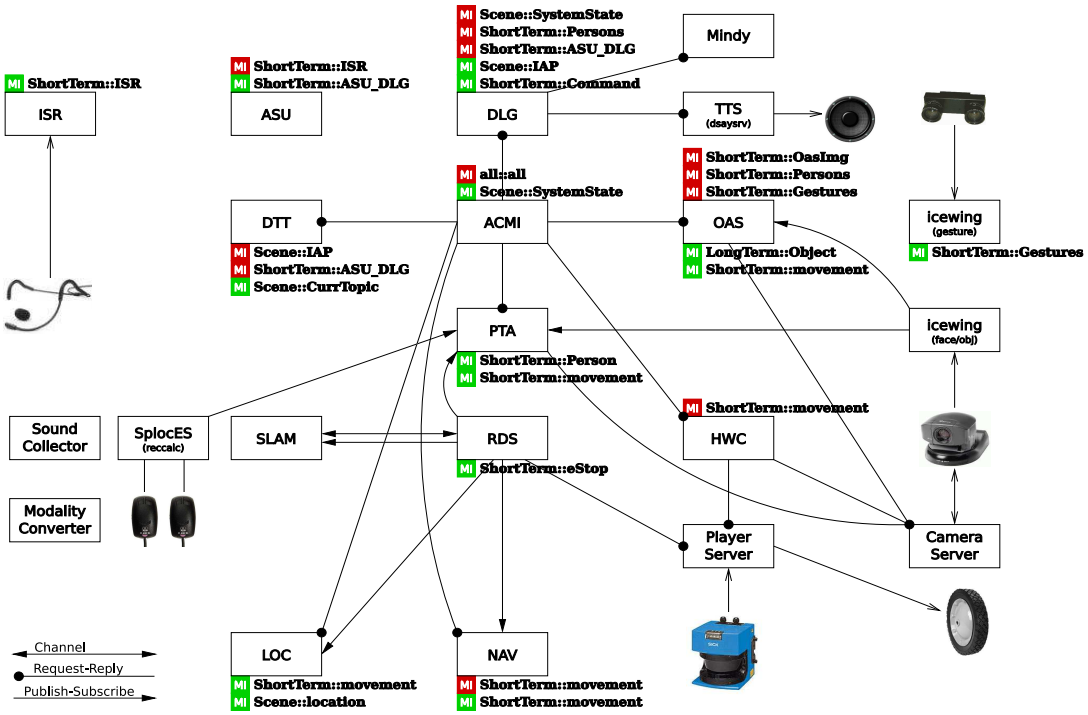


Figure 9.2.: *System architecture of the BIRON robot companion based on the IDI approach (from [Sie08]). All component interaction is based on the introduced interaction patterns utilizing three different instances of an active memory. Green and red rectangles attached to components indicate memory access while request-reply and publish-subscribe is indicated by the different line types.*

Following up on a brief explanation of BIRON's system architecture, the novel face memory part will be explained and the activities of the system when a user is entering a discourse with this robot aiming at natural human-robot-interaction are exemplified as an integration scenario.

9.1.1. System Architecture

From a functional viewpoint, the original architecture of BIRON was inspired by a three-layer hybrid architecture [FKH+05], as it yielded a flexible way to organize a system which integrates autonomous control and human-robot-interaction capabilities. In order to further enhance the robot companion with additional functional features, we adapted the previously realized integration and control architecture towards an extended use of the principles of information-driven integration and the memory model concepts. This refactoring of BIRON's software architecture was carried out in the course of the COGNIRON project [Sie08] with the aim to exploit concepts of information-driven integration to ease collaborative research and software development on this platform.

Figure 9.2 depicts a current sketch of the BIRON system architecture. It integrates over twenty different services that are realized by about twice the number of component implementations ranging from reactive processes such as obstacle avoidance (contained in the NAV service, see Figure 9.2) over arbitration functions (ACMI) to high-level processes for interaction and dialog control (DLG).

The extension of BIRON's capabilities by a face memory towards a robot capable of improved social interaction serves as a case study for the usefulness of the information-driven integration in cognitive robotics. Thereby, it demonstrates the interplay of perceptual and deliberative processes according to the introduced models.

9.1.2. A Face Memory for a Sociable Robot

Considering the ability to get to know and re-recognize human interlocutors by means of their face as a core cognitive function for a social robot we need to ask the question, how the mutual introduction and the recalling of faces is embedded into the general interaction scheme. Different sources of information are available, as the identity of a person might be the result of the current conversation ("What is your name?") or obtained from analyzing the person's appearance. Which knowledge source to combine is depending on the current content of a *face memory* and the conversational state.

The memory here serves as a central aggregator of relevant information. It allows the system to determine whether a person is known or not. In case the person is already known and the robot is certain about her identity it can just activate its knowledge about this person, while in the other case it has different options. First, it can take initiative and ask the new person for her name. Alternatively, it may continue conversation with an implicit but not yet named user model and wait for the name of the person to be mentioned sometime. Currently, the robot asks for the human's name whenever a yet unknown persons is engaged in conversation, hence implementing a certain curiosity in the robot behavior. However, it should be noted that the robot does not require to know every person in its vicinity. Besides recognizing and memorizing people's faces the system also comprises a person tracking and attention functionality called person anchoring that is similar to the anchoring process explained in the previous chapter. It establishes and tracks anonymous hypotheses about surrounding persons. Storing face views of these persons of interest alongside in a memory allows to immediately compute a new face representation an the basis of the last seen face patches. This is in accordance to human behavior, as we do not start looking at someone's face after hearing the respective name. Instead, we already memorize the appearance when initiating the conversation.

Figure 9.3 shows the part of BIRON's system architecture that is responsible for the realization of the face memory functionality. Utilizing the information-driven integration approach no changes have been necessary to reuse the already existing dialog subsystem. The coordination between the different processes is solely event-driven as explained in Chapter 6. While the perceptual memory is configured for short-term memorization of hypotheses generated from the stream of low-level sensor data, the episodical memory stores and processes higher-level symbolic information which is valid for longer periods of time. In the following, we will shortly describe the functionality of the different integrated services as depicted in Figure 9.3.

Perceptual Processing

Within BIRON's system architecture a number of different components performing bottom-up processing of incoming sensor data provide large parts of the perceptual capabilities of our interactive robot. For the face memory functionality we focus on three of these processes. Firstly, a voice detection component analyzes the cross-power spectrum phase to estimate the relative locations of multiple speakers. As soon as a speaker is detected, this and the spatial origin of the corresponding audio

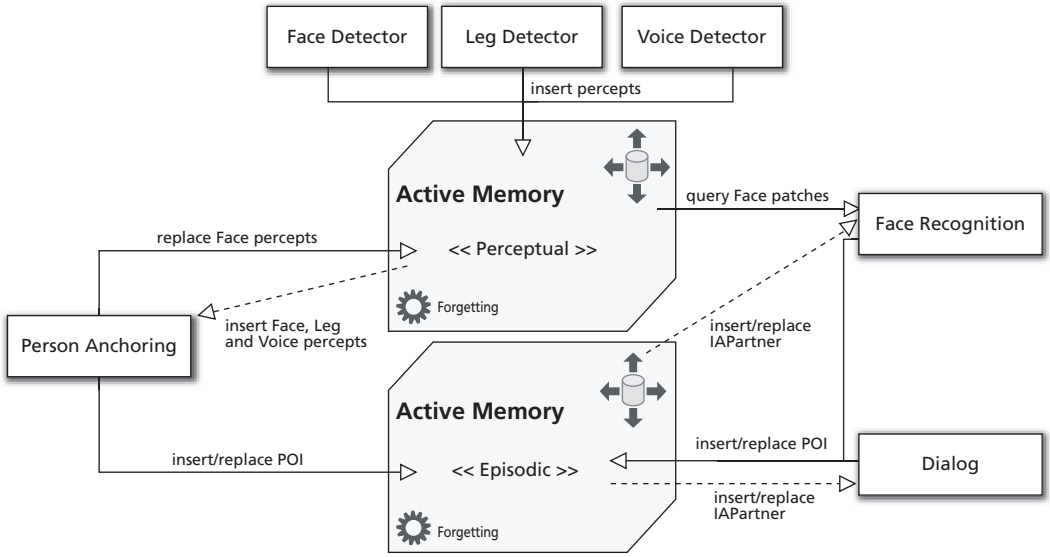


Figure 9.3.: *Relevant parts of BIRON's architecture for an interactive face memory shown in the notation introduced earlier. The face memory utilizes two memory instances and several perceptual services on different functional levels.*

signal is submitted to the perceptual memory. A second process is a "leg" detector, which scans the surrounding of the robot for pairs of legs by analyzing the data available from the attached laser scanner in order to generate hypotheses about possible human interaction partners standing or moving in front of the system. The face detection component as the third process in this extracts for each video frame the detected facial regions and inserts these in the perceptual memory of our robot together with a referring face hypothesis for subsequent processing.

Person Anchoring

A key component facilitating a face memory for an interactive robot is a multi-modal person anchoring process. Our realization is inspired by the approach introduced by Coradeschi & Safiotti [CS01] similar to the anchoring service used in the VAMPIRE assistance system. Anchoring in general can be interpreted as a process that links perceptual information about real world entities, e.g., faces, to symbols that reliably represent the found entity over a certain period of time. Anchoring processes in active memory architectures are employed to populate the episodic layer with information generated from the data available in the perceptual layer. Within BIRON's face memory, information generated by the individual modalities in the perceptual layer (face, legs and speakers) is anchored separately in the person anchoring component itself and is afterwards assigned to a person-of-interest (POI) hypothesis. A new POI will be created iff one of the input percepts does not match any of the existing modality anchors. Existing POI hypothesis are maintained as long as at least one of its three modality anchors can be tracked continuously. New POI hypotheses and their updates are submitted as episodic information to the corresponding active memory instance. Additionally, the person anchoring component frequently updates the face information in the perceptual memory with a reference link to the corresponding POI for subsequent use through the face recognition component.

Dialog

For a social robot it is important to be capable of social communication, e.g., by speech understanding. These features are realized in our robot by a dialog subsystem [LW07]. The currently realized model is inspired by the grounding-principle [Cla92], which states that within a conversation the communication partners need to coordinate their mental states based on their mutual understanding.

Adding up on the actual dialog functionality, a social robot must be able to distinguish between different persons communicating actively in its surrounding and to identify as well as align its communication to a human interaction partner focusing its attention on the robot itself.

While the former function is in the responsibility of the person anchoring module, the latter is an additional service that is provided by the dialog subsystem. As soon as the dialog is triggered by a specific initiation phrase (*"Hello Biron!"*) from a person that is registered as a POI in the episodic memory, the dialog selects this person as its interaction partner (IP) and in turn submits this new information with the preserved ID from the POI hypothesis to the episodic memory.

The identification of its communication partners without repetitive asking the human for his name, significantly improves the interaction experience. This is due to the fact that the dialog manages individual user models stored in the episodic memory. In consequence, it is possible to, e.g., adapt the speech recognition component to speaker dependent profiles before a conversation starts or to optimize its interaction by not repeating instruction already known by the respective person.

Face Identification

Within this architecture, the face recognition component, which is described in greater detail in [Lan07], makes use of several sources of information generated by other components, e.g., the IP hypotheses and their corresponding face patches. This information is utilized to perform the classification of the robots' communication partners. When the user has been trained previously and the classification is successful, the information available in the episodic memory is updated by the corresponding class name. Otherwise the face patches are used to train a new classifier as soon as the name of the communication partner has been acquired through the dialog subsystem. While the POI anchors and the name of the current interaction partner - if set by the dialog component - is retrieved through event notifications from the episodic memory, a query on the perceptual memory is performed to retrieve recent face patches that correspond to the current interaction partner.

9.1.3. Interaction Scenario

Figure 9.4 exemplifies the dynamic interactions between the components of the system in terms of activities that are carried out in the face memory when a human user enters the robots' interaction area, looks at BIRON and finally starts the interaction by greeting it with the initiation phrase.

As soon as the human approaches the robots sensors and her legs are detected, the episodic memory notifies the person anchoring component about the new "leg" percepts. In turn, a local modality anchor is set up and a new POI hypothesis is submitted to the episodic memory. While possible in parallel, let us assume for this example that the face of the user is detected as he further approaches the robot. As a consequence of this activity, two things happen concurrently: the detected faces and their views

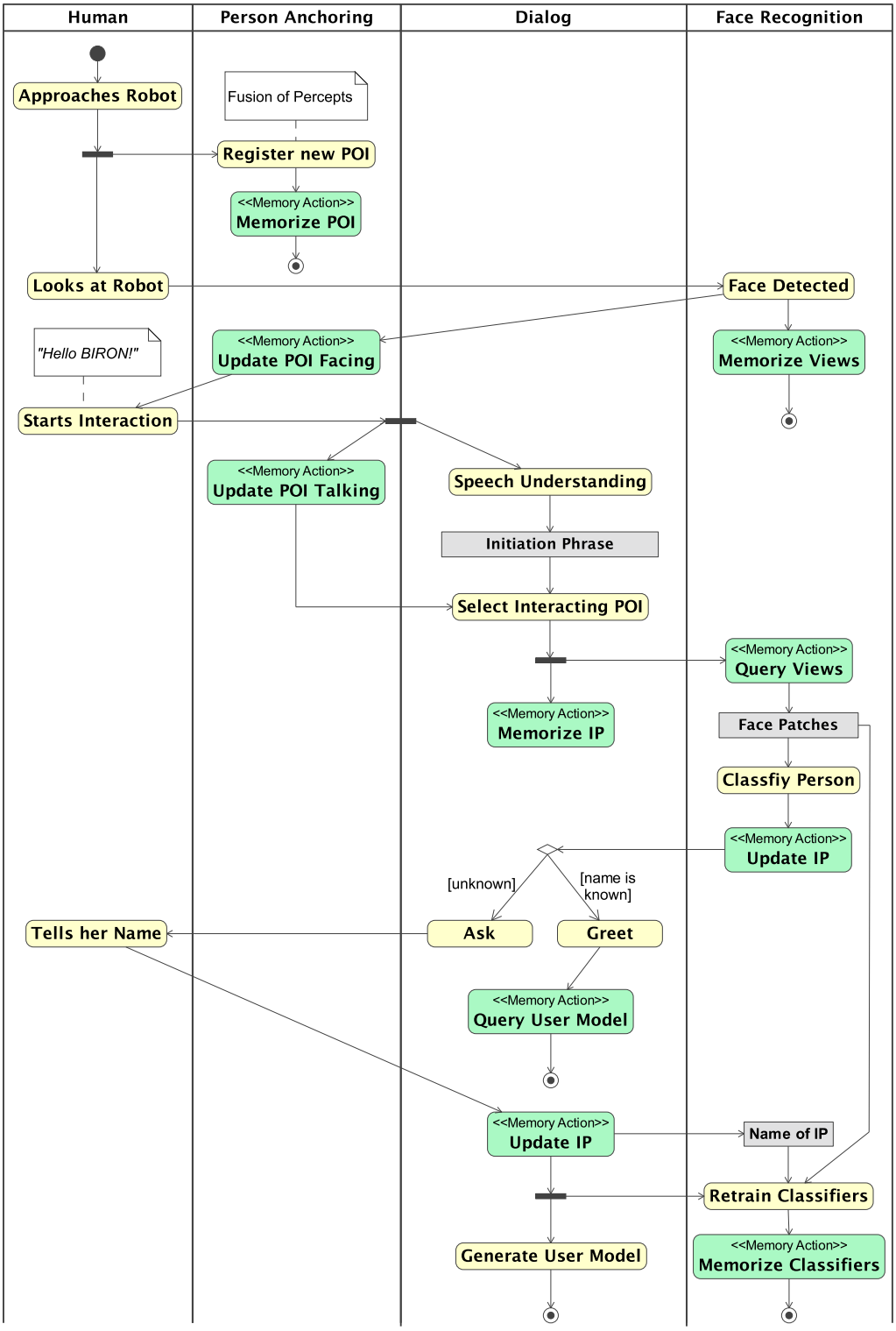


Figure 9.4.: *Activities carried out by the human communication partner and the robot system when initiating a conversation utilizing a face memory. Interactions with active memory instances are shown in green boxes, while other relevant data flow is shown in Grey rectangles.*

are submitted to the perceptual memory and the previously established POI profile is extended by the information that the person is now “facing” the robot.

The next action the human carries out to start a new discourse with BIRON is to address him by speech using the initiation phrase. This leads to an update of the corresponding POI hypothesis, which is enhanced by the information that this person is talking. Additionally, the dialog component selects this POI as its interaction partner iff the speech understanding result provides the symbol for the initiation phrase.

Dependant on this decision, the information about the selected IP is submitted to the episodic memory. Once the IP hypothesis is available, the face recognition component is activated by a corresponding event and starts to query the recent face patches corresponding to this interaction partner. These views are used for the following classification step that yields an update of the IP hypothesis in the episodic memory. It is enhanced either by the name of the communication partner in case of a successful classification or it just left empty to indicate that this human is so far unknown.

The following activity, once more triggered by the update of the IP hypothesis is carried out within the dialog component in two different ways based on the name information updated previously by the face recognition available within the IP hypothesis. When the system does already know the name of its communication partner, the final activities in this example are the retrieval of the corresponding user model from the episodic memory as well as the adaptation of the dialogs’ interaction strategy and the greeting of the IP using its name. In case the user could not be classified successfully from the set of recent face patches, the dialog subsystem asks the user for this information and uses the label retrieved from its speech recognition module to update the IP hypothesis with the given name. In this case the face recognition is triggered by the updated IP hypotheses and starts to train a classifier for this previously unknown person. Finally, the dialog generates a new user model that is used in subsequent interactions when this communication partner is hopefully recalled by the robots’ face memory system.

The previous sections present a unique face memory that is conceptually well integrated into a larger architecture of robot companion using the IDI architecture. It links interactive introduction of interlocutors with an online learning face classification scheme. The results presented in [HWLS08] not only confirm that the face memory facilitates a way of mutual control necessary for a socially acceptable interaction and the adequacy of the chosen perceptual methods, but also elicit the benefits of the information-driven integration approach and hence underpins its suitability as a basis for building hybrid software architectures for robots with cognitive abilities.

Before we are going to discuss some of the insights gained during application of the IDI architecture in the COGNIRON project for the collaborative development of BIRON’s software layers, let us briefly look at two other robotic research systems that make use of the approach presented in this thesis.

9.2. An Anthropomorphic Robot for HRI Research

In contrast to the aforementioned mobile robot, BARTHOC is an anthropomorphic robot, cf. Figure 9.5, that is able to show facial expressions and use its arms and hands for deictic gestures for effecting more natural interaction with humans. On the other hand the robot can also recognize pointing gestures and also coarsely the mood of a human, achieving an equivalence in production and perception of different communication modalities.

Taking advantage of these communication abilities a system has been developed where BARTHOC provides information retrieval services acting similar to a receptionist. As a first intermediate step towards this scenario research on the task of introducing the robot to its environment has been carried out [SHS07]. This scenario already covers a broad range of communication capabilities. The interaction mainly consists of an initialization by e.g. greeting the robot and introducing it to new objects, which are lying on a table, by deictic gestures of a human advisor.

Once more software integration is necessary to integrate all the perceptual and deliberative components needed for an experimental realization of this scenario. The software architecture of BARTHOC lends itself to a good example for supporting effective research with the information-driven integration architecture. For BARTHOC, basically the same set of services could be reused as are running in the BIRON systems due their loosely coupled software design applying the IDI models. Mainly, hardware control and a scenario specific component needed to be added or replaced. All other services like dialog or perceptual processes are furthermore able to operate in both scenarios. Thus, efficient research on experimental cognitive systems is effected as the duplicated development of similar functionalties can be avoided.

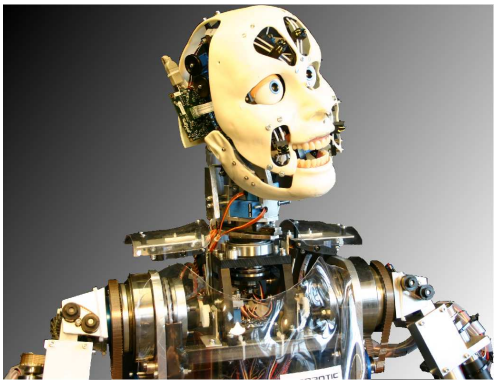


Figure 9.5.: *The head of BARTHOC without its artificial skin.*

In the scenarios addressed with the BARTHOC robot, all applications avoid the usage of human-unlike wide range sensors like laser range finders or omnidirectional cameras. However, in order to avoid loosing track of interaction partners due to the limited area covered by the given sensors, recently a short time person memory was developed that extends the existing anchoring of people, which once more underlines the general utility of memory functions for cognitive systems aiming at interaction [SHS07]. Furthermore, a long time memory was added to store person specific data, which can be recalled to improve tracking results.

9.3. A Control Architecture for Manual Intelligence

As motivated by Ritter et al. in [RHS07], the study of manual intelligence, e.g., how human-like grasping capabilities can be transferred to an artificial cognitive system, may serve as a key problem for the design of cognitive robotics architectures that is more manageable than the design of a complete functional cognitive architecture. Even so, their hypothesis is that grasping is a sufficiently rich problem to provide essential insights into the architectural principles enabling natural cognition.

Manipulative acts involve the structuring of a complex physical interaction between a highly redundant, articulated manipulator and a physical object as shown in Figure 9.6, which can be highly variable. Dextrous manipulation is a form of mechanical control that is pervaded by frequent discontinuous changes of the contact topology between the actuator and the object. As a result, dextrous manipulation calls for an unusually tight coupling between continuous control and more discrete, symbol-like representations that can deal with issues such as topology-switching and encapsulation of parameter uncertainty.

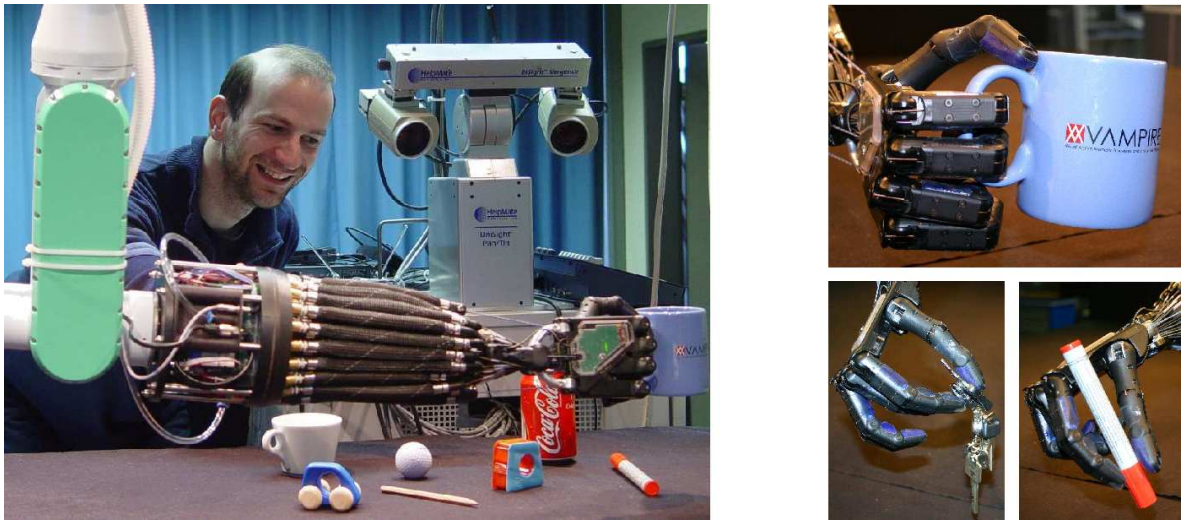


Figure 9.6.: *Grasping as a "rosetta stone" for research on cognitive models (from [RHS07]).*

According to Ritter et al. [RHS07], it is the level of coupling between continuous, sensorimotor control loops and discrete, symbol-like representations that seems to be a prerequisite for realizing system structures and corresponding architecture, which shall finally lead to improved cognitive capabilities.

In order to conduct experimental research on the aspects of manual intelligence, a software architecture was developed that features a tight interconnection and coordination between subprocesses as well as a highly structured and modular design of the system and its state representation. In order to achieve this, dynamically configurable *Hierarchical State Machines* (HSM) are used, which reflect high-level system states encoded symbolically. These HSM's coordinate several behavior controllers that directly interfacing on a subsymbolic level low-level hardware controllers of a bi-manual hand/arm robotic system. For the interested reader [RHS07] provides further details about the technological and algorithmic properties of this approach.

However, even in this tightly coupled system which is not directly the primary target domain of the presented approach, the IDI architecture could be successfully applied to integrate the different HSM services. In the realized system the modality- and context-specific interaction patterns of a significant number of low-level subsymbolic processes are bound to elements provided by the HSM model. These elements feature a semantic interface on a symbolic level. For the event interchange between these individual HSM elements, event publishing and matching functions of the IDI architecture are used in this scenario. For the interaction with a number of external processes, e.g. to integrate perception and interaction services, the features of the memory and interaction models are applied.

Due to the fact that grasping and the interaction between model elements in this architecture represents a task that is much more sensitive to timing issues than the previously introduced examples, the utility of the IDI architecture with regard to this aspect is underlined.

9.4. Summary

The emphasis of this chapter was to demonstrate that the concepts of the IDI architecture were transferable to the new domain of interactive cognitive robotics. As one of the primary application areas of cognitive systems, robotic systems and corresponding research projects represent important opportunities for further studies on software integration and software architecture.

In contrast to rather low-level robotics middlewares which have been described in Chapter 5, the previous sections outlined that the use of the presented approach even in robotics is geared at a higher level of abstraction. Even so, its performance is still sufficient to coordinate system components that directly deal with reactive services or to interface with components that are closely coupled to actuators.

Furthermore, all of the presented systems make use of features that are part of the interaction and in particular the memory model. This underlines that memory features are an important generalizable function in cognitive systems and that the chosen approach was versatile enough to become applicable across different scenarios. In the BARTHOC and BIRON scenarios, the active memory additionally provides an avenue for dynamic adaptation and reconfiguration as, e.g., behavioral specifications are stored in the memory and automatically distributed to respective control components as soon as those are updated by other processes. On the basis of the memory model functions, current research [HS08a] is concerned with the identification of reusable domain specific interactions patterns that facilitate integration on an even higher level of abstraction.

The previous examples underlined that the IDI architecture not only provides the profound technological basis for software integration of experimental cognitive robotics but additionally facilitates collaborative work and software reuse.

Part IV.

Synopsis

A brief conclusion that summarizes and reviews the benefits of information-driven integration in the context of collaborative research projects on cognitive systems shall commence this dissertation.

10. Conclusion

I rarely end up where I was intending to go, but
often I end up somewhere that I needed to be.
– Douglas Adams

Douglas Adams saying is a good metaphor for research on developing a software architecture for experimental cognitive systems. The presented approach emerged by iteratively identifying requirements on software integration in cognitive systems research, evaluating related work and successively integrating and testing generalizable relevant functionality. Evolution occurred not only by adding new functionalities like the active memory but also with regard to the conceptual foundations of the IDI architecture. The architectural core evolved from a closed approach based on remote-procedure call techniques to a generic and extensible event-driven architecture with strong support for service-oriented principles.

This conclusion summarizes the key aspects of the information-driven integration approach, relates the developed concepts and insights found to the three perspectives spanned at the beginning of this thesis, to the identified requirements and finally to the research question posed in the beginning. To commence this dissertation an outlook is provided on possible future research directions.

10.1. Information-driven Integration in a Nutshell

The IDI architecture is a middleware with particular support for the integration tasks in experimental cognitive systems research. It enables efficient communication between applications and devices in a network of heterogeneous standard computers and operating systems by combining methods from service-oriented and event-driven architectures as well as tuplespaces into a coherent approach.

It directly supports publish-subscribe as well as request-reply and channel-based group communication patterns, virtually shared memory services, URL-based naming services, and permits expressive matching of extensible events, effected by a hybrid subscription model operating on XML documents. Event matching utilizes an extended message transformation approach that permits the use of stateful filters such as a compacting filter, which can, e.g., be applied to retrofit the interaction behavior of legacy applications. A coordination feature for modeling and external control of discrete event-based system components and universal adapter plugins for a modular toolkit focused at the development of real-time computer vision algorithms round out the services provided through the core architecture.

Based on this core, a set of tools are offered that ease practical system development, which, for instance, permit a distributed and transparent monitoring of the dynamics in a cognitive system architecture. The architecture has so far been implemented in C++ and Java and thus provides a good level of platform independence, which makes it for instance directly usable in Matlab [TM08] environments.

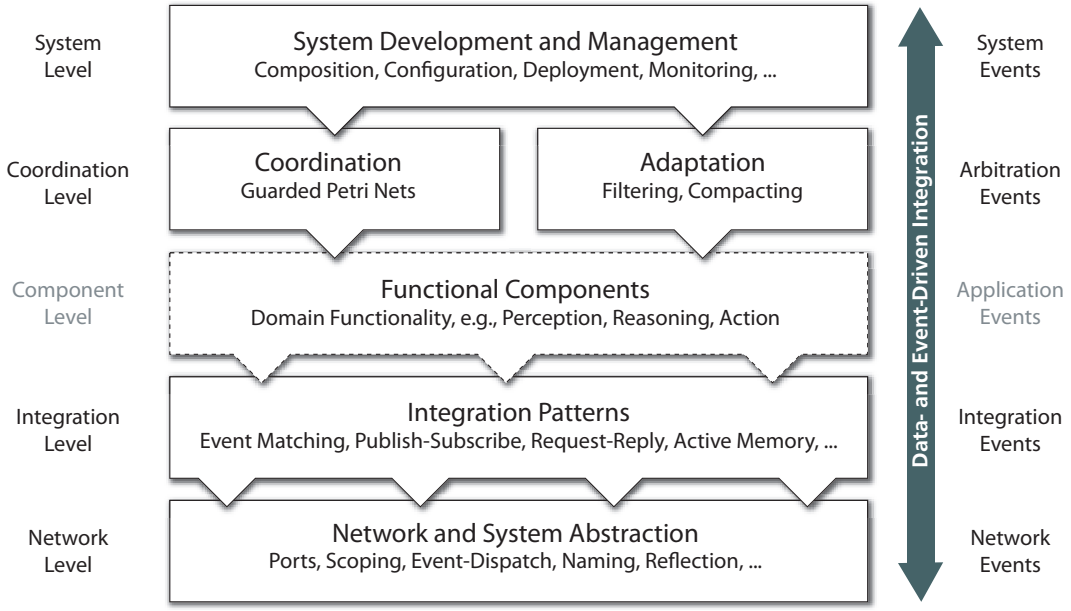


Figure 10.1.: *Overview of the information-driven integration architecture from a system-engineering perspective. The layers correspond to functionality available for component developers and system architects. While the former primarily deal with the component and integration level, the latter usually additionally exploit services in the upper layers. Enhancements are possible across all layers, including transport-specific extensions in the lowest layer.*

The different models that were introduced in Part II of this dissertation largely fulfill the needs that were identified during the requirements analysis carried out in Chapter 5, yielding a layered stack of functionality as shown in Figure 10.1 that supports the engineering of experimental cognitive systems. A particular focus of this work has been how to foster software development in collaborative research projects on experimental cognitive systems.

10.1.1. Facilitating Collaborative Development

While collaboration has been studied in software engineering and social sciences in great detail, cf. Chapter 3, software architectures for cognitive systems did not credit this issue first class importance. In contrast, the work in this thesis treats this aspect as critical for project success not only in large-scale projects from business information technology but also for larger collaborative, usually international, research projects. Recently, Ceravola and Goerick [CG06] did recognize this aspect, too. However, their approach is focused at application in a rather closed organizational integration context and might thus not be transferable to an oligarchic or anarchic environment as defined by collaborative research projects.

Acknowledging the importance of this perspective, strategic aims were derived that directly reflect this insight and have huge impact on subsequent design decisions. First and foremost, the resulting emphasis on loose coupling influenced many other software architectural aspects such as to choose an event-based integration style.

Considering the collaborative aspects and thus loose coupling as primary aims leads to many conflicting requirements. For instance, a compromise had to be found balancing fast component interactions with low latency on the one hand and programming models that actually implement these interactions in a loosely coupled manner and on a high abstraction level on the other hand. If in doubt, the approach taken in this thesis was to prefer usability, modularity and abstraction over performance, because it was unclear whether up front optimization would have been of any real benefit.

Due to the performance requirements and particularly the level of usability required, general purpose middleware or commercial solutions that provide similar or even more features were no suitable alternatives to the partially simpler, partially more complex problems of the given domain. Despite the non-commercial research background of the presented architecture it certainly resembles to a kind of novel *enterprise service bus* known from enterprise integration admittedly with a strong bias towards cognitive systems research environments.

10.2. Insights and Observations

The information-driven integration model has been successfully utilized for the design, development and operation of several instances of experimental cognitive systems in the VAMPIRE and COGNIRON EU projects as outlined in the previous two chapters. Furthermore, the IDI architecture has been used for the integration of several other systems of different sizes, e.g., [VAM06, Cog06, DES08, WKF07, SHS07], ranging from small student projects and projects with educative purposes to further individual projects embedded in larger national research programs and large-scale projects on service robotics.

Besides the iterative refinement of the integration architecture's core functionality and concepts, this broad application allows to reflect - from the three viewpoints defined in the introduction - on how different developers and architects actually used the IDI concepts and permits to discuss some lessons learned during the course of these projects.

10.2.1. The Functional Viewpoint

Successful integration and demonstration of many resulting system instances and the ability to evaluate those in real-world experiments with naive users actually serves as a proof for the suitability of the chosen system architecture for interactive cognitive systems. Besides that, the concepts of the IDI architecture that contributed most to the different projects from a functional perspective will be explained in the following and contrasted with popular alternatives:

- *Asynchronous vs. Synchronous Interaction*: Utilizing an event-driven architecture for asynchronous integration of many independent processes running in parallel in a distributed system actually increased the modular protection between individual components. In contrast to the synchronous, operation-oriented model, the failure of individual components is less critical as components in an EDA shall make only minimal assumptions about state of other participants. In contrast, components using a synchronous architecture may freeze a complete system due to resource starvation whereas in event-driven architectures, the components simply do not receive new events and may still be able to react on exceptional conditions.

- *Document- vs. Object-oriented Representation*: The use of XML helped in defining data types which were suitable for every involved project partner, particular if it was acted upon the guidelines of the document model, e.g., the *must-ignore* principle. In contrast to object-oriented class hierarchies, information-oriented representations facilitated the design of coarse grained service interactions, effecting loose coupling, improved understandability and performance.

- *Unified vs. Domain-specific Data Access*: Based on the document-oriented data model and the global event bus, both subscriptions evaluated on the transient event-based conversations as well as queries on the memory content expressed with XPath statements allowed components to freely retrieve information in a standardized fashion using declarative specifications. To achieve similar functionality based on, e.g., object-oriented data structures much more specific infrastructure and programming models like OQL [Obj00] would have been necessary.

- *Active Memories vs. Component-specific Data Management*: The active memory has been critical for the overall architecture and function of many of the systems developed so far for many reasons. Through extension of a native XML database towards a virtually shared memory, information becomes easily accessible for components utilizing declarative XPath expressions. Hence, it prevents the emergence of component specific *data silos* in cognitive system architectures. Interpreting the memory as an extended event-driven tuplespace architecture allows the design of high-level interaction protocols composed by a simple set of atomic operations.

- *Active Forgetting vs. Lease Times*: Modeling forgetting as an autonomic process frees developers from keeping track of memory element lifecycle. This eases component implementation, allows to associate memory element types with different temporal semantics and permits forgetting processes to evaluate meta information not considered by individual components prior to removing any memory elements.

- *Explicit Coordination vs. Stateful Interaction*: The development of guarded Petri nets permits rigorous modeling and simulation of system behavior on a high level of abstraction even without the actual components at hand. The concept of guards that are connected to the observation model provides a generic semantic coupling of this model to actions executed in an integrated system in order to effect a specific task behavior. The externalization of control from components to federations of domain specific controllers permitted to further reduce the complexity of individual components and limit their use of stateful interactions with other services.

- *Generic vs. Application Specific Adapters*: Within computer vision projects, partners expected support for the development of corresponding algorithms. By effecting the modularization of IceWing and the provisioning of generic infrastructure plugins, developers were able to use a well suited and efficient vision toolkit while system architects did benefit from an easy integration of processing results by the introduced set of generic plugins. This approach was far more useful than integrating each application manually with specific application adapters.

While these points were all beneficial, some aspects need further investigation. For instance, it is rather straightforward to define some heuristics when to apply which interaction pattern. In contrast, it is pretty hard to give generally applicable guidelines for decomposing individual components or services in terms of their dynamic coordination, e.g. when to use a subscription inside of a component and when to externalize stateful interactions with other components with the features of the coordination model.

10.2.2. The Collaborative Viewpoint

Besides fulfilling the functional requirements, the IDI architecture was designed right from the beginning towards facilitating collaborative software development as highlighted in the beginning of this section. Hence, one of the actual outcomes in this regard is high usability of the different features through a clear and straightforward programming model.

On the one hand, this is facilitated through the use of a limited set of recurring object-oriented building blocks, e.g., the possibility to register event-based callbacks with the same interface at different interaction patterns or the integration of a polymorphic event dispatching method. The Java API sketched in Part II represents the state of the current iteration of the corresponding framework implementation as a result of this joint effort. On the other hand, the use of standards based XML technologies throughout the whole framework, e.g., by using XPath both for selecting memory content as well as for the specification of content-based event subscriptions and the consideration of developer feedback further promotes usability and extensibility. Besides aiming at high usability, the following observations could be made during the different projects with regard to the proposed integration methods and questions of collaboration (again briefly contrasted with popular alternatives):

- *Pattern-based vs. Object-Oriented Vocabulary*: The definition of a common vocabulary for interaction patterns and integration entities such as *documents*, *events*, *subscriptions*, *services*, *components* and *interfaces* fosters efficient communication between developers about essential structures of cognitive systems on the level of the integration architecture. In contrast to describing software on the level of object-oriented structures, complexity can be reduced by omitting irrelevant detail in compact architectural descriptions; see Figure 8.7(b) for an example.

- *Dynamic vs. Static Interfaces*: The consequent focus on dynamic middleware techniques and the reflective properties of the document, notification and naming models eased the development of generic monitoring tools that were highly valuable during system development and for controlling the correct processing of data at the system integration level at runtime. For instance, central logging of participant interactions in BIRON eased the tracing of typical processing paths usually found in complex robot architectures.

- *Simulation vs. Live Operation*: The event-driven approach facilitates debugging and evaluation of integrated cognitive systems by *replaying* recorded event notifications. As the event metadata provides time as well as sender and receiver information, cf. Section 6.3), architectural layers can with certain limitations be replaced by components that are simulated by a generic emulation service. Development and evaluation of different algorithms or system configurations on comparable data has been much easier with this feature.

Though many of these assessments are truly subjective, the technology foundation has been laid by this work and its appropriateness has been substantiated by the numerous systems built. A possible area of future research could be to provide factual evidence on the performance of chosen engineering methods as done in software engineering by developer observation [PPV00, Sea99], e.g., encoding what amount of time is consumed by certain types of tasks, e.g., reviewing the documentation or system testing, during a coding or integration session. This discretization allows a kind of quantitative evaluation of the achieved usability. Besides that, simulation and testing imposes unsolved challenges with regard to emulation of complex and non-discrete components.

10.2.3. The Engineering Viewpoint

The first observation in this context was that due to the chosen interface granularity and the separation of structured and binary data contents, the transport and processing of XML data in distributed system architectures has up to now been no critical bottleneck for system reactivity. Furthermore, through the use of XML- and pattern-based service interfaces, the *flexibility* requirement was fulfilled. This results in changeability, easier adaptation and integration of new modules. As an example taken from the COGNIRON project, an existing localization and mapping module was replaced by a different module from other project partners in just one day [SSS08]. Utilizing the information-driven approach, even after this modification, the previous module could be instantaneously reactivated as no specific identity or reference type information was part of the service interfaces. Let us consider some additional lessons learned from the engineering viewpoint:

- *Interaction Patterns vs. Remote-Method Invocation*: In contrast to object-oriented remote method invocations, the abstraction level of integration is raised through the introduction of interaction patterns to that of architectural styles. Adding up on that, the defined interaction behavior is kept separate from the exchanged data messages and explicit distribution boundaries are enforced by the programming model. All these aspects contribute to the aim of encapsulating accidental but exposing essential complexity, cf. Chapter 4.
- *Loose vs. Tight Coupling*: Loose coupling is achieved through the event-based core, document-orientation and external configurability. A refactoring of the BIRON architecture on the basis of the IDI approach yielded in a dramatically reduced number of point-to-point connections [SSS08]. However, the ability to integrate vision algorithms in a tightly coupled fashion using the IceWing development environment, was essential for building real-time computer vision subsystems out of tightly coupled image processing plugins. Hence, the answer is not loose *or* tight coupling but to support both in a coherent way.
- *Declarative vs. Procedural Specification*: In contrast to burying relevant architectural detail in programming language constructs, most properties relevant on the level of the integration architecture can be declaratively specified in a way that is easily understandable by developers, if an information-oriented representation was chosen, cf. Section 6.2. Interpretability of the exchanged XML data types directly payed off in shorter development cycles during integration, because of the ability to view *and* to understand messages at runtime.
- *Schema Independence vs. Relational Schemata*: Another aspect useful both for development of the system and the runtime architecture is the ability to integrate new information types without having to physically restart any servers or to redeploy any database schema's as known from relational databases. The absence of fixed data schema's helps also in restructuring of content and allows for storing of completely new information structures which is useful, e.g., for learning architectures.
- *Independence vs. Vendor Lock-In*: Through the increased level of abstraction and the Port-based design of the core architecture, cf. Section 6.5, independence of a specific middleware technology or vendor is achieved. For instance, while previous versions of the resulting IDI architecture were based on the Internet Communication Engine [HS08b], which is an innovative object-oriented middleware, the current implementation is based on Spread, which is a group communication framework and thus promoting completely different concepts. Even so, the basic principles of the IDI architecture remained stable. This independence increases flexibility and avoids critical dependencies on a single technology.

From an engineering perspective, further work must be primarily carried out on scalability aspects with regard to the active memory implementation and the matching algorithms in the observation model. Regarding the latter, many approaches for optimized matching of multiple XPath statements, e.g., [CFGR02], exist that are well suited for evaluation and possible extension. The replacement of the request-reply based event handlers for the memory services by fully event-driven interfaces paves the way towards linear scalability of the memory model based on a flexible partitioning of the overall event space.

A disadvantage of the XML-based data exchange, which additionally impedes usability at first sight is that due to the call for information-driven representation, most methods for automatic data binding from abstract data types in programming languages to XML documents are not applicable. However, despite the fact that the position taken in this thesis is that careful design of shared data structures and corresponding domain specific accessor classes is worthwhile in software integration, a template-based XML parsing and serialization library [FW07] was developed in the context of this dissertation project, which dramatically eases this task and provides a solution to this problem.

10.3. Some Answers and New Questions

In the introduction of this thesis, a number of questions were posed that were addressed throughout this dissertation. Let us for the conclusion shortly recall the primary research question that guided my work in the different projects and on the development of the integration architecture: "*what are architectural concepts and paradigms suitable for handling the innate complexity of software development in cognitive systems research projects?*"

Not surprisingly, this thesis does not answer this question to its full extent and with universal validity. In order to do so, even more systems of different kind need to be developed, although already quite a number of examples exist that utilities the presented approach. Thus, the introduced models of information-driven integration and their concepts as presented in Part II of this thesis may pave the way towards further investigation on this question. For the projects the architecture has been applied in so far, the combination of functional and event-based composition, the exploitation of the in-band information for component coordination, and an increased level of abstraction for the design of these interactions in cognitive systems architectures are important aspects of an answer. This thesis presented a novel, holistic perspective on an emerging topic in experimental cognitive systems research, introducing an architecture that considers the integration context as an important source for specific complexity and adopting state-of-the-art methods from current software engineering research on software integration and distributed systems into a coherent and innovative approach, thus making them easily usable for cognitive systems domain experts.

Picking up on the introduction, the specific challenge of this work has been to find a *working* definition of an integration architecture that puts users and the researcher on software architectures in cognitive systems both in a position where they can explore new issues and ask questions they simply could not have asked earlier. Emphasizing the term *working* is of particular importance here as this beneficial situation can only be achieved if a technically sound platform is available, which inevitably is a huge engineering and dissemination challenge.

New Opportunities

However, as this state has been reached for the IDI architecture and stable implementations of the presented concepts are available, this yields an excellent opportunity for further research on advanced architectural functions. A promising trait for further research is to analyze the resulting dynamics of component interactions in these systems in an autonomic computing approach applying pattern recognition and data mining techniques in order to autonomously classify the situational context of a cognitive system. On the long run, this may lead to a meta-level for system self-awareness. Other examples are questions of adaptive coordination, where intelligent coordination models can be learned from data available at an architectural level or to further investigate the question what the specifics of the visual active memory are with regard to architectural style and cognitive architectures.

Finally, it should be mentioned that all the lessons learned could only be learned by a research policy that aims to actually build systems for the real world and by having great collaborators in the different projects the presented concepts were applied in. Let me thank them here for their commitment, patience and dedication to support the development of this approach.

Bibliography

[ABB+01] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. Macintyre. Recent advances in augmented reality. *Computer Graphics and Applications, IEEE*, 21:34–47, 2001.

[Acc08] Accusoft. Visiquest data and image analysis software, 2008. http://www.accusoft.com/products/visiquest/.

[AGK08] David Abrahams and Ralf W. Gross-Kunstleve. Boost.python. `http://www.boost.org/doc/libs/release/libs/python/doc/`, 2008. last checked 05/30/2008.

[Alb00] J.S Albus. 4-d/rcs reference model architecture for unmanned ground vehicles. *Robotics and Automation, 2000. Proceedings. ICRA '00.*, 4:3260–3265, 2000.

[And93] John R. Anderson. *Rules of the Mind*. 1993.

[AR08] Manfred Broy Andreas Rausch. *Das V-Modell XT: Grundlagen, Erfahrungen und Werkzeuge.* Dpunkt Verlag, 2008.

[AS98] Yair Amir and Jonathan Stanton. The spread wide area group communication system. Technical report, The Center for Networking and Distributed Systems, The Johns Hopkins University, 1998. CNDS-98-4.

[ASTMvS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[AZ05] P. Avgeriou and U. Zdun. Architectural patterns revisited - a pattern language. In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, Irsee, Germany, July 2005.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[Bai05] Baillie. Urbi: Towards a universal robotic low-level programming language. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems - IROS05*, 2005.

[BBHR04] H. Bekel, I. Bax, G. Heidemann, and H. Ritter. Adaptive Computer Vision: Online Learning for Object Recognition. In *Proc. Pattern Recognition Symposium (DAGM)*, volume 3175 of *LNCS*, pages 447–454. Springer, 2004.

[BBV06] Joscha Bach, Colin Bauer, and Ronnie Vuine. Micropsi: Contributions to a broad architecture of cognition. 4314:7–18, 2006.

[BCG07] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley & Sons, 2007.

[BCH+96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *OOPSLA '96: Proceed-*

ings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 69–82, New York, NY, USA, 1996. ACM.

[BCK05] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley Longman Publishing Co., Inc., Boston, USA, 2005.

[BCTW96] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering Methodology*, 5(4):378–421, 1996.

[BDG01] M. Birbeck, J. Duckett, and O.G. Gudmundsson. *Professional XML*. Wrox Press Inc., 2nd edition, 2001.

[Ber04] Berkeley DB Reference Guide Version 4.2.52. Technical report, 2004. Last checked 30th June 2004.

[BGD05] F. Bajramovic, Ch. Gräßl, and J. Denzler. Efficient Combination of Histograms for Real-Time Tracking Using Mean-Shift and Trust-Region Optimization. In *DAGM*, Heidelberg, 2005. Springer.

[BHW+05] Christian Bauckhage, Marc Hanheide, Sebastian Wrede, Thomas Käster, Michael Pfeiffer, and Gerhard Sagerer. Vision Systems with the Human in the Loop. *EURASIP Journal on Applied Signal Processing*, 2005(14):2375–2390, 2005. `http://www.hindawi.com/GetArticle.aspx?pii=S1110865704411275`.

[BHWS04] Christian Bauckhage, Marc Hanheide, Sebastian Wrede, and Gerhard Sagerer. A Cognitive Vision System for Action Recognition in Office Environments. In *Proceedings International Conference on Computer Vision and Pattern Recognition*, number 2, pages 827–832, 2004.

[Bir05] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[BKM+05] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. Towards component-based robotics. *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 163–168, 2-6 Aug. 2005.

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986: Uniform resource identifier (uri): Generic syntax. Technical report, The Internet Society, 2005.

[BMRS96] F. Buschmann, R. Meunier, H. Rohnert, and P. Sommerlad. *Pattern-Oriented Software Architecture*, volume 1: A System of Patterns. John Wiley & Sons Ltd., 1996.

[Boe88] Barry Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21:61 – 72, 1988.

[BPSM+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation. Technical report, World Wide Web Consortium, Feb 2004. `http://www.w3.org/TR/2004/REC-xml-20040204`.

[Bro91] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1-3):139–159, 1991.

[Bro95] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, anniversary edition, 1995.

[Bru05] Davide Brugali, editor. *IEEE ICRA 2005 Workshop on Software Development and Integration in Robotics (SDIR-I)*, Barcelona, Spain, 2005. IEEE RAS TC-SOFT, IEEE Robotics and Automation Society.

[Bru07a] Davide Brugali, editor. *IEEE ICRA 2007 Workshop on Software Engineering for Robotics II (SDIR-II)*, Roma, Italy, April 2007. IEEE RAS TC-SOFT, IEEE Robotics and Automation Society.

[Bru07b] Davide Brugali, editor. *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*. Springer, Berlin, 2007. ISBN: 978-3-540-68949-2.

[Bru07c] Davide Brugali. *Software Engineering for Experimental Robotics*. Springer Engineering, 2007.

[Bru08a] Davide Brugali, editor. *IEEE ICRA 2008 Workshop on Software Engineering for Robotics III (SDIR-III)*, Pasadena, CA, USA, May 2008. IEEE RAS TC-SOFT, IEEE Robotics and Automation Society.

[Bru08b] Herman Bruyninckx. Open robot control software, 2008. http://www.orocos.org.

[BS85] Ronald J. Brachman and James G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[Car06] Carnegie Mellon University's Software Engineering Institute. Software engineering glossary. WWW, October 2006.

[Cas03] Cristiano Castelfranchi. Cognitive systems: Towards an integration of symbolic and sensor-motor intelligence? *ERCIM News*, 53:10–11, April 2003.

[CAV07] CAVIAR Consortium. Caviar: Context aware vision using image-based active recognition, Aug 2007. IST 2001 37540, `http://homepages.inf.ed.ac.uk/rbf/CAVIAR/`.

[CBL+06] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud. Robotic software integration using marie. *International Journal of Advanced Robotic Systems*, 3(1):55–60, March 2006.

[CC94] Henrik I. Christensen and James L. Crowley, editors. *Experimental Environments for Computer Vision and Image Processing*, volume 11 of *Series on Machine Perception and Artificial Intelligence*. World Scientific Publisher, 1994. ISBN 981-02-1510-X.

[CC97] H. I. Christensen and Alain Chehikian. *Vision as Process: Basic Research on Computer Vision Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[CD99] James Clark and Steven DeRose. XML Path Language, W3C Recommendation. Technical Report REC-xpath-19991116, World Wide Web Consortium, Nov 1999. W3C Recommendation 16 November 1999, `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[CFGR02] Chee-Yong Chan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Efficient filtering of xml documents with xpath expressions. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, page 235, Los Alamitos, CA, USA, 2002. IEEE, IEEE.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CG06] Antonello Ceravola and Christian Goerick. An integrated approach towards researching and designing real-time brain-like computing systems. In *Proceedings of the AISB '06 Sym-*

posium on Nature Inspired Systems: Natures-Inspired Systems for Parallel, Asynchronous and Decentralised Environments, Bristol, April 2006.

[Cha04] David A. Chappell. *Enterprise Service Bus. Theory in Practice*. O'Reilly Media, 2004.

[Chr03] H.I. Christensen. Cognitive (vision) systems. *ERCIM News*, 53:17–18, April 2003.

[CJD+06] Antonello Ceravola, Frank Joublin, Mark Dunn, Julian Eggert, Marcus Stein, and Christian Goerick. Integrated research and development environment for real-time distributed embodied intelligent systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1631–1637. IEEE, 2006.

[Cla92] Herbert Clark. *Arenas of Language Use*. University of Chicago Press, 1992.

[CMG05] Toby H. J. Collett, Bruce A. MacDonald, and Brian Gerkey. Player 2.0: Toward a practical robot programming framework. In *Australasian Conference on Robotics and Automation*, Sydney, 5–7 December 2005.

[CNF01] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[Coc01] A. Cockburn. *Agile Software Development*. Addison-Wesley, 2001.

[Cog06] Cogniron Consortium. COGNIRON – The Cognitive Robot Companion, May 2006. `http://www.cogniron.org`.

[Cog08] Cogniron Consortium. COGNIRON Winter School on Human Robot Interaction (CWSHRI'08), January 2008.

[Cop91] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley Longman, 1991.

[COS04] COSY Consortium. Cognitive Systems for Cognitive Assistants. DR.11.1 Proc. Cognitive Systems Kick-Off meeting. CoSy-FP6-004250, dec 2004.

[CRTT97] N. Chleq, C. Regazzoni, A. Teschioni, and M. Thonnat. A visual surveillance system for the prevention of vandalism in metro stations. *EMMSEC'97*, 1997.

[Cru03] H. Cruse. The evolution of cognition- a hypothesis. *Cognitive Science: A Multidisciplinary Journal*, 27:135–155, 2003.

[CS01] Silvia Coradeschi and Alessandro Saffiotti. Perceptual Anchoring of Symbols for Action. In *Proc. Intl. Conf. on Artificial Intelligence*, pages 407–416, 2001.

[CSP03] M.K. Chandraker, C. Stock, and A. Pinz. Real Time Camera Pose in a Room. In *Int. Conf. on Computer Vision Systems*, volume 2626 of *LNCS*, pages 98–110, April 2003.

[CT04] John Cowan and Richard Tobin. XML Information Set (Second Edition), W3C Recommendation. Technical report, World Wide Web Consortium, Feb 2004. `http://www.w3.org/TR/2004/REC-xml-infoset-20040204`.

[CTS07] Jigna Chandaria, Graham A. Thomas, and Didier Stricker. The matris project: real-time markerless camera tracking for augmented reality and broadcast applications. *Journal of Real-Time Image Processing*, 2:69–79, 2007.

[CVS08] CVSSP, University of Surrey. Recognition And Vision Library, 2008. http://ravl.sourceforge.net.

[DBM88] U. Dayal, A. P. Buchmann, and D. R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database system. In *Lecture notes in computer science on Advances in object-oriented database systems*, pages 129–143. Springer-Verlag New York, Inc., 1988.

[DES08] DESIRE Consortium. DESIRE – Deutsche Service-Robotik-Initiative (German Initiative for Service Robotics), February 2008.

[DK76] Frank DeRemer and Hans Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, pages 321–327, 1976.

[DKRH94] B.A. Draper, G. Kutlu, E.M. Riseman, and A.R. Hanson. ISR3: Communication and Data Storage for an Unmanned Ground Vehicle. In *Proceedings International Conference on Pattern Recognition*, volume I, pages 833–836, 1994.

[DL99] T. DeMarco and T. Lister. *Peopleware: Productive Projects and Teams*. DORSET HOUSE PUBLISHING CO., INC, 1999.

[Dub08] Olivier Dubuisson. Asn.1 reference book. `http://www.oss.com/asn1/`, May 2008.

[EM02] Thomas Eiter and Viviana Mascardi. Comparing environments for developing software agents. *AI Communication*, 15(4):169–197, 2002.

[ER03] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories*. Addison-Wesley, Reading, MA, USA, 2003.

[ES99] W. Eckstein and C. Steger. The halcon vision system: An example for flexible software architecture. In *3rd Japanese Conference on Practical Applications of Real-Time Image Processing*, pages 18–23. Technical Committe of Image Processing Applications, Japanese Society for Precision Engineering, 1999.

[Eur01] European Commission. Work Programme 2002, Programme for Research, Technology Development and Demonstration under the Fifth Framework Programme, Nov 2001. `ftp://ftp.cordis.lu/pub/ist/docs/b_wp_en_200201.pdf`.

[Eur05] European Commission. Proposal for a Decision of the european parliament and of the council concerning the seventh framework programme of the European Community for research, technological development and demonstration activities (2007 to 2013), Apr 2005. `http://ica.cordis.lu/documents/documentlibrary/2461EN.pdf`.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[Fai06] Ted Faison. *Event-Based Programming: Taking Events to the Limit*. Apress, Berkeley, CA, May 2006.

[FB96] N. Freed and N. Borenstein. Multipurpose internet mail extensions. Technical report, Network Working Group, 1996.

[FFMM94] T. Finin, R. Fritzson, D. Mckay, and R. Mcentire. Kqml as an agent communication language. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, 1994.

[FHS04] J. Fritsch, N. Hofemann, and G. Sagerer. Combining Sensory and Symbolic Data for Manipulative Gesture Recognition. In *Proc. Int. Conf. on Pattern Recognition*, number 3, pages 930–933, Cambridge, United Kingdom, 2004. IEEE.

[Fia07] José Luiz Fiadeiro. Designing for software's social complexity. *IEEE Computer*, 40(1):34–39, 2007.

[FIG99] R. Fielding, UC Irvine, and J. Gettys. Hypertext transfer protocol – http/1.1. Technical report, Network Working Group, 1999.

[Fin99] G. A. Fink. Developing HMM-based Recognizers with ESMERALDA. In Václav Matoušek, Pavel Mautner, Jana Ocelíková, and Petr Sojka, editors, *Lecture Notes in Artificial Intelligence*, volume 1692, pages 229–234, Berlin Heidelberg, 1999. Springer.

[FJK+96] G.A. Fink, N. Jungclaus, F. Kummert, H. Ritter, and G. Sagerer. A Distributed System for Integrated Speech and Image Understanding. In *International Symposium on Artificial Intelligence*, pages 117–126, 1996.

[FKH+05] Jannik Fritsch, Markus Kleinehagenbrock, Axel Haasch, Sebastian Wrede, and Gerhard Sagerer. A Flexible Infrastructure for the Development of a Robot Companion with Extensible HRI-Capabilities. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 3419–3425, Barcelona, Spain, April 2005.

[FMN08] Paul Fitzpatrick, Giorgio Metta, and Lorenzo Natale. Towards long-lived robot genes. *Robotics and Autonomous Systems*, 56(1):29–45, 2008.

[Fra03] David S. Frankel. *Model Driven Architecture. Applying MDA to Enterprise Computing.* Wiley, 2003.

[FRF+02] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*, chapter Distribution Strategies, pages 87–94. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[FW07] Jannik Fritsch and Sebastian Wrede. *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*, chapter An Integration Framework for Developing Interactive Robots, pages 291–305. Springer, Berlin, 2007. ISBN: 978-3-540-68949-2.

[Gat07] Bill Gates. A robot in every home. *Scientific American*, pages 58–65, January 2007.

[GBB+06] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[Gep04] L Geppert. Qrio, the robot that could. *IEEE Spectrum*, 41:34–37, 2004.

[GHC+04] Nicolas Gorges, Marc Hanheide, William Christmas, Christian Bauckhage, Gerhard Sagerer, and Josef Kittler. Mosaics from Arbitrary Stereo Video Sequences. In C. E. Rasmussen, H. H. Bülthoff, M. A. Giese, and B. Schölkopf, editors, *Proceedings of the DAGM Symposium 2004*, volume 3175 of *Lecture Notes in Computer Science*, pages 342–349, Heidelberg, Germany, 2004. Springer-Verlag.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[GLF04] T. Gärtner, John W. Lloyd, and Peter A. Flach. Kernels and Distances for Structured Data. *Machine Learning*, 57(3):205–232, 2004.

[GMNR99] Martin Gudgin, Noah Mendelsohn, Mark Nottingham, and Hervé Ruellan. XML-binary Optimized Packaging, W3C Recommendation. Technical report, World Wide Web Consortium, Jan 1999. `http://www.w3.org/TR/2005/REC-xop10-20050125`.

[GR00] Charles F. Goldfarb and Yuri Rubinsky. *The SGML Handbook*. Clarendon Press, Oxford, 2000.

[Gra05] Gösta Granlund. Organization of architectures for cognitive vision systems. In Hans Hellmut Nagel and Henrik I. Christensen, editors, *Cognitive Vision Systems*, pages 39–58. Springer, Heidelberg, 2005.

[GT07] John Georgas and Richard Taylor. An architectural style perspective on dynamic robotic architectures. 2007.

[GVH03] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *ICAR 2003*, 2003.

[Han06] Marc Hanheide. *A Cognitive Ego-Vision System for Interactive Assistance*. PhD thesis, Technische Fakultät – Universität Bielefeld, December 2006.

[Hau99] D. Haussler. Convolution Kernels on Discrete Structures. Technical Report UCS-CRL-99-10, UC Santa Cruz, 1999.

[HBPM07] Jesse Hoey, Axel von Bertoldi, Pascal Poupart, and Alex Mihailidis. Assisting persons with dementia during handwashing using a partially observable markov decision process. In Gerhard Sagerer and Monique Thonnat, editors, *Proceedings of the 5th International Conference on Computer Vision Systems (ICVS)*, Bielefeld, Germany, March 2007. Bielefeld University.

[HBS04] Marc Hanheide, Christian Bauckhage, and Gerhard Sagerer. Memory Consistency Validation in a Cognitive Vision System. In *Proceedings International Conference on Pattern Recognition*, number 2, pages 459–462. IEEE, 2004.

[HBS05] M. Hanheide, C. Bauckhage, and G. Sagerer. Combining Environmental Cues & Head Gestures to Interact with Wearable Devices. In *Proc. of International Conference on Multimodal Interfaces*, 2005.

[Hoh06] Gregor Hohpe. Programmieren ohne Stack: ereignis-getriebene Architekturen. *OBJEKTspektrum*, 02:18–24, February 2006. in German, English version available at: `http://www.eaipatterns.com/docs/EDA.pdf`.

[Hoh07] Gregor Hohpe. Architect's dream or developer's nightmare? In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 188–188, New York, NY, USA, 2007. ACM.

[HS08a] Marc Hanheide and Gerhard Sagerer. Active memory-based interaction strategies for learning-enabling behaviors. In *Proceedings of the International Symposium on Robot and Human Interactive Communication (RO-MAN)*, Munich, August 2008.

[HS08b] Michi Henning and Mark Spruiell. *Distributed Programming with Ice*. ZeroC Inc., 2008.

[HW05] Erik Hollnagel and David D. Woods. *Joint cognitive systems : foundations of cognitive systems engineering*. CRC Press, 2005.

[HWLS08] Marc Hanheide, Sebastian Wrede, Christian Lang, and Gerhard Sagerer. Who am i talking with? a face memory for social robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Pasadena, CA, USA, 2008. IEEE, IEEE.

[HZW07] Nick Hawes, Michael Zillich, and Jeremy Wyatt. BALT & CAST: Middleware for cognitive robotics. In *Proceedings of IEEE RO-MAN 2007*, pages 998 – 1003, August 2007.

[IEE90] IEEE Standards Association. IEEE Std 610.12-1990, glossary of software engineering terminology. Technical report, IEEE, Dec 1990. Reaffirmed 2002, `http://ieeexplore.ieee.org/servlet/opac?punumber=2238`.

[IEE00] IEEE Architecture Working Group. IEEE Std 1471-2000, recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000.

[Int08] Intel Corporation. Open Source Computer Vision Library, 2008. Software and documentation available at `http://www.intel.com/technology/computing/opencv/`.

[Jac07] J. Jackson. Microsoft robotics studio: A technical introduction. *IEEE Robotics & Automation Magazine*, 14(4):82–87, Dec. 2007.

[Jen91] Kurt Jensen. Coloured petri nets: a high level language for system design and analysis. In *APN 90: Proceedings on Advances in Petri nets 1990*, pages 342–416, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Joh02] Pontus Johnson. *Enterprise Software System Integration: An Architectural Perspective*. PhD thesis, KTH, Royal Institute of Technology, 2002.

[KAU04] P. Kiatisevi, V. Ampornaramveth, and H. Ueno. A Distributed Architecture for Knowledge-Based Interactive Robots. In *Proc. Int. Conf. on Information Technology for Application (ICITA)*, pages 256–261, Harbin, China, 2004.

[KC04] Graham Klyne and Jerem J. Carrol. Resource Description Framework (RDF): Concepts and Abstract Syntax. Technical report, 2004.

[KCK07] Ilias Kolonias, William Christmas, and Josef Kittler. A layered active memory architecture for cognitive vision systems. In *Proceedings of the 5th International Conference on Computer Vision Systems*. Library of Bielefeld University, March 2007.

[KCkPK05] Gunhee Kim, Woojin Chung, Sung kee Park, and Munsang Kim. Experimental research of navigation behavior selection using generalized stochastic petri nets (gspn) for a tour-guide robot. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005)*, August 2005.

[KLP04] Rick Kjeldsen, Anthony Levas, and Claudio S. Pinhanez. Dynamically reconfigurable vision-based user interfaces. *International Journal of Machine Vision and Applications*, 16(1):6–12, 2004.

[KR94] K. Konstantinides and J. R. Rasure. The Khoros Software Development Environment For Image And Signal Processing. *IEEE Transactions on Image Processing*, 3(3):243–252, 1994.

[Kru95] Philippe Kruchten. Architectural blueprints - the "4+1" view model of software architecture. *IEEE Software*, 12 (6):42–50, 1995.

[KS04] Kristian Kvilekval and Ambuj K. Singh. Spree: Object prefetching for mobile computers. In *CoopIS/DOA/ODBASE (2)*, pages 1340–1357, 2004.

[KS06] James Kramer and Matthias Scheutz. ADE: A framework for robust complex robotic architectures. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4576–4581, Bejing, China, October 2006.

[KS07] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Auton. Robots*, 22(2):101–132, 2007.

[Lad94] Scott Robert Ladd. *Defensive Programming with C++*. John Wiley & Sons Inc, 1994.

[Lan01] A. Langer. Java programming idioms. *Technology of Object-Oriented Languages and Systems, 2001.*, pages 197 – 198, 2001.

[Lan07] Christian Lang. Personenidentifikation mit active appearance models. Master's thesis, Bielefeld University, 2007. in german.

[LBF+05] Thor List, José Bins, Robert B. Fisher, David Tweed, and Kristinn R. Thórisson. Two approaches to a plug-and-play vision architecture - caviar and psyclone. In *AAAI-05 Workshop on Modular Construction of Human-Like Intelligence*, Pittsburgh, PA, July 2005. Twentieth Annual Conference on Artificial Intelligence.

[LBFT05] Thor List, José Bins, Robert B. Fisher, and David Tweed. A Plug-and-Play Architecture for Cognitive Video Stream Analysis. In *Seventh International Workshop on Computer Architectures for Machine Perception*, pages 67–72, Palermo, Italy, Jul 2005. IEEE Computer Society.

[LF04] T. List and R.B. Fisher. CVML - an XML-based computer vision markup language. In *Proceedings of the 17th International Conference on Pattern Recognition, ICPR 2004.*, volume 1, pages 789–792. IEEE, IEEE, August 2004.

[LFP99] Yannis Labrou, Tim Finin, and Yun Peng. Agent communication languages: The current landscape. *IEEE Intelligent Systems*, 14:45–52, 1999.

[LHW+05] S. Li, A. Haasch, B. Wrede, J. Fritsch, and G. Sagerer. Human-style interaction with a robot for cooperative learning of scene objects. In *Proc. Int. Conf. on Multimodal Interfaces*, pages 151–158, Trento, Italy, 2005. ACM Press.

[Löm08] Frank Lömker. iceWing – A graphical plugin shell, 2008. `http://icewing.sf.net`.

[Lüt04] Ingo Lütkebohle. An active memory for cognitive processes. Projektarbeit, Bielefeld University, Faculty of Technology, July 2004.

[LW04] Ingo Lütkebohle and Sebastian Wrede. Catwalk. Einsatz der XML-Datenbank Berkeley DB XML. *iX, Heise*, 9/2004:68–73, 2004.

[LW07] Shuyin Li and Britta Wrede. Why and how to model multi-modal interaction for a mobile robot companion. In *AAAI Technical Report SS-07-04: Interaction Challenges for Intelligent Assistants*, pages 71 – 79, Stanford, 2007. AAAI Press.

[LWHF06] Frank Lömker, Sebastian Wrede, Marc Hanheide, and Jannik Fritsch. Building Modular Vision Systems with a Graphical Plugin Environment. In *Proc. of International Conference on Vision Systems*, St. Johns University, Manhattan, New York City, USA, January 2006. IEEE.

[LWS06] Shuyin Li, Britta Wrede, and Gerhard Sagerer. A dialog system for comparative user studies on robot verbal behavior. In *Proceedings on the 15th International Symposium on Robot and Human Interactive Communication*, pages 129–134, Hatfield, United Kingdom, September 2006. IEEE, IEEE Press.

[LWT94] Christopher Lindblad, David Wetherall, and David L. Tennenhouse. The VuSystem: A Programming System for Visual Processing of Digital Video. In *ACM Multimedia*, pages 307–314, 1994.

[Lö04] Frank Lömker. *Lernen von Objektbenennungen mit visuellen Prozessen*. PhD thesis, Universität Bielefeld, Technische Fakultät, 2004.

[MBCC+05] F. Michaud, Y. Brosseau, C. C. Côté, D. Letourneau, P. Moisan, A. Ponchon, C. Raievsky, J.-M. Valin, E. Beaudry, and F. Kabanza. Modularity and integration in the design of a socially interactive robot. In *IEEE International Workshop on Robot and Human Interactive Communication, ROMAN.*, pages 172–177. IEEE, August 2005.

[MC05] Rene Meier and Vinny Cahill. Taxonomy of Distributed Event-Based Programming Systems. *The Computer Journal*, 48(5):602–626, 2005.

[McC04] Steve McConnell. *Code Complete, Second Edition: A Practical Handbook of Software Construction*. Microsoft Press, June 2004.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, Inc., 1997.

[MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[MGS+07] Thomas Michalke, Alexander Gepperth, Martin Schneider, Jannik Fritsch, and Christian Goerick. Towards a human-like vision system for resource-constrained intelligent cars. In Gerhard Sagerer and Monique Thonnat, editors, *The 5th International Conference on Computer Vision Systems (ICVS)*. Bielefeld University, March 2007.

[Mic08] Microsoft. Distributed component object model (dcom) remote protocol specification. Technical report, Microsoft corporation, 2008.

[Min86] Marvin Minsky. *The society of mind*. Simon & Schuster, Inc., 1986.

[Min07] Mindmakers.org. OpenAIR: Specification and reference implementations, August 2007. `http://mindmakers.org/mindmakers/openair`.

[MK02] Holger Meyer Maike Klettke. *XML & Datenbanken. Konzepte, Sprachen und Systeme*. Dpunkt Verlag, December 2002.

[MLB07] Mark W. Maimone, P. Chris Leger, and Jeffrey J. Biesiadecki. Overview of the mars exploration rovers' autonomous mobility and vision capabilities. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) Space Robotics Workshop*, Rome, Italy, April 2007.

[MPV00] Luis Montano, Francisco José García Peñalvo, and José Luis Villarroel. Using the time petri net formalism for specification, validation, and code generation in robot-control applications. *I. J. Robotic Res.*, 19(1):59–76, 2000.

[MRB03] Asa MacWilliams, Thomas Reicher, and Bernd Brügge. Decentralized coordination of distributed interdependent services. In *IEEE Distributed Systems Online – Middleware '03 Work in Progress Papers*, Rio de Janeiro, Brazil, June 2003.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. Owl web ontology language overview. Technical report, The World Wide Web Consortium (W3C), 2004.

[MvNV+01] J. Maassen, R. van Nieuwpoort, Ronald Veldema, H.E. Bal, T. Kielmann, C. Jacobs, and R. Hofmann. Efficient java rmi for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[NC07] Henrik Frystyk Nielsen and George Chrysanthakopoulos. Decentralized software service protocol. Technical report, Microsoft Corporation, 2007.

[Net87] Network Working Group. RFC 1034: Domain names - concepts and facilities. Technical report, The Internet Engineering Task Force, 1987.

[Neu94] B. Clifford Neuman. *Scale in Distributed Systems*, pages 463–489. IEEE Computer Society, Los Alamitos, CA, 1994.

[Neu04] Bernd Neumann. Cognitive Vision - Remarriage of Computer Vision and AI? *KI*, 18(1):47–49, 2004.

[NL04] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services*. Addison-Wesley Professional, 2004.

[NSSK90] H. Niemann, G. Sagerer, S. Schröder, and F. Kummert. Ernest: A semantic network system for pattern understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:883–905, 1990.

[OAS06] OASIS. Reference model for service oriented architecture 1.0. Technical report, OASIS, 2006.

[OAS07] OASIS Open Composite Services Architecture (CSA) Member Section. Sca assembly model specification v1.00. Technical report, OASIS Open Composite Services Architecture (CSA) Member Section, 2007.

[Obj00] Object Data Management Group (ODMG). *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[OEF+06] Ott, O. Eiberger, W. Friedl, B. Bäuml, U. Hillenbrand, Ch. Borst, A. Albu-Schäfer, B. Brunner, H. Hirschmüller, S. Kielhöfer, R. Konietschke, M. Suppa, T. Wimböck, F. Zacharias, and Gerhard Hirzinger. A humanoid two-arm system for dexterous manipulation. *IEEE-RAS International Conference on Humanoid Robots*, pages 276 – 283, 2006.

[(OM08] Object Management Group (OMG). Robotics domain task force. `http://www.omg.org/robotics/`, May 2008.

[Ore99] Anders Orebäck. Components in Intelligent Robotics. Technical report, Royal Institute of Technology, KTH Stockholm, 1999.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053 – 1058, 1972.

[PB04] Robert Dupuis Pierre Bourque, editor. *SWEBOK*. American National Standards Institute (ANSI), 2004.

[Per08] Carlos E. Perez. SOA rediscovering modularity. `http://www.manageability.org`, May 2008.

[Pet81] James Lyle Peterson. *Petri Net Theory and The Modeling of Systems*. Prentice Hall, Inc., Englewood Cliffs, Massachusetts,, 1981.

[Pet05] Peter Tabeling. *Softwaresysteme und ihre Modellierung: Grundlagen, Methoden und Techniken*. Springer Verlag, Berlin, 2005.

[PPV00] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 345–355, New York, NY, USA, 2000. ACM.

[PUV03] W. Ponweiser, G. Umgeher, and M. Vincze. A Reusable Dynamic Framework for Cognitive Vision Systems. In *Workshop on Computer Vision System Control Architectures*, Graz, 2003. In conjunction with ICVS2003.

[PVWB04] Wolfgang Ponweiser, Markus Vincze, Sebastian Wrede, and Christian Bauckhage. Overview of software frameworks for use in cognitive vision approaches. Technical report, EC Vision, Network of Excellence, 2004. ECVision Specific Action 13-2, `http://www.ecvision.info`.

[PVZ05] Wolfgang Ponweiser, Markus Vincze, and Michael Zillich. A software framework to integrate vision and reasoning components for cognitive vision systems. *Robotics and Autonomous Systems*, 52:101–114, July 2005. Advances in Robot Vision.

[RBP04] M. Ribo, M. Brandner, and A. Pinz. A flexible software architecture for hybrid tracking. *Journal of Robotics Systems*, 21(2):53–62, 2004.

[RHS07] Helge Ritter, Robert Haschke, and Jochen J. Steil. *Perspectives of Neural-Symbolic Integration*, chapter A Dual Interaction Perspective for Robot Cognition: Grasping as a Rosetta Stone. Computational Intelligence. Springer, 2007.

[Rit] H. Ritter. The graphical simulation toolkit neo/nst. http://www.TechFak.Uni-Bielefeld.DE/ags/ni/projects/simulation_and_visual/neo/neo_e.html.

[RoS08] RoSta. Robot standards and reference architectures, 2008. http://www.robot-standards.eu/.

[Roy87] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, 1987.

[RPJ+03] K. Runapongsa, J.M. Patel, H.V. Jagadish, Y. Chen, and S. Al-Khalifa. The michigan benchmark: Towards XML query performance diagnostics. In *Proc. VLDB Conference*. Morgan Kaufmann, 2003.

[RRH99] A. Rares, M.J.T. Reinders, and E.A. Hendriks. Mapping Image Analysis Problems on Multi-Agent-Systems. Technical report, Information and Communication Theory Group, TU Delft, Nov 1999.

[RTF00] R. Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[SC97] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. *compsac*, 00:6, 1997.

[Sch06a] Christian Schlegel. Communication patterns as key towards component-based robotics task. *International Journal of Advanced Robotic Systems*, 3(1):049–054, 2006.

[Sch06b] Douglas Schmidt. Real-Time CORBA programming with TAO (The ACE ORB), 2006. `http://siesta.cs.wustl.edu/~schmidt/TAO.html`.

[Sch08] Jan Schaefer. Visualization and interaction server - a user interaction service for a cognitive system. Technical report, Bielefeld University, 2008.

[Sea99] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Enginering*, 25:557–572, 06/1999 1999.

[SEI08] Carnegie Mellon University Software Engineering Institute. Uls systems glossary. `http://www.sei.cmu.edu/uls/glossary.html`, Mai 2008.

[SFH+00] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. ari, L. Encarnac, a Gervautz, and W. Purgathofer. The studierstube augmented reality project. Technical report, Vienna University of Technology, 2000.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[SH01] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison Wesley Professional, 2001.

[SHH+08] Joachim Schmidt, Nils Hofeman, Axel Haasch, Jannik Fritsch, and Gerhard Sagerer. Interacting with a mobile robot: Evaluating gestural object references. Nice, France, September 2008.

[SHS07] T.P. Spexard, M. Hanheide, and G. Sagerer. Human-oriented interaction with an anthropomorphic robot. *IEEE Transactions on Robotics*, 23:852–862, 2007.

[SHWP07] H. Siegl, M. Hanheide, S. Wrede, and A. Pinz. An augmented reality human-computer interface for object localization in a cognitive vision system. *Image and Vision Computing, Special Issue on The Age of Human-Computer-Interaction*, 25(12):1895–1903, December 2007.

[Sie00] Jon Siegel. *CORBA 3. Fundamentals and Programming*. John Wiley & Sons, Inc., 2000.

[Sie08] Frederic Siepmann. Refactoring der systemarchitektur eines mobilen roboters für die multi-modale mensch-roboter interaktion. Diplomarbeit, Bielefeld University, Bielefeld, Germany, March 2008.

[Sim08] CMLabs Simulations. Vortex training simulators, 2008. `http://www.vortexsim.com/`.

[SK98] Ian Sommerville and Gerald Kotonya. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[Sle06] Sleepycat Software. Berkely DB XML, 2006. `http://www.sleepycat.com/products/xml.shtml`.

[Slo98] Aaron Sloman. Damasio, descartes, alarms and meta-management. *In Proceedings IEEE Conference on Systems, Man, and Cybernetics*, 3:2652–2657, 1998.

[SM99] Douglas C. Schmid Sumedh Mungee, Nagarajan Surendran. The design and performance of a corba audio/video streaming service. In *HICSS '99: Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 8*, page 8043, Washington, DC, USA, 1999. IEEE Computer Society.

[SMC99] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 38:231–256, 1999.

[Som01] Ian Sommerville. *Software Engineering*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 6th edition, 2001.

[SP04] H. Siegl and A. Pinz. A Mobile AR kit as a Human Computer Interface for Cognitive Vision. In *Int. Workshop on Image Analysis for Multimedia Interactive Services*, Lissabon, 2004.

[SSP00] G. Socher, G. Sagerer, and P. Perona. Bayesian reasoning on qualitative descriptions from images and speech. *Image and Vision Computing*, 18:155–172, 2000.

[SSRB00] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2: Patterns for Concurrent and Networked Objects. John Wiley & Sons Ltd., 2000.

[SSS08] Thorsten P. Spexard, Frederic H. K. Siepmann, and Gerhard Sagerer. A memory-based software integration for development in autonomous robotics. In *Proceedings of the 10th International Conference on Intelligent Autonomous Systems Intelligent Autonomous Systems*, number 10, Baden-Baden, Germany, 2008. IOS-Press.

[SST86] Steven Shafer, Anthony (Tony) Stentz, and Chuck Thorpe. An architecture for sensor fusion in a mobile robot. Technical Report CMU-RI-TR-86-09, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, April 1986.

[Sun99] Sun Microsystems, Inc. Jini technology architectural overview. Technical report, Sun Microsystems, Inc., 1999. `http://www.sun.com/software/jini/whitepapers/architecture.html`.

[Sun08] Sun Microsystems, Inc. Jini network technology. WWW, Feb 2008. `http://java.sun.com/developer/products/jini/index.jsp`.

[SV01] D. C. Schmidt and S. Vinoski. Object Interconnections: CORBA and XML, Part 1: Versioning. *C/C++ Users Journal*, May 2001. `http://www.cs.wustl.edu/~schmidt/report-doc.html`.

[TB01] Zahir Tari and Omran Bukhres. *Fundamentals of Distributed Object Systems*. Wiley-Interscience, 2001.

[TBF+06] D. Thirde, M. Borg, J. Ferryman, F. Fusier, V. Valentin, F. Bremond, and M. Thonnat. A real-time scene understanding system for airport apron monitoring. In *Proceedings of the IEEE International Conference on Computer Vision Systems (ICVS '06)*, pages 26–26, January 2006.

[The05] The ActIPret Consortium. The ActIPret Project, 2005. IST-2001-32184, `http://robsens.acin.tuwien.ac.at/actipret/`.

[The08] The Mathworks, Inc. The MATLAB Image Processing Toolbox, 2008. Software and documentation available at `http://www.mathworks.com/`.

[Tho05] Thomas Freytag. WoPeD - workflow petri net designer. Technical report, University of Cooperative Education (Berufsakademie) Karlsruhe, 2005.

[TLPD05] Kristinn R. Thórisson, Thor List, Christopher Pennock, and John DiPirro. Whiteboards: Scheduling blackboards for semantic routing of messages & streams. In *AAAI-05 Workshop on Modular Construction of Human-Like Intelligence*, Pittsburgh, PA, July 2005. Twentieth Annual Conference on Artificial Intelligence.

[TM98] E. Tulving and H. J. Markowitsch. Episodic and declarative memory: role of the hippocampus. *Hippocampus*, 3:198–204, 1998.

[TM08] Inc. The MathWorks. Matlab. Technical report, The MathWorks, Inc., 2008.

[TMA+95] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr. E. James Whitehead, and Jason E. Robbins. A component- and message-based architectural style for gui software. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, 1995.

[TMD+06] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney. Stanley: The robot that won the darpa grand challenge. *Journal of Field Robotics*, 23(9):661–692, September 2006.

[TPLD04] K. R. Thórisson, C. Pennock, T. List, and J. DiPirro. Artificial intelligence in computer graphics: A constructionist approach. *Computer Graphics Quarterly, ACM SIGGRAPH*, 38:26–30, 2004.

[TS08] TC-SOFT. Technical comittee on software engineering for robotics and automation, 2008. http://robotics.unibg.it/tcsoft/index.htm.

[USEK02] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *Robotics and Automation, IEEE Transactions on*, 18(4):493–497, Aug 2002.

[VAM04] VAMPIRE Consortium. VAMPIRE: Visual Active Memory Processes and Interactive Retrieval, Annex 1 - "Description of Work". IST-2001-34401, sep 2004.

[VAM06] VAMPIRE Consortium. VAMPIRE: Visual Active Memory Processes and Interactive Retrieval, Feb 2006. IST-2001-34401, `http:/www.vampire-project.org`.

[Ver91] David Vernon. *Machine vision: automated visual inspection and robot vision*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Ver04] David Vernon. European Research Network for Cognitive Computer Vision Systems, Mar 2004. `http://www.ecvision.info`.

[Ver08] David Vernon. Cognitive vision: The case for embodied perception. *Image Vision Computing*, 26(1):127–140, 2008.

[VJ01] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 Conference on Computer Vision and Pattern Recognition*, volume 1, pages 511–518, Kauai, Hawaii, December 2001.

[VNE+00] Richard Volpe, Issa A.D. Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. Claraty: Coupled layer architecture for robotic autonomy. Technical report, Jet Propulsion Laboratory, California Institute of Technology, 2000.

[vR04] Guido van Rossum. Extending and embedding the python interpreter, release 2.3.4. `http://docs.python.org/ext/ext.html`, 2004. last checked 05/30/2008.

[vR08] Guido van Rossum. Python/c api reference manual, release 2.5.2. `http://docs.python.org/api/api.html`, 2008. last checked 05/30/2008.

[Vxl08] C++ libraries for computer vision research and implementation, Jan 2008. http://vxl.sourceforge.net/.

[WA01] Jim Waldo and Ken Arnold. *The Jini specifications*. Jini technology series. Addison-Wesley, Reading, MA, USA, second edition, 2001.

[Wal05] P. Wallich. Tools & toys: I, roboticist. *IEEE Spectrum*, 42(10):63–65, October 2005.

[WFBS04] Sebastian Wrede, Jannik Fritsch, Christian Bauckhage, and Gerhard Sagerer. An XML Based Framework for Cognitive Vision Architectures. In *Proceedings International Conference on Pattern Recognition*, number 1, pages 757–760, 2004.

[WHWS06] Sebastian Wrede, Marc Hanheide, Sven Wachsmuth, and Gerhard Sagerer. Integration and Coordination in a Cognitive Vision System. In *Proceedings of International Conference on Computer Vision Systems*, St. Johns University, Manhattan, New York City, USA, 2006. IEEE.

[WJ05] R. E. Wray and Jones. An introduction to soar as an agent architecture. In *In Cognition and Multi-Agent Interaction: From Cognitive Modeling to So- cial Simulation, ed. R. Sun.*, 2005.

[WK03] Michael Weber and Ekkart Kindler. The petri net markup language. In *Petri Net Technology for Communication Based Systems.*, LNCS 2472. Springer-Verlag, 2003.

[WKF07] Britta Wrede, Marcus Kleinehagenbrock, and Jannik Fritsch. Towards an integrated robotic system for interactive learning in a social context. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems - IROS 2006*, Bejing, 2007.

[WL08] Sebastian Wrede and Ingo Lütkebohle. Integration expertise in hri research: A first study. In *ICRA'08 Workshop on Software Engineering for Robotics III*. IEEE RAS TC-SOFT, May 2008.

[WMLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehmann, and D. A. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[WPB+04] Sebastian Wrede, Wolfgang Ponweiser, Christian Bauckhage, Gerhard Sagerer, and Markus Vincze. Integration Frameworks for Large Scale Cognitive Vision Systems - An Evaluative Study. In *Proceedings International Conference on Pattern Recognition*, number 1, pages 761–764, 2004.

[WWH06] Sven Wachsmuth, Sebastian Wrede, and Marc Hanheide. Coordinating Interactive Vision Behaviors for Cognitive Assistance. *Computer Vision and Image Understanding*, 2006.

[WWHB05] Sven Wachsmuth, Sebastian Wrede, Marc Hanheide, and Christian Bauckhage. An Active Memory Model for Cognitive Computer Vision Systems. *KI-Journal, Special Issue on Cognitive Systems*, 19(2):25–31, 2005.

[WWWK97] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64, London, UK, 1997. Springer-Verlag.

[Yos04] Sakagami Yoshiaki. Intelligent function of humanoid robot asimo-system and integration of vision auditory behavior. *Journal of the Society of Automotive Engineers of Japan*, 58:22–27, 2004.

[Zer06] ZeroC Inc. The Internet Communications Engine, ZeroC Inc., 2006. `http://www.zeroc.com/ice.html`.

[ZGN04] T. Zinßer, Ch. Gräßl, and H. Niemann. Efficient Feature Tracking for Long Video Sequences. In C. E. Rasmussen, H. H. Bülthoff, M. A. Giese, and B. Schölkopf, editors, *Pattern Recognition, 26th DAGM Symposium*, volume 3175 of *LNCS*, pages 326–333, Tübingen, Germany, September 2004. Springer-Verlag, Berlin, Heidelberg, New York.

List of Figures