Researching and developing a real-time infrastructure for intelligent systems – Evolution of an integrated approach

Antonello Ceravola, Christian Goerick

2008

Available online at www.sciencedirect.com

ScienceDirect

Robotics and Autonomous Systems 56 (2008) 14–28

Robotics and Autonomous Systems

www.elsevier.com/locate/robot

Researching and developing a real-time infrastructure for intelligent systems — Evolution of an integrated approach

Antonello Ceravola*, Marcus Stein, Christian Goerick

Honda Research Institute Europe GmbH, Carl-Legien-Strasse 30, 63073 Offenbach/Main, Germany

Available online 29 September 2007

Abstract

In this paper, we describe the principles and the methodologies that we have researched for the creation of a software infrastructure for bridging the gap from brain-like systems design to standard software technology. Looking at the brain, we constantly take inspiration and choose the relevant principles that our computer-base model should/could be based on. This ranges from the evolution of the brain (phylogenetically and ontogenetically), the inherent autonomy of the currently identified areas, the intrinsic synchronization through the most basic control mechanisms that regulates interaction, communication, and modulation. With these principles in mind, we started to make a subdivision of our system into *instance*, *functional* and *computing architecture*, modeling each sub-system with processes and tools in order to create a basic infrastructure that supports the research and creation of intelligent systems. The basic elements of our infrastructure are the BBCM (Brain Bytes Component Model) and BBDM (Brain Bytes Data Model), created to enable the modularization and reuse of our systems. Based on those, we have developed DTBOS (Design Tool for Brain Operating System), the design environment for supporting graphical design, RTBOS (Real-Time Brain Operating System), the middleware that supports real-time execution of our modular systems, and CMBOS (Control-Monitor Brain Operating System) to enable the monitoring of running modules. We will show the feasibility of the established environment by shortly describing some of the experimental systems in the area of cognitive robotics that we have created. This will serve to give a more concrete understanding of the dimensions and the type of systems that we have been able to create.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Real-time; Modularity; Multiprocessing; Multithreaded; Parallel; Middleware; Integration

1. Introduction

The main focus of our research is to create large scale biologically inspired intelligent systems capable of interaction in the real world with real-time constraints. The underlying assumptions are that major progress in brain-like computing can only be achieved if the systems aspect is a central point of research and that the real power of brain-like computing will only unfold if real world tasks are approached. The general concept is not defined as copying the brain, but as understanding and transferring brain-like processing principles to technical systems in order to solve problems of computational intelligence. Therefore, great care has to be taken in order to transfer the principles of biological processing and not the constraints present in biological systems. Such principles constitute the main focus of our research, and, as proof of concept, we implement them on our target platforms (ASIMO and automobile). With such a process, we try to enable fundamental research to deal with problems and allow usage of methods that may result in concrete applications realizable in real world conditions.

The notion of a system comprises sensing, processing, and the generation of behavior in the real world by means of actuators, which close the loop to the sensors via the environment. We demand also a sufficient generality of the system with respect to the solution of several different tasks without a separate system-design for each task. Therefore, the spectrum of research topics ranges from biological models and processes via abstract system-theoretical aspects, simulation, and real-time processing environments to the engineering of hardware and software computing elements (see [23]). A large influence on the design of such intelligent systems comes from

* Corresponding author.
E-mail address: Antonello.Ceravola@honda-ri.de (A. Ceravola).

0921-8890/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.robot.2007.09.015

the area of real-time computing, since the systems must interact with the outside world by means of sensors and actuators. This demand for real-time execution exists in the sense of a defined upper limit of the time span from an (environmental) event to a system response. It is our strong belief that all those aspects have to be addressed concurrently in order to design a consistent overall system.

Since the knowledge of biological neural systems is still continuously growing, an iterative analysis by synthesis approach is pursued:

- Start the incremental design from the general to the specific;
- Observe and analyze the behavior of the system;
- Condense the gained knowledge into an increasingly dedicated architecture.

In order to research brain-like systems, a critical mass of architecture, infrastructure, and implementation has to be established. This will allow previously isolated concepts about some functional mechanisms in the brain to be tested in a comprehensive environment.

Standard computers and programming languages are, implicitly or explicitly, accepted as a general platform for researching and creating intelligent systems (very few researchers are building their own computing architectures like analog or neural computers). Our assumption here is that such technologies are sufficiently general and powerful, and that they do not represent a limitation for the targeted brain-like computing system we aim to build. But still, a set of generic tools and computers alone cannot solve all the issues that usually emerge in researching and creating intelligent systems. Current technologies, in the states are now available, are still too complex to be used for a large spectrum of researchers working in fields and having knowledge not closely related to the computer science domain. It is clearly necessary to research the hardware/software architectures and the methods and approaches towards the creation of such large-scale systems through a general research and development environment. Here, the effort is on the principles that regulate and organize the research work into a coherent and consistent design and implementation of the many parts that an intelligent system is composed of. Here, libraries, modules, components, and software infrastructures are necessary elements. Systems with a basic set of behaviors are also an indispensable part. But still, substantial work has to be carried out in defining a common set of data types, thereby identifying the language that the module's interface would use. With it, the way towards representation will be pursued. Additionally, a set of basic synchronization principles has to be defined, governing the data flow of a complex system: when does a signal trigger a computation? When is it only modulatory? How many inputs are necessary for starting the computation of a module? How to align parallel streams of data?

We are focusing on a research environment with seemingly contradictory requirements:

- Provide the necessary freedom for exploration, but
- make an interactive evaluation of the research possible.

This means that issues like synchronization, communication, granularity etc. are not determined a priori, but must be realizable in different forms within the environment.

The lessons we learned from the brain are as follows (some of them are not addressed for the first time, but according to our understanding they have not been addressed so coherently and comprehensively so far):

(1) The brain has been modeled as several areas that act autonomously but can be coherently processing when necessary. As a consequence, we assume basically all processing units as autonomous, and allow for a synchronization by events, data streams, or grouping of processing units. Even more radically, our basic communication framework guarantees that units cannot block each other in processing, but data can always be read and written.
(2) Our traditional understanding of units may finally not be applicable to the brain. Nevertheless, we start with some units (while acknowledging that the boundaries of those units may dissolve). As a consequence, we provide dedicated communication means at low computational cost for handling the massive amount of data to be communicated. This is done by considering interfaces between units that can become bigger and bigger. The technical requirements here are high bandwidth with low latency communication.
(3) Developmental or incremental design: We pursue an analysis by the synthesis approach, which means that our environment has to evolve with the knowledge gained from the research results. It resembles, to some extent, the processes of phylogeny and ontogeny in biology. As a consequence, we provide only very basic means for organizing processing and communication, but allow for adding more mechanisms on top of those means. This underlying computing architecture should not impose constraints on the functional model of the systems architecture to be researched (see [11,7]).
(4) The brain is the most complex structure known to man. Researching brain-like intelligence means that we have to be able to cope with large-scale systems. This applies to the design, to the execution, and to the evaluation of the experiments. There are three major consequences:
    - *Design*: GUIs and means for defining hierarchies are available now. Means for growing systems are under evaluation.
    - *Execution*: Deterministic distributed execution of large-scale systems (see [32,22] for an overview on distributed systems).
    - *Evaluation*: Means for observing large-scale systems with minimal interference with the processing inside these systems.
(5) The brain is embedded in the real world and organizes interactions. We consider interaction in a sufficiently rich environment as crucial in order to ask the relevant research questions. As a consequence, our environment must support this interaction, which essentially means that it must be predictable in execution time. This is a major difference

to other biologically oriented approaches, though of course not to traditional robotics oriented ones.

(6) In the brain, the communication determines the control flow. How this works exactly is unknown. Therefore, we need to be able to investigate various communication topologies and patterns without changing any local processing. As a consequence, we introduced a strict separation between processing and communication.

(7) Researchers should be able to experiment on the conceptual level without the burden of the concrete technical implementation. This was already partially addressed in the first point. As a consequence here, we consider an abstraction layer above the mechanisms for the necessary parallelization and synchronization.

These points are the main driving forces for the work described in this paper. They will be referred to throughout this document when we describe our contributions to a research infrastructure. The paper is organized along the structure of the overall process we use in our software system research and the tools that support us in the creation of real-time distributed embodied intelligent systems.

We have decided to support the research with the standard software engineering process: design, development, and testing. This means that for every cycle we may review our systems (see (3) in Section 1), consolidate and make a more coherent design and implementation (following an evolutionary approach – monotonically – and reusing the old part of the systems). The need for such a process comes from the inherent complexity of cutting an overall system into pieces (modules). In identifying the atomic parts and modeling them into software components for algorithms and components for data (see (6) in Section 1), we have made a clear distinction between processing and communication (see (6) in Section 1). All these concepts have been encapsulated into a uniform and fully integrated framework, reaching an infrastructure that is undoubtedly not common in computer science, and especially in the robotics domain.

2. Related approaches

Researchers are currently sharing the necessity of basing their research on a more structured software process, where it is possible to reuse and maintain large systems that can grow over time, instead of rebuilding a system from scratch when new principles or new functionalities need to be tested (without reusing much, in most cases nothing, from the old systems). In recent years, in the field of intelligent systems, robotics, and real-time systems, researchers have started to build software environments like the one we describe in this paper. Such a need comes from the assumption that the creation of intelligence requires a scientific and engineering effort focused on the same targets [19].

The approaches vary mainly in their abstraction levels and the pursued targets (see [35] for a comprehensive description of the main benefits of modular and dynamic systems). On a basic abstraction level, a modular approach is proposed by [24,33], which uses a component model (software packaging

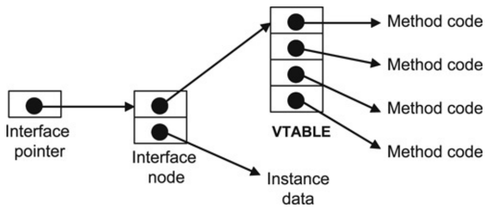

Fig. 1. Typical format of COM interface nodes (picture taken from [24]).

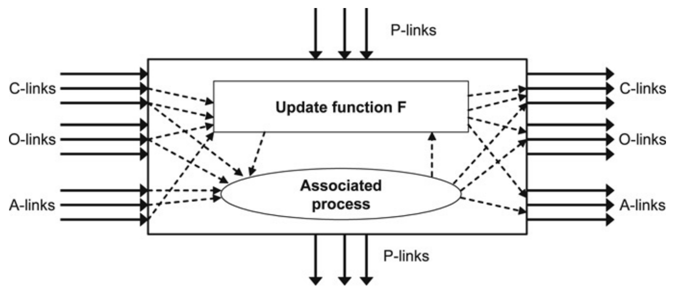

Fig. 2. Basic structure of an APOC component (picture taken from [33]).

patterns, [13]). In [24] Lüders refers to Component-Based Software Engineering (CBSE) as the base for the inspiration for achieving modularity. He reviews some of the widely used component models like: JavaBeans, ActiveX, Component Object Model, and CORBA. The main focus here is the applicability of such a component model to real-time systems. In this context, the major constraints are the possible overheads introduced by such a component model in terms of execution predictability and memory consumption. For the same reasons, we decided to base our BBCM/BBDM component model on the principles coming from CBSE. On the other hand, we did not adopt an existing component model. We decided to create our own in order to minimize overheads through: (a) a component interface that does not require any additional computation other than the functionality the user wants to encapsulate into the component (execution predictability); (b) defining input/output communication with components through pointers in order to avoid unnecessary memory copying (memory overhead).

In [24], several benefits gained through the adoption of a component model are highlighted, including the possibility of having a more intensive module-test phase, where a single component can be separated from the rest of the system and tested in a confined test environment. Also, the dynamic instantiation is facilitated due to the property of components having a well defined interface (see COM interface in Fig. 1). This property becomes very important once components are automatically handled by a middleware or execution environment.

In [33], the problem of organizing a behavior-based system is analyzed by means of using the APOC framework (Activating – Processing – Observing – Components). In such an architecture, the concept of behavior is mapped to the definition of a component. A component can represent a single behavior; and the links between components represents the information and control flow among behaviors. In Fig. 2 the basic structure of a component is represented.

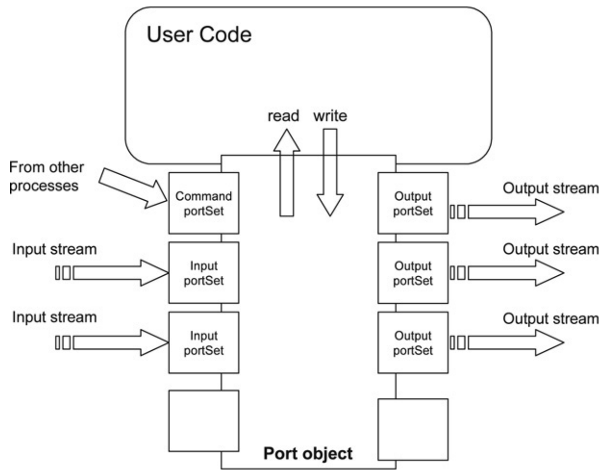

Fig. 3. Structure of a YARP port concept (picture taken from [16]).

Several types of input/output connections are implemented in the APOC component model: A-Link (activation links, transferring information within a time interval); O-Link (observe link, observe state of other components); C-Link (instantiate component or link, run-time creation of new components or new connections). In our BBCM/BBDM component model, we decided to implement only two type of links: (a) data links (links for data values, like an integer or an image; need a pointer to the data); (b) event links (links that simply trigger an event; need a pointer to a function). These two types of links are sufficiently general for covering the types of interaction a component can have with other components or with an external context. A similar concept of port-based interfaces can be found in [5]. Taken from the way a port is defined in CORBA 3.0, the concept of port has been realized with more attention on performance through the creation of *providers/uses* interfaces. These applications are parallel in a data-flow context. For a more analytical analysis of the component performance, see [37].

In [10], the system Player/Stage is composed by a set of libraries that support data communication. Even if communication is one of the most important issues in modular systems, especially when building component-based systems, Player/Stage exports such functionality as a set of libraries. The focus of Player/Stage is more on the definition of drivers for the support of several sensors and effectors than on the actual support of a component-based infrastructure.

A similar approach is followed by YARP [16], which proposes a library that implements a data communication model. The emphasis here is on modeling a concept of input/output ports that handle several data types and buffering methodologies (see Fig. 3). Ports are represented by a triplet consisting of an IP address, a port, and an interface name. Through a name server, the user can access the port directly by the interface name. In addition to the YARP library, in [16] the usage of image/matrix manipulation libraries in combination with their development systems is proposed. The advantage here is that components/modules can benefit from a low level library that may give a significant acceleration for common image/sound manipulation or matrix operations.

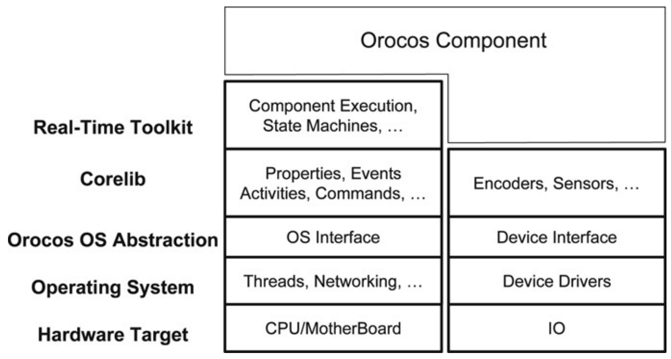

Fig. 4. Real-time toolkit layers (picture taken from [34]).

Approaches which are closer to the one that we propose in this paper, are described in [8,31,26,34,36,28], where the aim is placed on an integrated environment which includes designing and monitoring tools. In these papers, many different approaches are pursued, custom to specific problems or to particular architectures.

OROCOS (Open Robot Control Software, [4,34]) is one of the most comprehensive systems that follows a similar approach to the one we have described in this paper. OROCOS is composed of: Real-Time Toolkit (Real-Time engine for control execution and communication of components); OROCOS Component Library (library of control components); OROCOS Kinematics and Dynamics Library, and Orocos Bayesian Filtering Library. The Real-Time Toolkit (RTT) is the engine used to execute OROCOS applications. It is organized as depicted in Fig. 4.

Generally, the most visible shortcoming of the cited solutions is that each of them is specialized for a particular domain or for a specific purpose. In other cases, some of the solutions described put too much focus on specific features, overloading them with too much functionality, which, in practice, is not really necessary.

We believe that it is crucial to have one single research and development environment that, while giving freedom to design and implement any architecture (see [26] for similar arguments), provides all the necessary support for system decomposition, communication patterns, scalability, and execution with different sequential or parallel paradigms. Moreover, simplicity and modularity are, in this respect, key issues in the development and the usage of such research and development environments. We believe that the adoption of a component model is a crucial aspect for developing intelligent systems, since building systems out of components/modules allows cutting complex problems into smaller pieces, reducing complexity, increasing reusability, and decreasing dependencies (see [27,12]).

3. Architectural structure and incremental design process

The design of the system is separable into three major parts: The computing architecture, the functional architecture, and the instance of an application. The computing architecture comprises all of the software and computer technology in order to provide a framework and an infrastructure for computing.

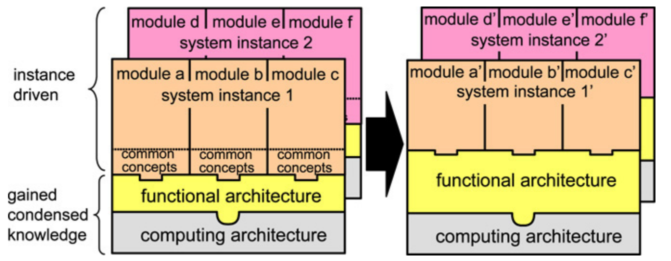

Fig. 5. Subdivision into computing architecture, functional architecture, and instance.

Here, the major design constraints and means are determined by technology, by real-time computing, and, at this stage, by only a few but fundamental principles from biology (see (7) in Section 1). Those constraints lead to a computing architecture, where several different functional architectures can be implemented.

A functional architecture represents the constraints of a hypothesis or model of the network of functional areas in the brain or any other general system concept that makes different modules or components interact. Since the design of the functional architecture is not a 'simple' design but a major research field, it can only be approached in an iterative fashion. Within a functional architecture, several different instances of applications can be implemented (see Fig. 5). The design decision to separate the system into computing and functional architecture was made in order to lead to a stable computing environment, one within which several functional architecture revisions can be evaluated. Any advances on the functional side may call for changes on the computing side, especially since the separation can never be perfect. But the idea is that changes on the computing side are on a slower time scale than on the functional side if the computing side is sufficiently general. The same applies to the relation of the functional architecture and instances (see (3) in Section 1). We use the term 'instance' for a system hypothesis that implements a certain set of functionality (e.g. object recognition system control-loop, interactive object learning system, ...); it represents one of the systems we build and investigate at a certain moment in time. The instance is the most frequently changing part of the whole system. By means of the different instances, some common knowledge and principles can be researched that are condensed in the next version of the functional architecture. This process leads to a continuously growing functional architecture providing richer means for future instances. See Fig. 5 for a visualization of this process.

Under our architectural assumptions, there are several dimensions which should be investigated for researching and creating intelligent systems. These are:

- Granularity or abstraction level of modules;
- Asynchronous vs. synchronous processing;
- Sequential vs. parallel processing;
- Elementary connectivity patterns;
- Communication paradigms;
- Generality of modules, learning, and additivity;
- State vs. stateless modules;
- Time scale of modules; and
- Data formats and types.

The first design of our system was composed of two layers: computing architecture and software architecture. As proposed in [6], we designed our modules on the required hardware for our system. Extending this architecture, we started with some principal considerations of the functional architecture, because this defines the global direction of the overall system. The computing architecture must provide the means for implementing this kind of a functional architecture, and the application will be realized within such a framework as a system instance. Again, both the architectures have to be sufficiently general in order not to limit the application, because here the new knowledge has to be gained that will be condensed into both the functional and the computational architecture.

3.1. *Computing architecture*

The computing architecture is the framework and the means for implementing functional architectures. In order to allow optimal flexibility for the different design dimensions on the functional side, the appropriate means of the computing side have to be provided. The computing architecture that we refer to in this paper is currently based on standard computers and standard operating systems. A certain degree of portability should be supported by a computing architecture in order to leave the possibility of testing systems on platforms that provide support for special hardware or software libraries.

The computing architecture should cover:

- Modularization support;
- System integration support;
- Simplifying the definition of parallel applications;
- Automatic data synchronization among modules/threads/processes;
- Ensure real-time performance (deterministic execution);
- Multi-platform support.

An abstraction layer over standard operating systems and compilers is necessary in order to support such features. We have created a middleware that, running on top of the operating system, handles concepts like modules, parallelism, and synchronization (see (7) in Section 1). Real-time performance and multi-platform support is achieved through a balanced combination of customized implementation (for most of the time critical parts) and external libraries (see (5) in Section 1).

3.2. *Functional architecture*

It is our opinion that *ex post* integration of independently designed modules will never lead to a well integrated system as a whole. Therefore, we propose inverse integration, i.e. a series of constitutive functional architecture hypotheses has to be devised, which guide the research and development of functional modules. There are several partial solutions to specific problems available in the scientific community, but their integration is either rather low or less coherent than necessary. According to our opinion it is crucial to aim for an integrated, incremental, and convergent development effort to reach the critical mass necessary for tackling fundamental problems of computational intelligence (see Section 1). The functional architecture provides the concepts that allow for the process stated above. Some of these elements are:

- General means for supporting the kinds of targeted processing needed, like image, sound, and matrix manipulation libraries;
- Computing and data module definitions;
- High level communication and timing protocols;
- On-line and off-line data inspection;
- Design phase support through both scripting and graphical environment.

A general assumption on the functional side is that the computational resources are sufficiently large in order not to limit the research and the development of concepts. It is the responsibility of the computing architecture to provide such kind of resources. Nevertheless, the research for functional algorithms should be guided by a set of principles that ensure that the fundamental design philosophy is met and the functionalities contribute to the goals. This requires a proper scalability and computational complexity of the algorithm. Here, with scalability we mean the possibility for an algorithm to produce meaningful results in a consistent time, independently of the size or frequency of the input data stream. Meanwhile, with computational complexity we identify the CPU usage and the overall dimension of an algorithm. Modularization, parallelization, and distribution are approaches for handling complex algorithms (reduction of complexity per module is a significant target). Algorithms, in a real-time context, should always take such issues into account. We have implemented such services by creating libraries for image, sound, and matrix manipulation, defining component models for data and computing modules, creating a graphical design tool, as well as an integrated monitoring system. The issues raised here will be discussed in more detail in Section 4.

3.3. *Instance*

If the described constraints in Sections 3.1 and 3.2 are met, a wide range of applications can be created with the proposed infrastructure. However, there is still a full spectrum of conventions and decisions that have to be considered/established in order to implement any instance application. But in this phase, the computing architecture and the functional architecture already provide a broad support. At this stage, new solutions, new approaches, and new methodologies should be researched and implemented in order to achieve the desired functionality.

In the instance layer, the following issues should be taken into account:

- Definition of a process for system design, creation, test, and development (more in the biological sense);
- Interface standardization;
- Architecture definitions at different levels;
- Module repository organization.

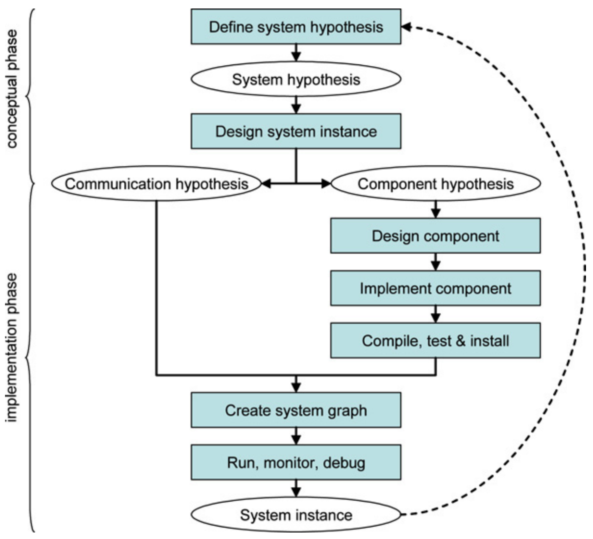

Fig. 6. Systems resarch process.

We have implemented such services by the definition of processes organizing our research and development phases. We have also created standard interfaces for our modules and a common data set in order to standardize communication channels (see Section 5). This standardization process has been achieved through the identification of a common set of port names (for our BBCM components), with a common data type and behavior. When a component uses one of them, we can relay the name, type, and behavior of the port. For the data set (the BBDM components), we have defined a minimal set of common data types used in our system for communicating data from/to a module (BBCM). We decided to follow a more generic data type definition: unspecific in the semantics – e.g. Image –, but instead, specific on the representation — e.g. Vector2D.

4. Systems research process

In Software Engineering, it is well known that the creation of large-scale systems must be supported by a well defined process. Such a process defines the various phases a project is composed of, the relation between such phases, and the interactions and the tools required to accomplish the necessary tasks. We decided to employ such a process for our systems research in order to gain advantage from it and structure our work. The overall process is shown in Fig. 6.

This process governs the life-cycle of our systems research. An iteration of the full process runs on the time scale of years. Each step of this process is executed by several actors and is actually governed by sub-processes that determine the more fine grained actions.

We distribute a set of checkpoints in the process to ensure a certain level of quality at each step. The process starts with a conceptual phase where we maturate and formalize the ideas we want to realize in our systems. In this phase, every researcher may use a different approach to reach his/her results. But in the transition phase (arrows in the graph), we coordinate and collect hypotheses into one common form. This is very important in order to cross-check the weak points or the unclear ones, or simply to confirm what seems to be accepted by the researchers. Other steps in the implementation phase are more concrete and require the creation of software modules, the testing of the functionality, or the analysis of properties that modules should have. For such steps we decided to develop a set of tools that support the accomplishment of the related tasks.

We will now revisit the process of Fig. 6 and analyze each step in more detail:

4.1. *Define system hypothesis*

This step belongs to the conceptual phase where the research work is concentrated on the definition of a system hypothesis. In this phase we elaborate the architecture, the functionality, and the methods to be researched. The goal here is to reach a coherent system hypothesis that contains all the functionalities and the principles we want to experiment on in an integrated concept.

4.2. *Design system instance*

This last step of the conceptual phase aims at a concrete design of the system. Here ideas and principles are mapped to concrete functions (component hypotheses) and relations (communication hypotheses). This step allows us to understand what type of components are required in order to develop the target system. This phase and the previous one belongs to the *instance* layer of our system subdivision (refer to Section 3.3).

Decomposing a system into components requires a certain balance between generality and specialization. The space of components can go from a simple operation like addition or multiplication, to more complex ones like filter banks, integrators and components that perform object recognition or speech synthesis. The main driving forces for choosing granularity are openness for extensions, reuse, and performance. In our experience, we have had the need to develop components which implement simple operations, and components that perform more complex functionality. The systems we are targeting are real-world systems. Decomposing a system in too fine-grained components may overload the communication channels. This has to be balanced with the choice of the employed communication paradigms. Communication between several components that use the memory channel of a single machine is much more convenient (in certain circumstances) than a communication channel over network connections among computers. Here there is again a wide spectrum of possibilities. Network communication between computers can be implemented in many different ways, e.g. TCP, UDP, and Reflective Memory Cards. The goal of this step is to identify the components we need and the communication patterns we want to employ for each part of the graph of the system (see (6) in Section 1). In this step, we also identify which components can be reused and assign them to a pool or dedicated computers. This partition for some part of the system may be constrained by the availability of sensors and effectors on certain hosts.

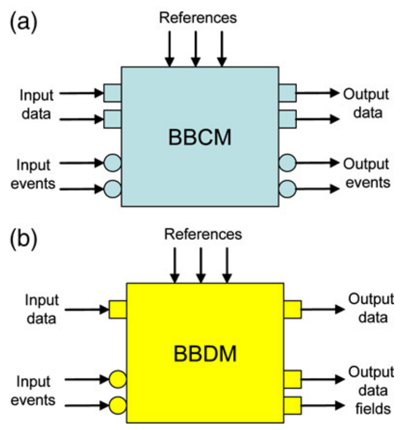

Fig. 7. (a) Brain Bytes Component Model interface; (b) Brain Bytes Data Model interface.

4.3. *Design component*

The implementation phase starts with the design of components. In our philosophy, we clearly separate processing from communication, to allow experimenting with different systems architectures, connectivity graphs, and communication means without touching the internals of the processing algorithms. Therefore, we decided to encapsulate the processing (algorithms) into components. We have defined the BBCM (Brain Bytes Component Model) specification, a very simple but powerful component model that implements the interface shown in Fig. 7(a).

All communication with a component is performed via this interface, which in the case of input and output are simple memory blocks handed over to the component from the outside. Events are data-less function calls. The BBCM has been implemented in the languages C, C++ and Matlab. The data being communicated usually follow a data component specification for data types. It is called the BBDM (Brain Bytes Data Model) and allows wrapping any data type into standard components with a well defined interface (see Fig. 7(b)).

The two component models (BBCM and BBDM) have been designed to work together (but they do not depend on each other). Furthermore, a C BBCM and a C++ BBCM component can be used and can be connected together in the same source code file. The proposed standardization level is, on the one hand, sufficiently strict in order to be a suitable basis for additional tools that bear much on the implementation burden from the researchers, and on the other hand sufficiently flexible in order to allow for experimenting with different processing and communication paradigms.

Through the adoption of software components, our systems are actually flexible and reconfigurable. We can easily reuse old components and test new ones with a very minimal effort. But now we have to deal with the following issues:

- *Select the appropriate size for components (complexity)*: A system can be composed of components that implement simple functions like: addition, subtraction, loop, conditional, etc. On the other hand, components may implement a full object recognition algorithm or a full set of motion behaviors. Surely in the first system each module can be easily reused, but the connectivity graphs of such a system would not clearly reflect the control flow of the system but rather an intricate net of modules interconnected with each other. By contrast, the second system would more clearly show the functionalities of the system, but it cannot easily be extended through the insertion of new modules since it would not be possible to affect the computation done within a module.

- *Define an appropriate interface*: The questions here are: What should be exposed? With what granularity? Should specific or general data types be exposed? It is surely beneficial to define interfaces that expose generic data types (integers, vectors, ...), since this approach leads to an easily connectivity between components.
- *Handle synchronization*: When should a component generate an output? When are all inputs available? Should output be generated when the first input arrives? If a component has more than one output, should it produce a new value for all each time? Surely part of the need for synchronization comes from the matured culture of sequential computing/programming. Standard computing environments assume that a program is composed of a set of instructions that are executed one after the other (sequentially). In these conditions, synchronization is implicitly given by the fact that an operation is executed only after the execution of the previous one. Looking at biological systems, more parallelism and more asynchronous processing is perceived. Therefore, it is possible that more than one operation is executed at the same time, raising the need to create operations with less interdependency, and thence more capabilities of keeping coherence even in unexpected sequences of execution.
- *Maintain a sufficient level of documentation*: Once a certain number of components is reached in a group of researchers, the need for clear and accessible documentation becomes an important matter. Reusability can be technologically achieved, but retrievability is a different issue.

4.4. Implement component

This step consists of creating and coding the different components. It can be performed by the researcher or outsourced to external companies. Sufficient care should be taken in this step and the previous one. Since both of them belong to the *functional architecture* (see Section 3.2), decisions about the generality or the specificity of a component may influence the next iteration of the full process. This step is supported by different tools and conventions:

Coding and naming conventions: We decided to employ a set of conventions for our software. Through the coding conventions, we improve readability and ensure a certain level of quality. Here, we tried to keep the set of conventions as small as possible in order to minimize the effort to learn and use them. We support their usage through web tools.

Image and sound libraries: Among others, main fields of research are image, sound, and speech understanding as well as robot manipulation. For these reasons we have created VLW (Visual Library Wrapper), and SLW (Sound Library Wrapper); libraries that support image and sound manipulation (e.g. arithmetic operations, filtering, transformation, and conversions). These libraries are wrappers (providing a standard interface to our software) around existing libraries. The wrapped libraries are: the IPP library [1], the Media Lib library [3] and a plain C version used on platforms that do not support IPP or Media Lib. These libraries are heavily used for building the algorithms implemented in our BBCM components.

Template generator: The creation of components is a fundamental step in the process, and therefore supported by automatic code generators that create the initial skeleton for a BBCM or BBDM component from a simple textual description of its interface. We created SMDL (Simple Macro Descriptive Language), a language for transforming any document into a template. The usage of automatic code generators has many benefits, like:

- The generated code always conforms to our standards (conventions and libraries);
- The generated code is less prone to errors;
- Large amounts of source code, makefiles, settings and directory structures can be generated in a fraction of time.

4.5. Compile, test, and install

This traditional step is supported by several standard and custom tools that remove some of the tasks that are usually associated with this phase for a multiplatform environment:

Versioning system: We are currently using SVN (Subversion [2]) for source code versioning. This tool (open-source, freely available on the internet) handles directory and source code file versioning in a local or networked configuration.

Multiplatform makefile system: We have based our build system on the GNU make [39]. Our build system is composed of two sets of files: 'makefile' (read-only for users), which contains all rules we need for building, installing, and handling the versioning system for all types of modules and 'makeVar', which contains user settings (including paths, libraries, and compiler options). Our build system is multi-platform and supports projects for C, C++, PHP, Matlab CMEX and Java.

Multiplatform deployment structure: In order to store and share our components, libraries, and applications, we have created a multi-platform installation directory structure. This directory structure (that looks the same for all platforms we use) includes files, shared libraries, binaries, documentation and configuration files. In order to avoid the common problem of parallel development on a single source tree, we opt for a solution where we share compiled code (binaries, libraries). Every installed module contains the following configuration files: packageVar (for compiler settings and paths) and a file TcshSrc (for execution settings). To include a module just the respective packageVar and TcshSrc have to be used. In this way, the required compiler and execution settings are inherited. Moreover, our build system gives support for this multi-platform installation tree when compiling and installing modules.

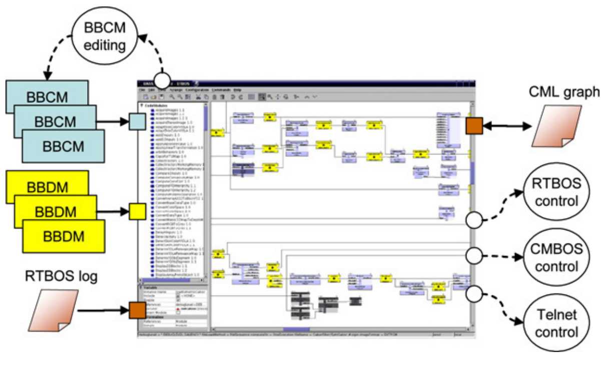

Fig. 8. Design Tool for Brain Operating System (DTBOS).

4.6. *Creation of system graph*

At this step of the implementation phase, the communication hypothesis and the implemented components are used to build the system graph. Clearly, we are in the *instance* layer of our system subdivision (see Section 3.3). To support and simplify the design and the creation of our applications, we have developed DTBOS (Design Tool for Brain Operating System). The tools described in this section are part of our *functional architecture* (see Section 3.2).

DTBOS is an "RAD-like" development environment that allows importing of RTBOS modules (code (CM) and data (DM) modules) into a template-tree with which users can draw applications in a graphical way (see (4) in Section 1). CMs/DMs are drawn as boxes with input/output ports; connections can be created by just dragging links from one port to another. DTBOS can read and save files in the same script format (CML, C Macro Language) accepted by RTBOS. Moreover, with DTBOS we can debug our applications through a playback functionality that reads an RTBOS log session and shows, through animation, the executed modules, step by step. A screenshot of DTBOS is shown in Fig. 8.

With DTBOS, we can easily handle graphs composed of several hundreds of components; the same applies to controlling the data and functional flows as well as modifying portions of graphs without touching the entire system. Such flexibility yields a high speed in creating systems, so that now they are quite rapidly increasing in size. This new situation raised new issues that we have had to deal with:

- *Explicit wiring vs. generating graphs through scripts*: Designing and creating a system composed of a well defined set of modules is a task that can be handled without too much effort. But in a system where more developmental aspects should be tackled, the purely static design becomes insufficient. In this case, it is not possible to explicitly add some more modules and draw a fixed connectivity for them. Here the growth process may be supported by a developmental program that runs in the system itself. In this case, it becomes necessary to have a script-based (or run-time based) generation of portions of the graphs.
- *Manual vs. automatic thread/process assignment*: With DTBOS, we have created the possibility of assigning threads and processes to each module of a graph. This is necessary for controlling the CPU usage depending on the type of computation a system should perform and for the synchronization of module execution. Parts of a graph can be executed within one thread in order to avoid too many threads that would simply wait for each other and compute their functionality one after the other. The partitioning into processes is necessary (see (4) in Section 1) in order to be able to distribute the execution of a big system over several machines (we partition our system in order to execute only one process per machine).
- *How to visualize synchronization conditions*: Implicitly or explicitly, we need to implement a set of synchronization points in our graphs. Synchronization is used to ensure that certain properties of the data flow towards some components are respected. Consider a component of a vision system that receives the left and the right camera images and computes a disparity map. The component would require that the images it receives in the left-image and right-image input have been taken from the cameras at the same time. Such synchronization conditions may have local and global dependencies. In case of local dependencies, the component may implement such conditions directly inside its algorithm. If there is a global dependency, usually we have specialized components that only perform the synchronization, and then propagate the synchronized data to the next components. Due to the specialization of the synchronization conditions, it is currently not possible to display them in the graph.
- *Layout of big graphs*: With DTBOS we use a two dimensional canvas to draw our graphs. All components are represented as rectangular boxes and placed in the canvas. To support the creation of such big graphs, it is convenient to have automatic layout functionalities. Such functionalities should respect a monotonic organization of the graph in order to keep the relative position of the components throughout the growth process of the graph.
- *Computing startup parameters*: Each instance of the BBCM/BBDM components can be customized through a set of reference parameters. Currently, in our graphs, such references are represented through constants that can be set at the graph level. This is not sufficient, since there are several cases where two or more components share a set of references with functional dependencies (a reference value of a component can be computed from one or more references of other components). To handle such inheritance-like situations, the possibility of computing references (through the necessary equations) is required.

4.7. *Executing, monitoring, and debugging*

The last step of the implementation phase consists of the execution of the system, the debugging, and the testing in simulated and real-world conditions. The following tools support this step:

Distributed middleware for real-time applications: We have created RTBOS (Real-Time Brain Operating System), a distributed middleware for multi-platform real-time modular applications. The requirements from the beginning were to

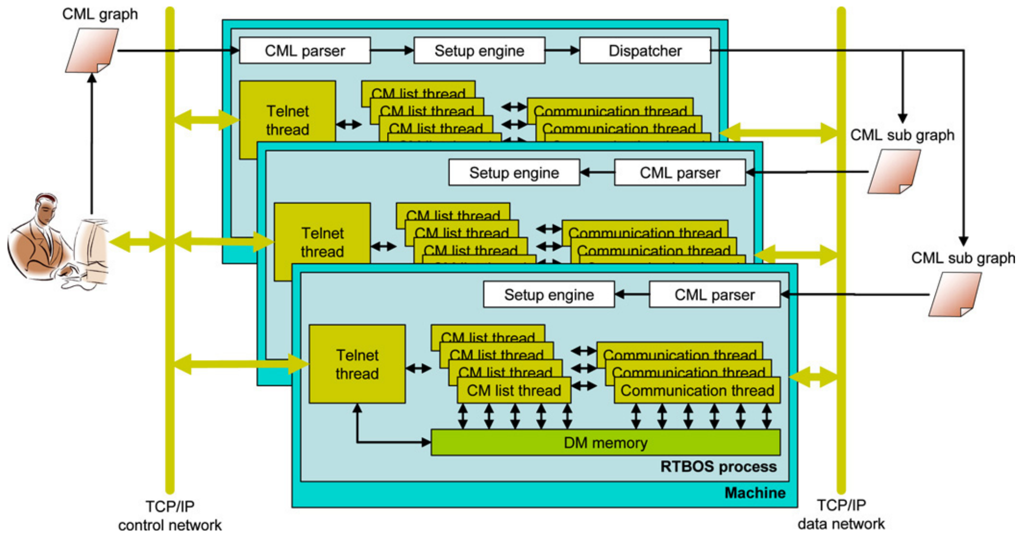

Fig. 9. Run Time Brain Operating System (RTBOS).

provide an integration environment that is suitable for vision, sound, internal prediction and decisions, as well as for behavior generation and the control of actuators. The sensory streams demand for a high data bandwidth, the internal processes for a high connectivity, and the control aspects for a low latency and a strict determinism in processing and communication. The internal mechanisms account for those requirements by providing low overhead communication and data sharing mechanisms as well as deterministic thread control (see (6) in Section 1). An RTBOS application is defined by a set of: Computing Modules (CM), Data Modules (DM), a Connectivity Graph and Execution Patterns. A CM is an RTBOS module that encapsulates a BBCM component; a DM is an RTBOS module that encapsulates a BBDM component; a Connectivity Graph defines the connections between CM and DM, while an Execution Pattern determines the ways CMs are executed. We have identified several patterns, which can be defined for arbitrary sub-partitions of the system instance.

The major Execution Patterns are:

- *Parallel*, which executes each CM in a separate thread. This execution pattern is necessary for executing parts of the system in parallel. It is the basic execution pattern, since our underlying hypothesis is an asynchronous system that can be synchronized if necessary (see (1) in Section 1).
- *Sequential*, which executes all CMs in a sequence, all in one single thread. The choice of this execution pattern allows for a fine grained control of the computing resource allocation.

When RTBOS starts an application, it loads the CM/DM modules defined in the CML script file (generated by DTBOS); connects them following the Connectivity Graphs, initializes all modules and then executes the full application (Setup engine in Fig. 9). The parallelism of an RTBOS application may employ threads (within a machine) or processes (across machines). RTBOS, through the concept of Execution Patterns drastically simplifies the design and the creation of parallel applications. Moreover, RTBOS automatically handles data allocation, communication, and synchronization between threads and processes even across computer boundaries (see (7) in Section 1). RTBOS is part of the *computing architecture* since it deals with more technical aspects strongly related to the hardware (refer to Section 3.1).

The script file (CML) is plain text and can be quickly edited with any text editor. It is a simple language providing a set of commands for managing applications as described above. It can be easily parsed and generated by other tools like the design environment DTBOS (actually it can also be compiled statically with RTBOS and all modules, since it conforms to the C language). This feature leads to a higher acceptance for both researchers preferring to work with script files and researchers preferring GUIs.

The experience with RTBOS has shown us that we can easily execute several systems, distributed on many machines, with a minimal effort and with good performance. The adoption of a script file for the definition of an application is surely an efficient solution. On the other hand, we have now to deal with:

- *Keeping real-time constraints in a constantly growing system*: Currently some of our applications are the result of the contribution of several researchers. Each one of them is able to research and develop a part of the system alone; and only in a later stage integrate his modules into the common system. Even if we all try to develop our algorithms with low time-execution requirements, in the integration phase we still discover time-execution problems when assembling our systems. To keep certain real-time constraints (even if in most of the cases we need soft-real-time) additional tools and methods are required. In an integrated system composed of several hundreds of modules, it is still difficult to identify the reasons for timing problems. It is also difficult to find the possible approaches to keep time-execution to the required ranges with our current methodologies.
- *Debugging and monitoring system behavior*: Debugging and monitoring our applications is still a rather complex task for all researchers. It is surely possible to debug one or a few modules alone, but when we are in the integration process, it is still not possible for us to keep track of several modules,

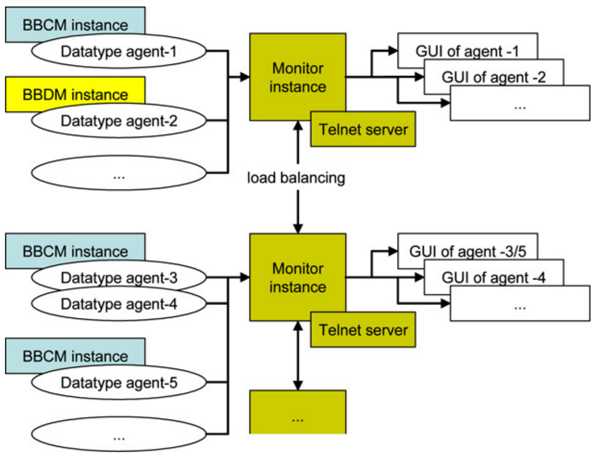

Fig. 10. Control-Monitor for Brain Operating System (CMBOS).

threads and machines in a way that enables us to determine the behavior of the whole system in an easy way. Here again, new tools and methods are required in order to reduce complexity and let users understand, on-line or off-line, the actual system flow (see (4) in Section 1).

- *Startup of a system*: Our applications are still requiring a strictly synchronous startup. This implies that the first iteration of all modules is started at the same time. This is due to the fact that some modules strongly depend on the sequentiality of events when executed with respect to data initialization (setting of default values to data modules). We are aware that such restrictions cannot be present in the long term, but removing this restriction means actually redesigning the data flow of some components.

Monitoring control system: We have also created the monitoring system CMBOS (Control Monitor system for Brain Operating System). CMBOS is a daemon process that runs in a monitoring machine (see Fig. 10). This daemon accepts requests for data monitoring from one side while it displays the requested data in an appropriate user interface automatically. CMBOS contains a set of already defined user interfaces that show 2D graphs, counters, images and other types of visualization tools. Users can implement new visualization tools and new data types to expand the functionality of CMBOS (see (4) in Section 1). CMBOS is the youngest part of our software environment and still needs further development. The main problems we are currently facing are:

- *Minimize interferences with the running system*: Clearly, like any monitoring system, we have to ensure that the usage of a monitor process to inspect the internal data of a running system induces the least possible interference. The system should behave nearly the same with or without inspection of the data. Here there are several approaches available, like assigning a dedicated CPU for monitoring-related computations, and adopting reflective memory or delegating monitoring to a low priority process/thread. The solution we have adopted employs a server (the Monitor Server) that listens for any monitoring data and sends this data to another computer that physically shows the data.
- *Define appropriate visualization and/or combine visualization*: The process of monitoring requires that the internal data of a system are visualized on a computer screen in a form that helps the researchers to understand and relate the system's behavior with what has been visualized. It is not always possible to display the same data in the same format (LCD display, 2D graph, histogram). In some cases the visualization format has to be defined specifically for certain modules in order to really capture how a module is functioning. In other cases, it is also required to combine several data streams and display all of them in one single view, so that time and spatial relations are kept.

4.8. Closing the loop

Based on the described overall process, we have now introduced all tools for supporting the workflow. In close collaboration with researchers, we are continuously working on the refinement and the extension of the covered concepts. We have already started the integration of all of the mentioned tools into one single environment. Our target is to make DTBOS the central interface, and ultimately the only tool that users would need to know. From DTBOS it will be possible to create new BBCM/BBDM components, edit them, compile them, import them automatically into the toolbar, use them to draw applications, execute RTBOS sessions and open CMBOS monitoring sessions directly through the DTBOS interface (see Fig. 8).

All our tools have been designed to work on several platforms. We currently support Linux, Sun Solaris, partially Windows (native and CygWin) and VxWorks. We are interested in expanding the list of supported platforms in order to improve stability, portability, and performance.

5. Application and experiments

The tools described in this paper have been growing together with our scientific research systems over the last couple of years. The valuable feedback from researchers concerned with their own functional methods and algorithms has found its way directly into the conceptual improvements of our tools. The tools, especially RTBOS, have been tested on benchmark problems in order to examine their theoretical performance. As a result of these experiments, we could estimate the overhead introduced by RTBOS compared to a hand coded system. Assuming extremely simple processing elements and spending very little time on actual computation, the system is mainly communicating, putting a high load on the middleware. This assumption represents a theoretical worst case setting for middleware systems. The value we measured was approximately one percent of the overall processing time.

We will now report on some experiments using the tools in real-time, real world settings. There are three major areas of applications and experiments to report on; others are subject to current research with publications under preparation.

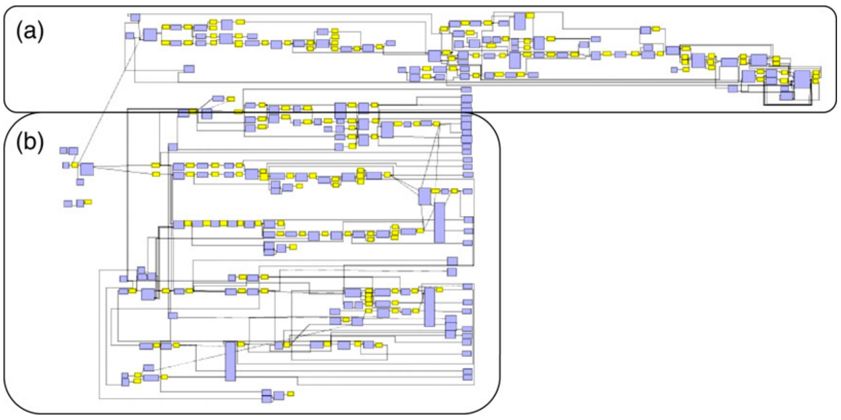

Fig. 11. DTBOS graph of our active vision system. (a) Online object recognition and learning sub-system; (b) Gaze control sub-system.

5.1. *Interactive vision system*

The first area is concerned with an interactive vision system composed of: image acquisition, saliency computation (from several cues: color, intensity, saturation, motion, disparity), visuo-motor mapping and learning, gaze selection, object recognition and learning and camera head control.

This system is able to accomplish the following tasks: visuo-motor mapping learning, interactive attention modulation, object fixation and recognition, scene exploration, and simple feature-based object search. Some of the corresponding work has been published in [30,18,21,17,25,38].

In total, 13 researchers have been contributing to and working with this system in an incremental fashion.

The system has a brain-like systems architecture, corresponding to the dorsal and ventral processing streams in the human visual cortex including the superior colliculus' controlling gaze. The snapshot of the connectivity graph with the respective subsystems is shown in Fig. 11. It has now been growing over the last four years, reaching 202 BBCM instances as processing modules and 157 BBDM instances as data modules. The average number of instances per coded BBCM is around two, while the average number of instances per coded BBDM is five.

Those numbers are a mild estimate for the reuse of components within one systems instance. There are several variations of the system from at least six computational threads up to a fully parallel system with the number of threads in the order of the number of processing modules. The execution time for the gaze selection computation cycle is in the range of 200 ms, the time for the on-line learning between 80 and 320 ms, and the time for the actuator control is about 500 ms. The current version runs distributed across two Pentium IV 3 GHz, one with 2 CPUs and the other with 4 CPUs, both running Linux connected via dedicated GigaBit Ethernet.

For this system, clearly the connectivity/communication and the number of processing modules to manage them are the main challenges for the software environment, less the latency constraints for the processing.

5.2. *Binaural sound localization system*

More emphasis on the latency constraints is put in the second area, which is concerned with a real-time binaural sound localization system. The system performs the estimation of the pan angle of incoming sound events and the control of the gaze direction of a robot head based on this estimation in real-time. It is composed of the following stages: sound acquisition; 3 streams for sound position computation (IIT, IID and IED); integration stage and head control. The corresponding work has been published in [20] and [29].

The main two subsystems are the hardware interface and the sound localization computation. The hardware interface performs the stereo acquisition of the sound streams and the control of the ASIMO head, while the sound localization computation performs a parallel estimation of the sound source pan angle. The stream cycle is 50 ms for both subsystems and the number of threads is ten. The total number of BBCM instances is 222, and the total number of BBDM instances is 275. Again, the two subsystems are distributed across several Xeon 3.6 GHz computers, four with 2 CPUs and one with a single CPU, all running under Linux connected via dedicated GigaBit Ethernet. The connectivity graph of the system is sketched in Fig. 12. In total, seven researchers are contributing to this system.

While the previously presented vision system is mainly asynchronous, the auditory system is more synchronous. The reason is that the visual setting's signal type is more continuous, allowing for a more loose kind of synchronization, while the auditory setting is composed of rare events that have to be tightly synchronized in order to meaningfully fuse the different analysis streams into one stable pan angle estimation. Both kinds of processing paradigms could be equally well implemented using the proposed tools.

5.3. *Visually guided whole body interaction system*

The third area is concerned with a visually guided whole body interaction system for our robot ASIMO. The system allows ASIMO to gaze, looking for close objects, selecting one and trying to approach the selected target by whole body motions. Parts of this work are being published in [14,15,9]. This system is composed of three main sub-systems: ASIMO; a vision sub-system; and a behavior control sub-system. The vision sub-system receives images form ASIMO's stereo

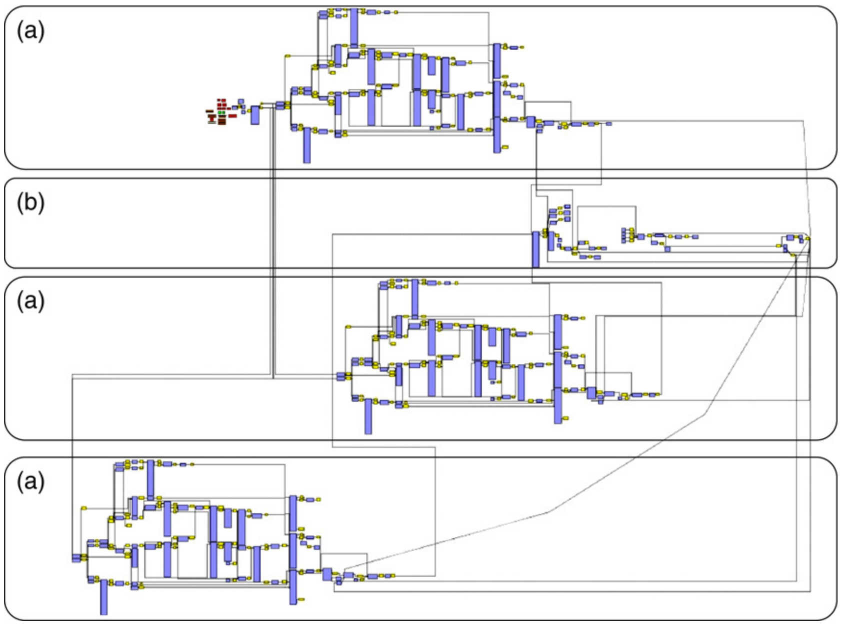

Fig. 12. DTBOS graph of our real-time binaural sound localization system. (a) sound localization computation parallelized on three processors; (b) Hardware interface sub-system.

camera and creates a set of possible targets. The control subsystem selects one target and then controls the behaviors that ASIMO has to execute. Behaviors are currently: searching, tracking, and reaching. The vision subsystem has a stream cycle of 150 ms, the control system has a stream cycle of 5 ms. The total number of instantiated BBCMs is 29, the total number of instantiated BBDMs is 33. The connectivity graph is sketched in Fig. 13. The system is distributed across three different machines, one equipped with an ARM CPU on-board on ASIMO, and two Pentium IV 3 GHz with two CPUs, the first running under VxWorks and the other two with Linux connected with a dedicated GigaBit Ethernet. The mixture of handling sensor data coming from the environment and the actual systems control puts the hardest constraints on the tools, especially on RTBOS.

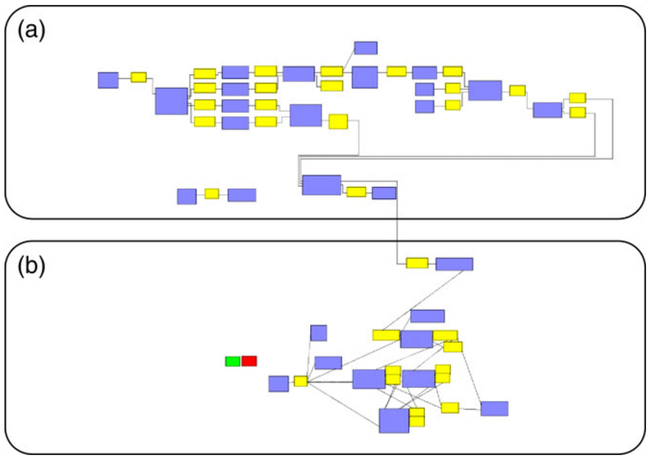

Fig. 13. DTBOS graph of our visually guided whole body interaction system. (a) vision sub-system; (b) behavior control sub-system.

6. Summary

In this paper, we described the principles and the methodologies that we have researched for the creation of an integrated research and development environment, the base of our brain-like intelligent systems. We have described the process that governs the life-cycle of our systems research; through this process we have analyzed the main tools we use for integrating our software systems. The component models (BBCM and BBDM), the design tool (DTBOS), the execution engine (RTBOS) and the monitoring tool (CMBOS) constitute the main parts that an integrated software environment should have in order to ensure a certain level of quality and efficiency.

In addition to the experience within the three different fields cited in Section 5, there are some general observations we would like to report on. We could experience that the limits of our tools are reached if we are close to the limits of the underlying operating systems and hardware. Those limits are mainly due to general purpose schedulers and limited network bandwidth. For us, this is a mild indicator that the overall overhead introduced by our tools is small.

Major parts of our experiments have been contributed by non-systems programmers. With the aid of the offered tools, they were able to create and manage real-time multi-threaded parts of the systems. The overall reuse of components across the different fields depends on the level of granularity of the components. Generally speaking, the reuse is higher for finer-grained components. For example, there is a large overlap of the simple components between the visual components of the first and the third system, but exact numbers have to be determined.

The chosen decomposition of the systems as well as the means for working visually and monitoring various systems' properties allows for real research on the systems level. As pointed out in the paragraph on the auditory systems, different kinds of processing and communication paradigms can be created and managed with the same tools. For example, in the area of the active vision system, we could investigate different synchronization mechanisms without touching any of the processing modules.

In the last four years, we have gained experience in using our integrated software environment, and we have clearly demonstrated that we are now able to setup large systems as the result of the work of several researchers in our lab.

Acknowledgments

The work presented in this paper has been created by a small group, but has been supported and inspired by the ideas and the work of several researchers in our lab. We want to thank Frank Joublin for his continuous requests for necessary features and for the time he invested in meeting and reviewing, especially when this project started. Thanks to Mark Dunn for his contribution on the basic software infrastructure and to Julian Eggert for inspiring several solutions and for the creation and maintenance of the libraries that constitute the core of several algorithms.

References

[1] Intel IPP Library. http://www3.intel.com/cd/software/products/asmo-na/eng/perflib.

[2] Subversion (SVN). http://svnbook.red-bean.com/.

[3] Sun Media Lib. http://www.sun.com/processors/vis/mlib.html.

[4] The OROCOS project. http://www.orocos.org.

[5] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, Brent Smolinski, Toward a common component architecture for high-performance scientific computing, in: Proceedings of the 1999 Conference on High Performance Distributed Computing, Redondo Beach, CA, 1999.

[6] Tamim Asfour, Karsten Berns, Rüdiger Dillmann, The humanoid robot ARMAR: Design and control, in: Proceedings International Conference on Humanoid Robots, Humanoids 2000, Boston, MIT, USA, 2000.

[7] Len Bass, Dr. Paul Clements, Dr. Rick Kazman, Software Architecture in Practice, Addison-Wesley, Boston, MA, 1998.

[8] Berthold Bäuml, Gerd Hirzinger, Agile robot development (aRD): A pragmatic approach to robotic software, in: Proceedings International Conference on Intelligent Robots and Systems, IROS, Beijing, China, 2006.

[9] Bram Bolder, Mark Dunn, Michael Gienger, Herbert Janßen, Hisashi Sugiura, Christian Goerick, Visually guided whole body interaction, in: IEEE Int. Conf. on Robotics and Automation, 2007.

[10] Brian Gerkey, Richard T. Vaughan, Andrew Howard, The player/stage project: Tools for multi-robot and distributed sensor systems, in: 11th International Conference on Advanced Robotics, Coimbra, Portugal, ICAR'03, 2003.

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerkand, Michael Stal, Pattern-Oriented Software Architecture: A System of Patterns — Volume 1–2, John Wiley & Sons, West Sussex, England, 1996.

[12] Peter Carruthers, Moderately massive modularity, in: A. O'Hear (Ed.), Mind and Persons, Cambridge University Press, 2003.

[13] William T. Councill, George T. Heineman, Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley, USA, 2001.

[14] Michael Gienger, Herbert Janßen, Christian Goerick, Task-oriented whole body motion for humanoid robots, in: Proceedings of the IEEE/RSJ International Conference on Humanoid Robots, Humanoids, 2005, Tsukuba, Japan, 2005.

[15] Michael Gienger, Herbert Janßen, Christian Goerick, Exploiting task intervals for whole body robot control, in: Proceedings of the International Conference on Intelligent Robots & Systems, IROS, IEEE, 2006.

[16] Giorgio Metta, Paul Fitzpatrick, Lorenzo Natale, YARP: Yet another robot platform, International Journal on Advanced Robotics Systems (2005).

[17] Christian Goerick, Inna Mikhailova, Heiko Wersing, Stephan Kirstein, Biologically motivated visual behaviors for humanoids: Learning to interact and learning in interaction, in: Proceedings of the IEEE/RSJ International Conference on Humanoid Robots, Humanoids, 2006, Genoa, Italy, 2006.

[18] Christian Goerick, Heiko Wersing, Inna Mikhailova, Mark Dunn, Peripersonal space and object recognition for humanoids, in: Proceedings of the IEEE/RSJ International Conference on Humanoid Robots, Humanoids, 2005, Tsukuba, Japan, 2005.

[19] Gordon Cheng, Sang-Ho Hyon, Jun Morimoto, Ales Ude, Stephen C. Jacobsen, Cb: A humanoid research platform for exploring neuroscience, in: Proceedings of the International Conference on Humanoid Robots, Humanoids, 2006.

[20] Martin Heckmann, Tobias Rodemann, Frank Joublin, Christian Goerick, Björn Schölling, Auditory inspired binaural robust sound source localization in echoic and noisy environments, in: Proceedings of the International Conference on Intelligent Robots & Systems, IROS, IEEE, 2006.

[21] S. Kirstein, H. Wersing, E. Körner, Rapid online learning of objects in a biologically motivated recognition architecture, in: 27th Pattern Recognition Symposium DAGM, Springer, 2005, pp. 301–308.

[22] Jeff Kramer, Distributed software engineering — invited state-of-the-art report. http://citeseer.ist.psu.edu/167613.html.

[23] Phillip A. Laplante, Real-Time Systems Design and Analysis, Wiley-IEEE, Piscataway, NJ, 2004.

[24] Frank Lüders, Adopting a software component model in real-time systems development, in: Proceedings of the 28th Annual NASA/IEEE Software Engineering Workshop, IEEE Computer Society Press, February 2004.

[25] Inna Mikhailova, Werner von Seelen, Christian Goerick, Usage of general developmental principles for adaptation of reactive behavior, in: Proceedings of the 6th International Workshop on Epigenetic Robotics, Paris, France, 2006.

[26] Issa A.D. Nesnas, CLARAty: Towards standardized abstractions for robotic systems, in: Workshop Principle and Practice of Software Development in Robotics, ICRA2005, Barcellona, Spain, 2005.

[27] Richard N. Langlois, Modularity in technology, organization, and society, in: Department of Economics, University of Connecticut, 1999.

[28] Olivier Stasse, Yasuo Kuniyoshi, PredN: Achieving efficiency and code re-usability in a programming system for complex robotic applications, in: International Conference on Robotics and Automation, ICRA, San Francisco, CA, USA, 2000.

[29] Tobias Rodemann, Martin Heckmann, Björn Schölling, Frank Joublin, Christian Goerick, Real-time sound localization with a binaural head-system using a biologically-inspired cue-triple mapping, in: Proceedings of the International Conference on Intelligent Robots & Systems, IROS, IEEE, 2006.

[30] Tobias Rodemann, Frank Joublin, Edgar Körner, Saccade adaptation on a 2 dof camera head, in: Horst-Michael Groß, Klaus Debes, Hans-Joachim Böhme (Eds.), Third Workshop on Self-Organization of AdaptiVE Behavior, SOAVE 2004, Ilmenau, VDI-Verlag, Düsseldorf, 2004, pp. 94–103. Fortschrittsberichte des VDI.

[31] RTI, ControlShell User's Manual, Version 7.0, Real-Time Innovation Inc., California, USA, 2001.

[32] The PDP Research Group, David Rumelhart, James L. McClelland, Parallel Distributed Processing, MIT Press, Cambridge, Massachusetts, USA, 1986.

[33] Matthias Scheutz, Virgil Andronache, Architectural mechanisms for dynamic changes of behavior selections strategies in behavior-based systems, 2004.

[34] Christian Schlegel, A component approach for robotics software: Communication patterns in the OROCOS contex, in: Workshop Principle and Practice of Software Development in Robotics, ICRA2005, Barcellona, Spain, 2004.

[35] David B. Stewart, G. Arora, Dynamically reconfigurable embedded software—does it make sense? in: Second IEEE International Conference on Engineering of Complex Computer Systems, ICECCS'96, Montreal, 1996.

[36] Kazuo Tanie, Standardization of robotics components and future robotics business, in: AIST, Japan, 2004.

[37] Shengquan Wang, Sangig Rho, Riccardo Bettati, Wei Zhao, Toward real-time component-based systems, in: Proceedings of IEEE International Real-time Systems Symposium (RTSS) Work-In-Progress Session, Lisbon, Portugal, 2004.

[38] H. Wersing, S. Kirstein, M. Götting, H. Brandl, M. Dunn, I. Mikhailova, C. Goerick, J.J. Steil, H. Ritter, E. Körner, A biologically motivated system for unconstrained online learning of visual objects, in: Proc. Int. Conf. Art. Neur. Netw. ICANN, 2006.

[39] GNU Make. http://www.gnu.org/software/make/manual/make.html.

Antonello Ceravola studied Computer Science at the University of Pisa, Italy. He worked in the field of IT software for five years dealing with multimedia systems, large scale software infrastructure for telecomunication systems, multi-tier applications and workflow engine for process management systems. From 2001 he joined Honda Research Institute Europe GmbH where he is currently Project Leader for the group of Brain-like Software Technology. His research interest inlcude software component engineering, real-time computing, middleware, integration environments and biologically inspired software systems.

Christian Goerick studied Electrical Engineering at the Ruhr-Universität, Bochum, Germany, and at the Purdue University, Indiana, USA. He holds a Doctoral Degree in Electrical Engineering and Information Processing from the Ruhr-University of Bochum. During his time in Bochum he was Research Assistant, Doctoral Worker, Project Leader and Lecturer at the Institute for Neural Computation, Chair for Theoretical Biology. The research was concerned with biologically motivated computer vision for autonomous systems and learning theory of neural networks. Dr. Goerick is currently Chief Scientist at the Honda Research Institute Europe GmbH, with responsibility for Embodied Brain-like Intelligence covering research on behavior based vision, audition, behavior generation, robotics, car assistent systems, systems architecture as well as hard- and software environments.