EU FP7 CogX
ICT-215181
May 1 2008 (52months)

DR 7.4:
Design Methodologies for Integrated Cognitive Systems

Graham S. Horn, Marc Hanheide,
Kristoffer Sjöö, Marko Mahnič

⟨cogx@cs.bham.ac.uk⟩

Due date of deliverable:	May 30 2012
Actual submission date:	May 30 2012
Lead partner:	BHAM
Revision:	final
Dissemination level:	PU

Over the course of the CogX project we have built a series of increasingly sophisticated integrated cognitive systems that are capable of self-understanding and self-extension. This has required us to combine several capabilities for sensing, acting and reasoning into a coherent architecture. We present the CogX Layered Architecture Schema (CLAS) which is a new conceptual view of the design that we have used throughout the project. CLAS explicitly separates a system into three functional layers: at the top is the domain-independent deliberative layer; at the bottom is a (mostly) domain-dependent competence layer which contains subsystems that provide (modal) knowledge to the system and may be tasked to carry out actions; and in the middle is the belief layer which combines information from the competences into an amodal, probabilistic representation that is used by the deliberative layer. We also discuss some of the methods that we used to implement systems following this design.

1 Introduction

During the four years of the CogX project we have built a series of increasingly sophisticated integrated cognitive systems which have been described in published papers and deliverable reports[1].

An *Integrated Cognitive System* combines multiple capabilities, including reasoning, perception and action, in order to (robustly) achieve tasks in incomplete, uncertain, dynamic environments. Such a system must be capable of processing inputs from multiple modalities (e.g. vision and range sensors, text or speech) and acting using multiple output modalities (e.g. moving the robot base or sensors, actively capturing images or interacting with people via speech or text-based dialogue). The core capability, however, is the ability to carry out reasoning based on models that the system creates about the world in which it operates.

The overall aim of work package 7 was to create experimental integrated systems and to analyse them systematically. This report addresses the last of the specific objectives for the work package: *Understanding appropriate methodologies to empirically and formally analyse the behaviour of integrated robot systems.*

This report presents the methodology that was used to build two such integrated cognitive systems: *Dora*, a mobile robot that extends its knowledge of its environment [2] and *George*, a robot with vision and manipulation capabilities that learns from a tutor [3]. In the next section, we describe our design for architectures for integrated cognitive systems. We then present some methods that we used when implementing systems following this design. Together these constitute our methodology.

2 CogX Layered Architecture Schema (CLAS)

We base our approach to architectures on that used in the CoSy project[2] because the requirements against which the CoSy Architecture Schema (CAS) [9] was developed were also valid for CogX. In particular, CAS deals with the following *run-time* requirements: *dynamism* – the environment in which the robot operates is constantly changing; *uncertainty* – sensors are inaccurate and the robot's actions may fail to have the consequences that were planned for; *multiple modalities* – information from multiple sensory modalities must be combined in order to make decisions, and multiple action modalities must be used in a coordinated manner; *re-taskability* – the robot must be able to switch to new tasks or interleave them with existing ones, based on goals generated internally or by others.

[1]See http://cogx.eu
[2]http://www.cognitivesystems.org

An architecture built using CAS consists of a collection of subarchitectures. Each subarchitecture contains a number of concurrently active processing components which exchange information via a shared working memory. Software development is carried out using CAST, a toolkit that implements CAS [8]. Information processing is primarily event-driven – components register *change filters* with working memories and receive *change events* when the change filter is matched by a change to the contents of the working memory.

In general, integration architectures following CAS are composed of a number of (usually *modal*) subarchitectures which are all endowed with equal rights, permissions, and policies. However, for systems featuring task-driven self-understanding and self-extension, we need to have a deliberation subarchitecture that considers existing knowledge gaps in the context of the overall tasks given to the system, using unified, *amodal*, representations. This approach (of combining modal representations into unified amodal representations which can be reasoned with) has been taken in both the CoSy and CogX projects. Here we present a new conceptual view of this approach which explicitly maps CAS subarchitectures into *functional layers* – these layers have been implicitly present in the architectures used throughout these two projects. We refer to this as the *CogX Layered Architecture Schema (CLAS)*; as shown in Figure 1. The layers can be considered to be an organisational tool that separates the modal and amodal elements of an architecture.

At the lowest level we have the modality specific *competence layer*. Competences consist of a set of components designed to jointly provide a certain rather well encapsulated and independent behaviour or functionality. A competence can be thought of as a subsystem that *provides knowledge* to the system and which can possibly be *tasked* to carry out actions. Competences are inherently domain-dependent and the ones present in different CogX systems vary. Most of the competences are realised by a single subarchitecture (some subarchitectures may provide multiple competences) but some competences require cross-modal interactions and hence are provided through the collaboration of multiple subarchitectures. For example, the Dora system includes competences to build and navigate maps, categorise places and manage dialogue, which are realised in the dedicated subarchitectures “spatial”, “categorisation”, and “dialogue”, respectively. It also includes competences to search for objects and persons, and these are provided by a combination of components within the “spatial” and “vision” subarchitectures.

The *deliberative layer* at the top of the architecture contains motivation management and planning components . This layer is shared across different CogX systems (both Dora and George) and is designed to be domain-independent. The “switching planner” uses either a continual planner or a decision theoretic planner depending on the robot’s subjective degrees of be-

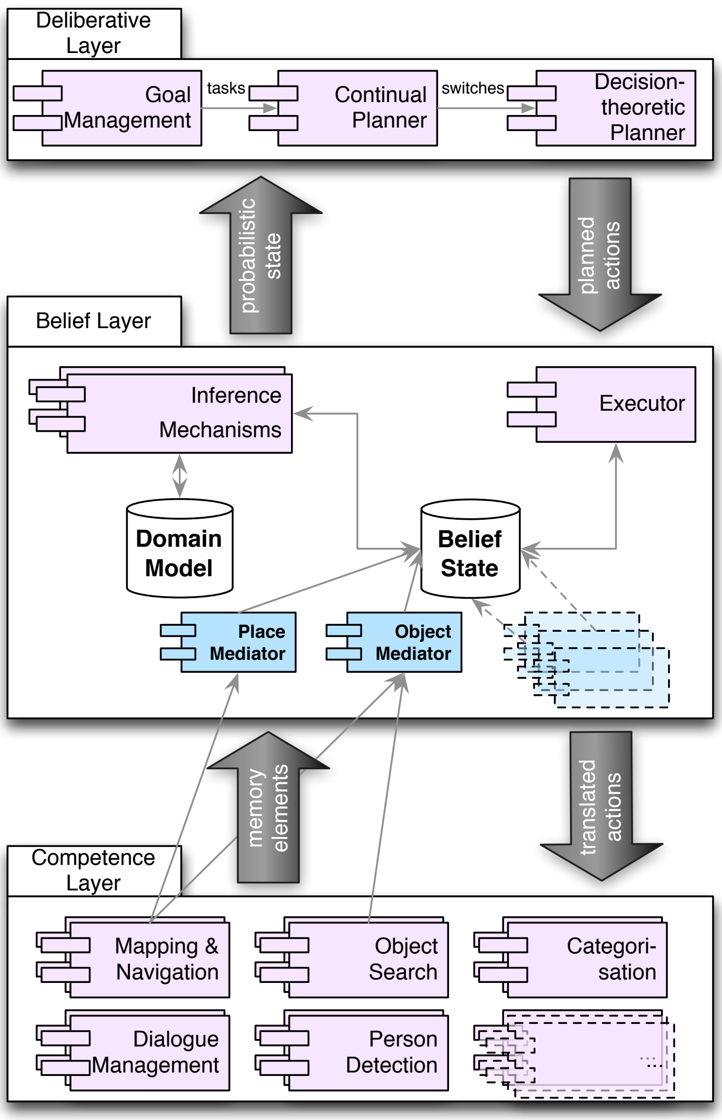


Figure 1: CogX Layered Architecture Schema (CLAS).

lief (see below) and progress in plan execution. Motivation management prioritises the execution of goals given to the robot or generated internally [1].

In between the competence and deliberative layers, the *belief layer* fulfills three main functions: (i) *select* (modal) information gathered through different competences; (ii) *maintain relations* between pieces of information (entities); and (iii) *unify* the gathered and pre-defined knowledge of the system into a formal amodal, probabilistic representation facilitating reasoning and planning within the deliberative layer. The *binding* subarchitecture fills this role in the George system. In the Dora system we consider the conceptual and default subarchitectures to be in this layer.

In CogX, we represent *beliefs* as multivariate probability distributions over logical formulas [14]. They constitute the main representational framework in the belief layer, and this formalism is domain-independent and used by both George and Dora. Multiple inference mechanisms can be used. In Dora, the probabilistic relations between entities are encoded in a *conceptual map* and a probabilistic inference mechanism using chain graph reasoning can infer new, unseen relations [11, 7]. In George, Markov Logic Networks are used to infer relations and references for shared beliefs.

The belief layer as a whole (including the probabilistic relations and the pre-defined knowledge) forms the probabilistic belief state that is then accessible to the deliberative layer, i.e. the planning framework. There is a clear separation between the layers, as the planners cannot access any information in the competence layer. This strict decoupling ensures that the top layer remains domain-independent. To actually devise plans, the belief layer representation is compiled into a DTPDDL [5] state that represents the initial state of the planning problem.

The actual information encoded in the belief layer is domain-dependent, according to the domain model (used for planning) and the mediators (see below) deployed.

The representation and processes that form the belief layer have certain requirements that need to be fulfilled:

- Coverage: Because of the strict decoupling, the belief layer needs to cover every aspect that shall be considered in planning. Anything that is not translated into the belief layer will not be accessible to the planners.

- Traceability: In order to be able to relate any generated plans back to the original information generated in the competence layer, back-references always have to be maintained for any information in the belief layer.

- Recency: The representation needs to be kept up to date, i.e. effects of actions observed in the world need to be propagated through the belief state in a timely way.

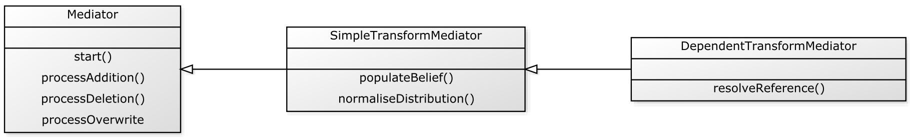


Figure 2: An excerpt from the Mediator class hierachy

- Consistency: The overall state needs to be both structurally and probabilistically consistent. This entails that all entities that are referred to from relations need to exist and be valid, and also that all probabilities fulfill the basic laws of probabilistic calculus, e.g. that probabilities for one random variable sum up to 1.

2.1 Mediation framework

All of these requirements are met by the *mediation framework*, which is implemented as part of the belief layer. The general idea for mediators is that they are small, computationally light-weight, and reusable components that are specialised for different types of information generated on the competence level. There are a number of different abstract types which can be specialised. Coverage is assured by the specialisation that selects exactly the information required from the dedicated working memories and translates it into the belief structures. When we talk about small components here, we refer to the fact that the whole framework is not implemented as one monolithic block, but that for each modal type of information (e.g. the detection of an object announced in the visual working memory) there is one mediator each which monitors the shared memory space in the system for the addition, deletion or modification of any such entity it is registered for. As a consequence, our current systems feature between 5 (George) and 10 (Dora) different mediators. The decomposition into individual components in CAST contributes to the “recency” as updates are dispatched asynchronously and can occur in parallel as all processing in mediators follows the event-driven processing paradigm. Of course this yields issues for synchronisation, and in consequence consistency, which need to be addressed explicitly by those mediators.

As described earlier, there is a hierarchy of abstract components that can be specialised for the individual types (excerpt depicted in Figure 2). Hence, the framework is easily extensible. The most simple abstract mediator is the `SimpleTransformMediator` which monitors one specific type and transforms the modal, domain-specific data type into a belief structure by selecting relevant information. It does not require any other information or relate to anything outside the currently processed entity. It ensures consistency by making sure that any generated distribution over logical forms follow the laws of probabilistic calculus (and normalises accordingly if they don't) and by processing modifications to the belief in the order of events received. Based on this simple mediator a `DependentTransformMediator` implements an abstract class to implement mediators that need to consult other information not available in the entity that raised the event or that have to maintain or form a relation to another belief. These mediators need to make sure that the information they consult or refer to is (i) already present as a belief, (ii) is up to date, and (iii) that the correct reference (realised through unique CAST IDs) is available. Such `DependentTransformMediator`s feature support for this through appropriate synchronisation techniques, that will invoke a separate worker thread that waits for any references to "external" information to be available in the belief state before committing the update itself. This certainly results in delays as there are synchronisation points introduced in the otherwise asynchronous dispatch. However, this is unavoidable if one needs to ensure consistency. These two subclasses are the major abstract mediator types. They also maintain a full history over any entities that contributed to the respective belief to allow to refer back to the original source and implement traceability.

An example is the `ObjectMediator`, which translates detection of an object into a belief about the existence of that object at a particular place in the world. It is a subclass of the `DependentTransformMediator`. Therefore, this mediator conceptually monitors both the "Object Search" and the "Mapping & Navigation" competences. Whenever a new object detection occurs it is published on the "vision" working memory (in accordance with the general CAS processing principles). This sighting will be picked up by the mediator and will be related to the current location, obtained from the "spatial" working memory, to form a belief that comprises a relation to the place where the object has been seen.

3 Methods for developing CLAS systems

In this section we present some of the methods that we developed when implementing actual system architectures following our proposed schema. Systems built following CLAS (using CAST) are inherently concurrent and this can lead to complex interaction patterns between components. Sections 3.1 to 3.3 describe design patterns that can be used to control the interactions between components via command strucutures on working memories. Section 3.4 and 3.5 describe how the shared state space provided by the working memories can be exploited for testing, debugging and visualisation using tools that we have developed in CogX.

3.1 Synchronisation

We have described how the mediators provide synchronisation between knowledge held within individual competences and knowledge held in the belief layer. There is also a need to provide a degree of sychronisation between the planner and the competences which carry out actions specified in the generated plans, in order to avoid frequent changes to the data on which the planner operates. Crucially, if the planner is triggered while the robot is still moving, the resulting plan is likely to be based on outdated information, leading to re-planning and unnecessary processing. It may also cause the planner to conclude that an action it has just taken was unsuccessful, erroneously, because the results of the action did not yet have time to appear on the working memory.

Hence the need to ensure synchronisation between, for example, the movement competence (provided by the spatial subarchitecture) and the planner. This is achieved by placing *command structures* on working memory at the competence level. The planner writes actions for execution (which are translated by the (domain specific) Executor (cf Figure 1) into actions understood by the competences), and it is up to the executing competence to guarantee that all of the effects of the action are present, and that the state is stable, before the action is flagged as complete. Inside the subarchitecture itself, the action is decomposed into various activities, also controlled by command structures placed on the working memory. These too carry completion flags which are overwritten once the activity is complete, allowing synchronisation both with the planner and within the subarchitecture where necessary.

3.2 Component interaction choices

One of the key features of CAS (and therefore also CLAS) is that information is shared between components by writing it in structures held in working memories that are (potentially) accessible for any other component to read and update. This has the benefits of reducing the coupling between components and of exposing information for introspection by additional components (e.g. for visualisation, see section 3.5).

Plan execution in a robotic system involves controlling hardware – for example, using a manipulator to pick up an object or moving the robot base to a new location. The closed loop control necessary for such subsystems is not naturally expressed using interactions on working memories. Similarly, interfacing with sensors is more naturally done using a remote procedure call (RPC) framework to inject sensor data into components in the competence layer. To this end, an RPC facility was added to CAST to provide both another form of publish-subscribe communication (the other being registration for changes on working memories and subsequent propagation of change events) and service calls to trigger processing in specific *Server* components.

The existence of the RPC framework led to some developers choosing to use it for interactions between components within the competence layer. This was done for two main reasons: for convenience (because it is easier to think in those terms rather than in terms of working memory interactions); and for performance (holding the obstacle map in a single component which can perform calculations on behalf of other components is faster than sharing the obstacle map via working memory). However, undisciplined use of RPC can lead to complex inter-dependencies between components and, moreover, it breaks the information sharing approach by passing data directly between components.

A design pattern was therefore developed to implement service calls over working memory (WM) interactions via a request-reply protocol. The protocol is based on WM entries that have three parts: the request, the result and the status. The request and the result are arbitrary data structures. The status represents three general states of the WM entry: 'init'; 'succeeded'; and 'failed'. The caller starts the communication when it writes a new WM entry. The initial status of the entry is 'init' and the request field is filled. Then the caller waits for an overwrite change event on the written entry. In some cases the caller component implements a time-out which aborts the communication if the processing of a request takes too long. In this case the overwrite event is ignored, but the callee is not notified. A 'cancel' state could be added in the future, to avoid nugatory processing in the callee. The callee waits for a WM-ADD event. When it arrives, the parameters of the call are read from the WM entry and processed. If processing is successful the callee fills the result part of the WM entry and sets the status to 'succeeded'. Otherwise the result part is invalid and the status is set to 'failed'.

Using this request-reply protocol allows developers to avoid the need for protective measures (such as locks on data structures) in the server component.

3.3 Transactions

A transaction groups multiple interdependent actions into an atomic unit of work in order to ensure that the state of the system is maintained in a consistent state. Event-based systems do not naturally support transactions, so a design pattern has been developed to allow these to be carried out efficiently. The pattern is to use a dedicated control structure on working memory, as described below.

An example transaction in the Dora system is *map loading*. In many cases, it is desirable to start the system with a degree of knowledge already gathered from earlier runs. As this knowledge is stored distributed over multiple components, some care is required when restoring the knowledge state.

The basic method is for each component to load its state from a file, and process the items as they arrive, in the same manner as when they are created during a live run. However, this can be problematic when the knowledge in one component is in relation to that in another. For example, observations of the shape properties of a Place are naturally dependent upon the existence of that place; similarly, knowledge pertaining to rooms is dependent upon the Places constituting that room. When the robot is running, these dependencies are satisfied naturally by the order in which each item is first created.

A related issue is that some components do expensive processing in reaction to each new map item added to working memory; processing which is overridden if more data is added immediately afterward. This includes the conceptual map inference procedure which runs afresh with each new Place added. During map loading the intermediate map states are of no use to the system and represent a waste of resources.

To address these issues a *map loading control structure* is created on the working memory. This structure has flag fields which allow each component involved in loading a map to signal when it has completed its own portion of the load process, and has written to WM the entirety of its loaded state. Other components dependent on this state can then proceed with their own intialisation.

3.4 Testing and Debugging

Developing and debugging concurrent systems that consist of large numbers of components (about 100 in Dora v4) is hard. We adopted standard best practices of using a continuous integration system[3] to build the software and run tests every night (using the Stage simulation[4]). We regularly ran the systems on the real hardware in realistic environments.

Debugging also needs introspection of state at runtime. One of the significant benefits of having a shared state space is that it is easy to write additional components that can monitor working memory interactions and display working memory contents. We ported the data-driven failure detection approach developed at Bielefeld University [6] to the Dora system and a final-year undergraduate student at the University of Birmingham carried out some initial investigations as to its efficacy for detecting failures due to the absence of expected working memory interactions. Early results were promising although we have some concerns over its relevance to the types of system errors that we tend to encounter most often. We developed a Working Memory Editor component that uses the Java reflection API to both display the contents of selected working memory contents in a human

[3]http://jenkins-ci.org/

[4]http://playerstage.sourceforge.net/

readable format and also enable the contents to be changed at runtime – which is useful for generating test cases.

3.5 Visualisation

The use of visualisation tools is essential, both during development and deployment (in our case, when carrying out experiments or demonstrating the system's capabilities). A visualisation and user interface framework has been developed that was designed to support complex, potentially distributed, systems such as those we have developed.

The framework is client-server based. The main part of the framework is the Display Server with a public API implemented on top of the TCP/IP protocol. The clients can establish a 2-way communication with the server. Visualisation data is sent from the clients to the server while user interface events are optionally sent from the server to the clients. The server manages the received data and composes the data into configurable 2D, 3D or HTML views that are displayed on demand. A more detailed description of the visualisation framework is given in Annex A.1 [10].

For visualisation of data with a metric spatial interpretation in the Dora system, the open-source tool *Peekabot*[5] is used. Peekabot is a client-server architecture where the server does the displaying, and multiple clients may connect over a network to add different 3-dimensional graphical primitives. This feature is very useful in a system with multiple, separate components, since it allows each to provide its own data for visualisation independently. This is especially true for debug visualisation.

The system instantiates a central visualisation component that is tasked with displaying spatial knowledge that appears on the working memory – including the robot and its position, places, and objects, as well as inferred conceptual information. Reading this information from the WM ensures that visualisation accords with what all the other components of the system receive; its "public state". By constrast, information displayed directly by the various processing components themselves pertains to their respective *internal* state, and can be used to verify the proper functioning of the component itself; this includes such things as raw sensor data and obstacle maps. Figure 3 shows an example of Dora visualisation in Peekabot.

[5]http://www.peekabot.org/

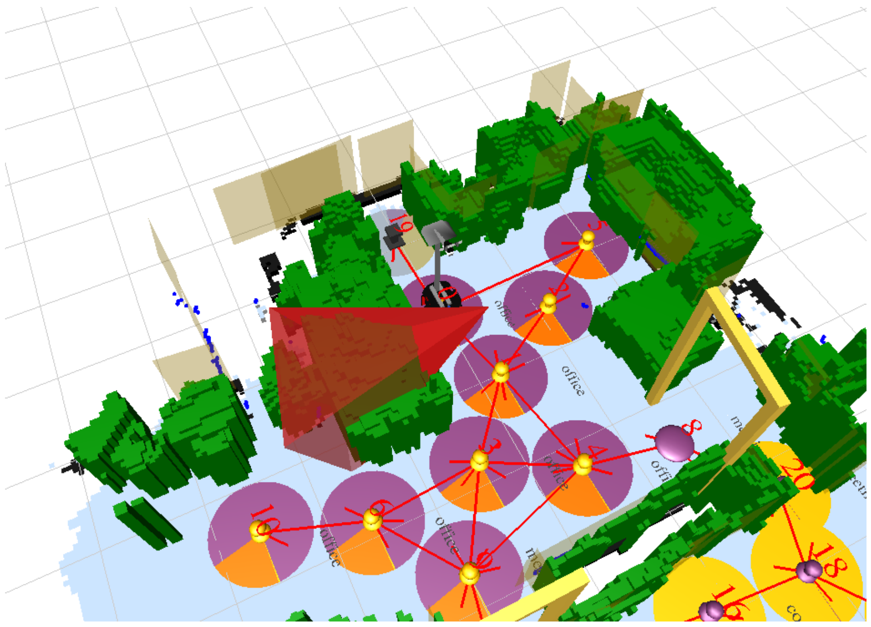


Figure 3: Peekabot visualisation for the Dora system. Obstacles are shown as green blocks. The Places used for planning are shown as pins on the ground, together with the paths connecting them. The “pie charts” around each Place represents the confidence in various classifications of the room in question. Blue dots and beige rectangles show laser scans and SLAM line features, respectively, while a red pyramid shape represents a view cone used to search for an object. Each component in the system can contribute its own visualisation independently.

4 Conclusion

We have described our methodology for building integrated cognitive systems using layered architectures that separate domain independent functionality for deliberative processing (goal management and planning) from (typically) domain specific competences that provide (typically modal) knowledge about the world and may be taskable to act in the world. The separation is achieved by combining knowledge from the competences into a probabilistic belief state that is used by the deliberative layer.

The CogX Layered Architecture Schema is a subspace of the CoSy Architecture Schema. Each of the layers consists of one or more subarchitectures which each has a working memory. In this layered design, access to the shared working memories is restricted to components in adjacent layers only.

The integrated cognitive systems that we have produced, George and Dora, show that this approach can be used successfully to support the type of knowledge handling required for a system that reflects on its knowledge, detects knowledge gaps and plans for knowlege gathering actions. In particular, the deliberative layer, the representations in the belief layer, and many of the competences, are common to both systems.

There is a clear tension in a cognitive robotic system between the needs of the higher and lower levels. The high-level, deliberative, part of the system requires a centralised representation of beliefs for which the CAST approach of shared working memories (shared knowledge and shared state) has been shown to be a good one. For low level components that consume sensor data or control hardware, a more natural approach would be to use publish/subscribe or service calls. For components within the competence level a mixture of the two approaches can be appropriate. The approach taken by the NIFTi project [4] is to use CAST for high-level, cognitive, components and ROS [12], which provides both stream-based communication and services, for lower-level subsystems.

We believe that the event-driven paradigm, used by CAST, is well-suited to support a number of different interaction patterns such as working memories, data publishing, and also remote procedure calls. This has also been considered by other recent middleware frameworks such as the Robotics Service Bus[13]. CAST could be extended to provide the state-free message passing that ROS offers (through *topics*) – for example, by adding "non-persistent" memories that allow components to "add" (publish) data which is propagated to subscribing components – in order to use a single middleware for the whole system.

References

[1] CogX consortium. Deliverable DR 1.3: Architectures and representations for introspection and motive management in a robot. Technical report, CogX, 2011.

[2] CogX consortium. Deliverable DR 7.3: Analysis of a robot that explains surprise. Technical report, CogX, 2012.

[3] CogX consortium. Deliverable DR 7.5: A curiosity driven self-extending robot system. Technical report, CogX, 2012.

[4] NIFTi consortium. DR 7.1.3: Integration and end-user evaluation for human-instructed exploration. Technical report, NIFTi, 2011.

[5] Moritz Göebelbecker, Charles Gretton, and Richard Dearden. A switching planner for combined task and observation planning. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence (AAAI 2011)*, page NA, 2011.

[6] Raphael Golombek, Sebastian Wrede, Marc Hanheide, and Martin Heckmann. Online data-driven fault detection for robotic systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, USA, 2011.

[7] Marc Hanheide, Charles Gretton, Richard W. Dearden, Nick A. Hawes, Jeremy L. Wyatt, Andrzej Pronobis, Alper Aydemir, Moritz Göbelbecker, and Hendrik Zender. Exploiting Probabilistic Knowledge under Uncertain Sensing for Efficient Robot Behaviour. In *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2011.

[8] Nick Hawes and Jeremy L. Wyatt. Engineering intelligent information-processing systems with CAST. *Advanced Engineering Informatics*, 24(1):27–39, 2010.

[9] Nick Hawes, Jeremy L. Wyatt, Mohan Sridharan, Henrik Jacobsson, Richard Dearden, Aaron Sloman, and Geert-Jan Kruijff. Architecture and representations. In Henrik I. Christensen, Geert-Jan M. Kruijff, and Jeremy L. Wyatt, editors, *Cognitive Systems*, volume 8 of *Cognitive Systems Monographs*, pages 51–93. Springer Berlin Heidelberg, April 2010.

[10] Marko Mahnič and Danijel Skočaj. A visualization and user interface framework for heterogeneous distributed environments. In *MIPRO, 2012 Proceedings of the 35th International Convention*, Opatija, Croatia, May 2012. **Attached paper, Annex A.1**.

[11] Andrzej Pronobis and Patric Jensfelt. Large-scale semantic mapping and reasoning with heterogeneous modalities. In *Proceedings of the 2012 IEEE International Conference on Robotics and Automation (ICRA'12)*, Saint Paul, MN, USA, May 2012.

[12] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[13] Johannes Wienke and Sebastian Wrede. A middleware for collaborative research in experimental robotics. In *IEEE/SICE International Symposium on System Integration (SII2011*, Kyoto, Japan, 2011.

[14] Jeremy Wyatt, Geert-Jan Kruijff, Pierre Lison, Michael Zillich, Thomas Mörwald, Kai Zhou, Michael Brenner, Charles Gretton, Patric Jensfelt, Kristoffer Sjöö, Andzrej Pronobis, Matej Kristan, Marko Mahnič, and Danijel Skočaj. Deliverable DR 1.2: Unifying representations of beliefs about beliefs and knowledge producing actions. Technical report, CogX, 2010.

A Annex

A.1 A Visualization and User Interface Framework for Heterogeneous Distributed Environments

Bibliography Marko Mahnič and Danijel Skočaj. A visualization and user interface framework for heterogeneous distributed environments. In *MIPRO, 2012 Proceedings of the 35th International Convention*, Opatija, Croatia, May 2012. **Attached paper, Annex A.1**

Abstract Systems that require complex computations are frequently implemented in a distributed manner. Such systems are often split into components where each component is employed to perform a specific type of processing. The components of a system may be implemented in different programming languages because some languages are more suited for expressing and solving certain kinds of problems. The user of the system must have a way to monitor the state of individual components and also to modify their execution parameters through a user interface while the system is running. The distributed execution and programming language diversity represent a problem for the development of graphic user interfaces.

In this paper we describe a framework in which a server provides two types of services to the components of a distributed system. First it manages visualization objects provided by individual components and combines and displays those objects in various views. Second, it displays and executes graphic user interface objects defined at runtime by the components and communicates with the components when changes occur in the user interface or in the internal state of the components.

The framework was successfully used in a distributed robotic environment.

Relation to WP In this paper we present a client-server oriented visualisation framework that is used for monitoring the operation of a running system and is an important tool for the integration (WP7) and presentation (WP9) of the system.

A Visualization and User Interface Framework for Heterogeneous Distributed Environments

Marko Mahnič, Danijel Skočaj
University of Ljubljana, Faculty of Computer and Information Science

***Abstract*—Systems that require complex computations are frequently implemented in a distributed manner. Such systems are often split into components where each component is employed to perform a specific type of processing. The components of a system may be implemented in different programming languages because some languages are more suited for expressing and solving certain kinds of problems. The user of the system must have a way to monitor the state of individual components and also to modify their execution parameters through a user interface while the system is running. The distributed execution and programming language diversity represent a problem for the development of graphic user interfaces.**

In this paper we describe a framework in which a server provides two types of services to the components of a distributed system. First it manages visualization objects provided by individual components and combines and displays those objects in various views. Second, it displays and executes graphic user interface objects defined at runtime by the components and communicates with the components when changes occur in the user interface or in the internal state of the components.

The framework was successfully used in a distributed robotic environment.

I. Introduction

Complex integrated systems often have to perform different complex tasks in parallel. For example, an intelligent cognitive robotic system has to be able to execute various tasks, which may heavily differ in complexity, requirements, and implementation of the corresponding solutions. Because of the complexity such a system is often implemented as a set of components that run in different threads across different processes running on different computers. Each component might be developed in a different programming language, either because the problem it solves is best expressed in that language or because of a developer's preference.

The operation of the system has to be monitored and controlled while the system is running. Often a graphic user interface (GUI) provides both the means to visualize the results and the inner state of the system and a set of actions that the user can execute to modify the behavior of a running system. It is very important for such a system to offer a user-friendly user interface that enables efficient interaction with the system. It is of equal importance to enable visualization of additional information about the execution of the system, at different levels of detail, which also enables efficient debugging and thorough analysis of such an integrated system. Due to the distributive and non-homogeneous nature of such a system, a centralized control and monitoring of the execution is a very challenging task.

In the research community the importance of development of efficient visualization tools and user interfaces is often under emphasized. Since the main focus of interest in research is targeted on the research work and problem solving, less attention is usually payed on the development of user interfaces (UI). This is even more true when the research involves multi-threaded or distributed systems in which a component and its UI run in distinct processes on separate computers. Quite often the component provides some kind of UI while it is being developed on its own but when the component runs with other components the UI is disabled because of incompatibility with the rest of the system. While such components could be controlled and monitored when developed and used in isolation, such an approach is completely inappropriate when the component is used as a part of a wider integrated system. It does not allow for display of information that the user would require, nor does it allow for efficient debugging of such a system.

In this paper we present our solution to this problem. We have developed a framework[1], that enables efficient visualization and UI management in heterogeneous distributed environments. The framework provides a server whose primary task is the management and rendering of objects that represent the information about the results or inner state of a running system at different levels of detail. The display objects are provided by different components and are combined by the server into views which are rendered in an arbitrary number of frames. The server displays UI elements such as tool buttons and dialog windows provided by the components and notifies the components when actions are performed on those elements, thus enabling the control of the components.

The paper is organized as follows. In the following section we define the problem and then present our solution in Section III as well as implementation details in Section IV. In Section V we describe the implementation of the proposed framework in a distributed robotic system. We conclude the paper with the final discussion and concluding remarks in Section VI.

II. Problem definition

As soon as the processing is distributed across multiple threads, the complexity of the user-interaction subsystem

[1]The source code is available at https://github.com/mmahnic/cogx-display-server.

raises. Even more so when the application is split into multiple processes running on multiple computers (*nodes*) possibly under different operating systems and using different programming languages for the development of components that constitute the system. In a highly configurable and scalable system the components can be configured to run all in a single process or each in a process dedicated to it. Running a component isolated in a process is desired when the component is being developed and tested. In a production environment, especially when the components share a lot of data it may be more efficient to execute many of them in a single process using multiple threads.

In a GUI application based on a standard framework like Qt, GTK or Java Swing, there is usually a dedicated thread that processes UI events. The developer has to be careful not to do too much processing in the event handling procedures, otherwise the UI code doesn't run often enough and the interface becomes unresponsive. Instead, working threads are created that process the data and pass the results back to the event processing thread for display or further processing. The resources for visualization are shared between multiple threads and the developer must pay a lot of attention to proper inter-thread protection mechanisms.

Standard frameworks like Qt or GTK allow only one event thread so loading two or more components that implement their own UI would create additional event threads and crash the application. For this reason the UI is usually implemented in a separate component, which is often part of the main application. This means that each component has to be developed in two parts: one part that does the actual work on a remote node and one that provides the UI for the component. The UI part can be either a source-level module for the main application or a *plugin* that can be loaded into the main application at runtime.

The two parts of the component may be developed in different languages. Because of this it is usually impossible to establish the communication between the parts using communication protocols provided by the framework like Java Remote Method Invocation (RMI). Instead, special protocols have to be developed or a more generic communication infrastructure has to be used.

By analyzing these problems we arrive at the following technical requirements that the visualization and user interface framework should fulfill:

- It should make the visualization of different information coming from different parts of a distributed system centralized and clear.
- It should reduce the amount of communication needed to transfer the visualization data to the server.
- It should reduce the problems of UI inter-thread communication.
- It should be scalable in the sense that new components and new servers can easily be added.
- It should allow simple extension of existing components to support such kind of visualization and UI.
- It should support the development of components in different programming languages.

We are not aware of any system that would fully meet these requirements and that would use a dedicated server for the management and execution of user interfaces defined by remote components at runtime. The technologies that are the closest to this idea are the X-Window System and Web Browsers. In X-Window System a client application can be configured to export it's display to a different machine. This requires that a component is implemented as a GUI application which is most often not desired. Web browsers display user interfaces that are generated by remote servers. Until recently only simple interfaces could be created but with the emergence of HTML 5 and WebGL [1] this will improve. For security reasons a Web browser page can only receive data from a single server so it can't communicate with each component separately. Instead an additional intermediate server is required to integrate data provided by multiple components.

In the development of robotic systems the Robot Operating System (ROS) [2] is becoming widespread. It supports two types of visualization. The first type is the off-line visualization of the messages of various types that were sent between individual components and were stored in a log file. The second is the on-line 3D interactive visualization of the systems state. The client interfaces exist in C++ and Python.

Other applications that are more task specific, such as Peekabot [3], are also used for 3D visualization, however they are not as general as the framework presented in this paper and they do not address all the requirements listed above. Having these requirements in mind, we have designed the system that is described in the following section.

III. A Distributed System

As shown in Figure 1 the system has a central display server that communicates with multiple local and remote client processes. In every client process a set

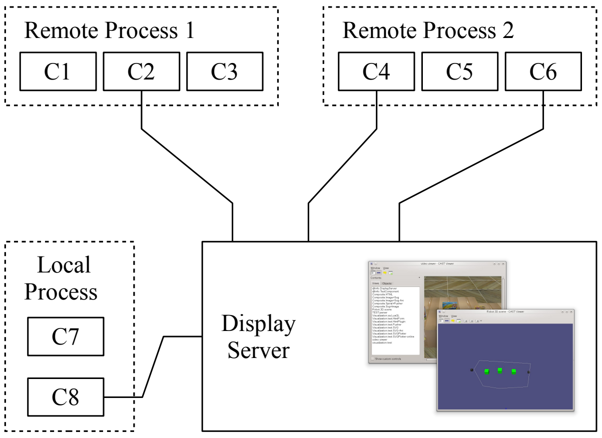


Fig. 1. Components running in multiple local and remote processes send objects to the display server and receive events from the UI elements.

of components is running each in its own thread. The mapping between processes and components is arbitrary although the components that process similar data are often instantiated in the same process.

The components generate and process data. The data may represent the end result of the processing or serve as input for other components. One could imagine that several types of data could be considered. Because of the variety of data and its interpretation, the display server doesn't try to visualize raw data. Instead, the client components transform the raw data into display objects (raster images, SVG images, 3D models, HTML documents) and send them to the server. The server keeps the objects until they are replaced or deleted and combines them into user-defined views. The display server creates windows on demand and offers a quick way to switch between different views in every window.

The components that support user interaction prepare the UI elements and send them to the display server which stores them and displays them when necessary. Simple UI elements are associated with views and are displayed in a dynamic toolbar of a window whenever an associated view is displayed in the window. Dialog forms are usually more complex so they are displayed in special dialog windows. Each dialog form is displayed on a separate page of a tabbed control.

HTML documents can also be used for user interaction if they contain specially prepared elements. In this case the display server injects additional code into the document that generates events for the client component when an active element is clicked or sends the values of the fields of a form to the component when the form is submitted.

IV. Implementation Details

A. The Overall Structure

The display server is internally split into three loosely coupled components: the public Ice interface, the model and the GUI. The significant components of a client and a server are shown in Figure 2.

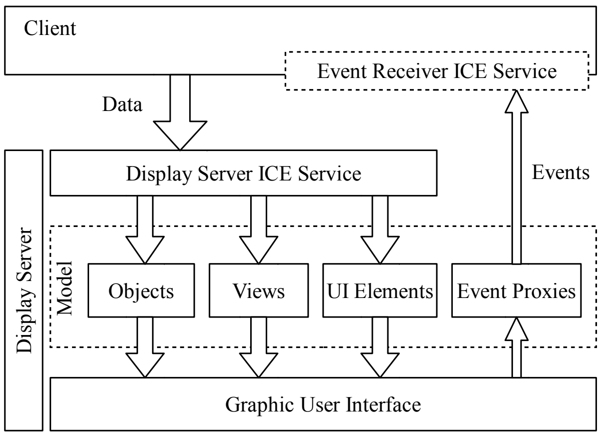


Fig. 2. The architecture of the framework

The public interface provides a proxy through which the clients can send requests for adding and modifying display objects, views and UI elements that are managed by the server. Each connection from a client runs in a new thread, but the threads are protected from each other by the use of locking mechanisms.

The objects are stored in various containers in the model of the display server. The model implements only the storage for the objects and is not aware of the types of objects it stores. This makes the model independent of the framework chosen for the implementation of the GUI.

The GUI is implemented with the Qt framework [4]. The framework was chosen because it is one of the most widespread open source C++ frameworks supported on multiple platforms. It offers excellent support for visualization of data in 2 and 3 dimensions and it embeds WebKit [5], one of the most advanced web browser engines. The communication between the Qt UI elements (widgets) is based on signals that are emitted when various events occur. The signals integrate easily with other non-UI elements and they were used to implement the event proxies in our framework.

The development environment of the Qt framework provides also and excellent form designer. The designed forms are stored as XML documents that can be compiled into native code of an application or loaded into a running application and instantiated without compiling. On top of all a scripting engine is integrated into the framework with which it is possible to create and execute code on the fly.

B. The Communication Interfaces

The distributed system is built on top of the Zero C's Internet Communications Engine (Ice) [6], [7] which provides the infrastructure for communication between the components of the system. The communication is based on remote method invocation (RMI). The communication interfaces (structures, classes and service interfaces) are defined in Specification Language for Ice (Slice). Slice compilers generate communication interfaces for many programming languages and the generated code is easily integrated into the components.

The display server Ice service provides only one-way communication so the clients can only send the data to the server. The clients that want to handle events emitted by the UI elements they create, have to instantiate an Event Receiver Ice service and register it with the server. Once the connection is established the server starts notifying the client about the UI events.

C. Views

A view is an entity that holds a list of visualization objects to be displayed in a window of the graphic user interface. It is associated with one of the rendering context types: 2D scenes, 3D scenes and HTML documents.

A view is hidden until the user selects it for display in a window. At that point a rendering context is created for the view. When the objects of a view are rendered, each object is asked to provide a renderer for the view's rendering context. If an object can't provide a renderer, it is skipped.

When a view is being displayed, individual objects and/or their parts can be hidden by the user. The information about the visibility is stored with the view since an object can be displayed in multiple views and keeping visibility information with the object would hide that object in all views. A view is also associated with a list of simple UI elements defined by the clients. The UI elements are added to a dynamic toolbar when the view is shown in a window and removed when the view is hidden. Currently only tool buttons are supported.

D. The User Interface Elements

When a component has complex settings that require more than simple tool buttons, forms can be used. The framework takes advantage of the Qt XML format that is used to describe the user interfaces and of the scripting engine.

The component that needs a form-based interaction with the user sends two documents to the display server. The first is the XML document with the design of the form and the second is the JavaScript code that represents the logic of the form. The display server instantiates the form and waits for the user to interact with it.

The communication between the component and the form is implemented in the following manner. Whenever a value has to be passed from the form to the component the function provided by the display server that accepts an ID and an object is called from JavaScript. This function transforms the object into JavaScript Object Notation (JSON) [8] and sends it to the client as an event with the ID and the data. The component can set the values of the form if it sends JavaScript code to the display server. The code is executed in the scripting engine that belongs to the form and executes its logic.

E. Visualization Objects

The display server supports the following basic types of objects: raw and encoded raster images, SVG images, HTML documents and Lua OpenGL scripts. All objects except raster images can be composed of multiple parts which can be manipulated individually by the client. This greatly reduces the amount of data that has to be transferred from a client to the display server.

1) 2D Objects: Raster images are transfered from a client to the server as arrays of bytes. Images may be in any format supported by the Qt library. When the server receives a raster image it converts it to an uncompressed RGB image which is suitable for fast rendering.

SVG images were added to the system as a way to overlay annotations on raster images. For this reason the display server renders only self-contained SVG documents that don't reference any external resources like raster images. To add annotations to an image the client creates an SVG document with the annotations and defines a view that includes both the raster image and the SVG document.

2) HTML Objects: HTML objects are composed of multiple chunks that belong either to the head or to the body of the document. The chunks are merged into a full HTML document every time any of the chunks is updated, but only if the document is being displayed in a window and the auto-refresh option is enabled in the GUI.

HTML chunks can contain active elements such as forms or (active) HTML tags that notify a display client on `onClick` events. When such chunks are rendered into the full document additional JavaScript and HTML code is added to the document. This code generates events for the display client when an active element is clicked or a form is submitted.

3) 3D Objects: To keep the Ice interface simple we use the Lua programming language [9] with the LuaGL extension [10] to represent objects in our visualization of 3D scenes. Lua is a scripting language that is used in a wide range of industrial applications and in particular it is a leading language in game development. It is simple, fast and easily embeddable into applications. It's written in strict ANSI C so it is quite easy to write extensions in C/C++. The LuaGL extension maps OpenGL C functions and constants to Lua. With this it is possible to define 3D objects and entire scenes by calling OpenGL functions from Lua scripts.

3D objects are composed of multiple parts and every part is a Lua script. The scripts are created by the clients and sent to the display server. A LuaGL object is rendered into a 3D context by executing the scripts of all its parts.

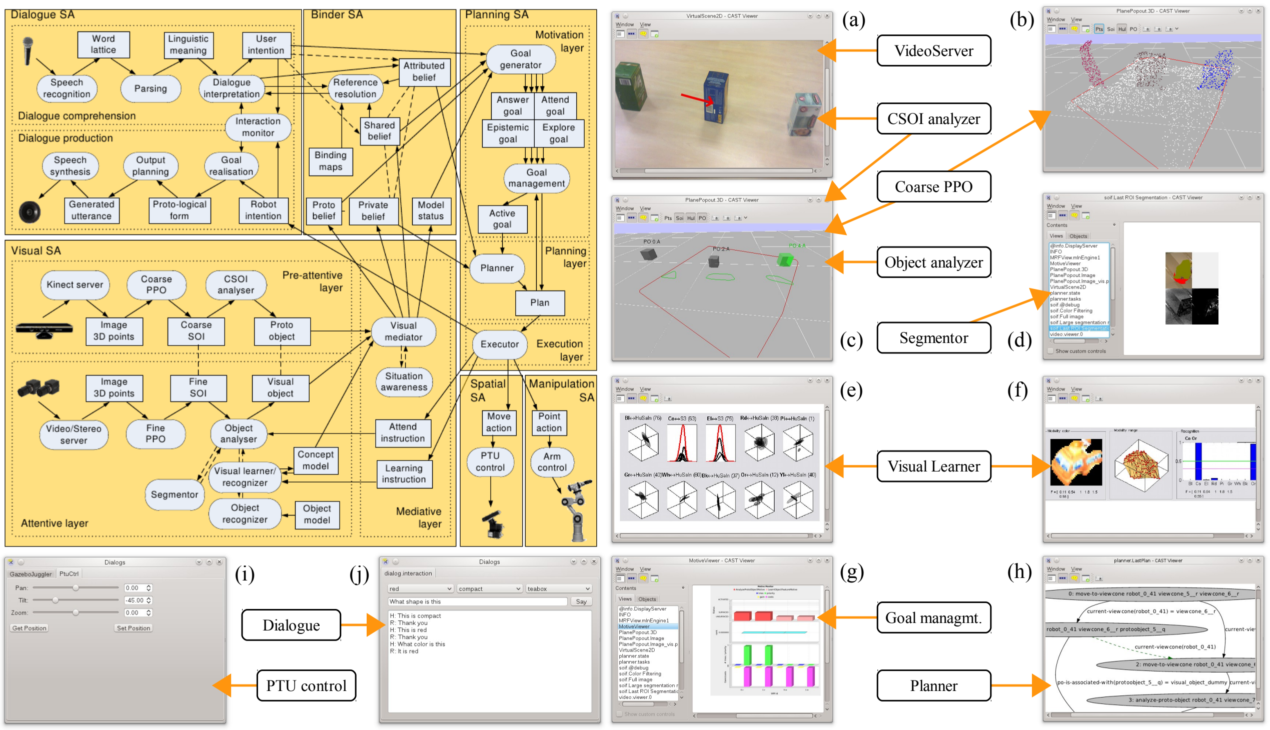
In a complex 3D scene it is sometimes difficult to find a view that shows it in a way that gives the most information to the user. For this purpose a display client can define positions and orientations of the camera from which the objects it creates are best visible. These camera positions show up in the display server GUI as tool buttons and the user can easily switch between different views of the scene.

V. Implementation in a Robotic Environment

A. Description of the Environment

The cognitive robot that we have developed as a part of the CogX project [11] is a mobile platform that uses various sensors such as range lasers, odometers, video cameras and microphones. The cameras are mounted on a pan-tilt unit (PTU) and a robotic arm is mounted on the robot base. The robot is used in different scenarios employing a different set of sensors and performing different operations in each task. Because of the diversity and complexity of the tasks the system was designed to work as a set of components distributed over a network of computers (*nodes*).

The system is based on the CoSy Architecture Schema Toolkit (CAST) [12] that provides the environment in which the components can communicate with each other. The communication is based on a set of working memories (WM) that also serve as the containers of entries that represent the robot's state and knowledge. The system doesn't have a central server. Instead a set of CAST



Dialogs

dialog.interaction

red | compact | teabox

What shape is this [Say]

H: This is compact
R: Thank you
H: This is red
R: Thank you
H: What color is this
R: It is red

Fig. 3. The components of a cognitive robotic system. The large area on the top left represents the components, the connections between them and the data they produce. On the right and on the bottom are the windows of the display server showing different views, objects and dialogs created by the components.

servers is started on every node, one for each supported programming language: C++, Java and Python. To load the components into the servers a CAST client is used that reads a configuration file with the descriptions of the components and sends to every CAST server information about which components to load and what configuration parameters to pass to them.

B. *Visualization of the System's State*

The components of a cognitive robotic system that learns in interaction with a tutor about the properties of objects [13] are displayed in the top left part of Figure 3. The components generate various visualization views which are also displayed in the figure. Before the display server was introduced every component managed its own interface. Components that had a GUI interface created a number windows on multiple computers. Components used a logging system or even terminal output to display meaningful information about their state. This way the information about the system's state was spread across multiple graphic or terminal windows on different computers. Furthermore the components were not able to cooperate in the generation of complex views.

The centralized visualization server we have developed significantly alleviated the problems mentioned above and solved the visualization problems in a systematic way. In the following we describe the visualization interface that we introduced into the system and show a few example views that are displayed during the execution of the system. A very brief description of some of the processes is also given to better illustrate the type and diversity of the data that is being displayed. A detailed description of the system and individual components is omitted, since it is beyond the scope of this paper.

In Fig. 3-a we see an example of cooperation of two components. The Video Server generates images that are displayed in a 2D view and are annotated with an SVG line provided by CSOI analyzer that shows the direction of the next camera move. In Figures 3(b-c) the same view is displayed but with different objects enabled with the tool buttons of the dynamic toolbar. The view is built with the cooperation of three components. The Coarse PPO displays the 3D point cloud (Fig. 3-b), the detected principal plane (red line) and the spaces of interest (SOIs) which are potentially the objects that the robot learns about. The components CSOI Analyzer and Object Analyzer process the SOIs and create new entities that represent stable SOIs and recognized objects. The objects from all three components are displayed in a 3D view shown in Fig. 3-c.

The Visual Learner is the component that has the knowledge about object properties. Its input are segmented images which are provided by the Segmentor component. Segmentor extracts an area which represents the object from a video image using the information stored in a SOI. The result of the segmentation is displayed in a 2D view shown in Fig. 3-d. On the left side of the window in Fig. 3-d a list box with a list of available views is shown. Typically a running system creates more than 40 different views and the list box enables the user to quickly switch between the views.

The models that the Visual Learner is using for the recognition and represent the knowledge are shown in Fig. 3-e. After a successful recognition an image with the recognition results is created which is shown in Fig. 3-f. The Visual Learner is written in Matlab and doesn't communicate directly with the display server. Instead it writes the figures and the HTML documents to the disk where they are picked up by the File Monitor component (not shown) which sends them to the display server every time they are changed. With the use of the File Monitor any application that can write a file can be integrated with the framework with only minor modifications.

Fig. 3-g shows a 2D view with an SVG document that displays the settings of the motivation subsystem and the status of the goals that the robot may achieve. A surfaced goal is passed to the Planner that creates a plan with a set of actions that lead to the achievement of the goal. An example of a graph for a plan is shown in Fig. 3-h. The displayed plan is an SVG document generated by Graphviz [14] which is called by the File Monitor every time the Planner writes a file with a textual description of the graph.

Two dialog forms are shown in Figures 3(i-j). In Fig. 3-i a form created by the PTU control enables the user to manipulate the pan-tilt unit manually. In Fig. 3-j a form created by a component from the Dialog SA provides a way to interact with the robot by typing and is used when speech recognition is not reliable.

Additionally the server manages various HTML documents that are generated by different components. One kind of document is a table of WM entries of a certain type that is created by a WM Viewer and is composed of WM entries serialized into HTML chunks that represent table rows. One document is created for each type of WM entry that the user wants to monitor. The Component Status document holds information about the states of the components that provide the chunks for the document. The Visual Learner component uses an HTML document with buttons and loads a different database when a button is pressed. Each component can generate an arbitrary number of additional HTML documents. This way the user can monitor the state of individual components as well as of the entire system all the time, at any level of detail.

VI. Conclusion

In this paper we addressed the problem of visualization and user interface in heterogeneous distributed environments. It is important for such a system to offer a user-friendly user interface that enables efficient presentation of results and interaction with the system, as well as visualization of additional information about the execution of the system, at different levels of detail, which also enables efficient debugging and thorough analysis of such an integrated system. In this work we analyzed the requirements of a visualization and user interface framework, presented the designed solution that fulfills these requirements and described the implementation details. The framework provides a server whose primary task is the management and rendering of objects that represent the information, which is supposed to be visualized to the user. The display objects are provided by different components and are combined by the server into views which are rendered in an arbitrary number of frames. The server also displays UI elements such as tool buttons and dialog windows that enable the user to control the components.

We showed how the proposed framework was used in the case of an integrated cognitive system. It proved to be very useful for the development and use of this system. The centralized visualization server enables more efficient and focused monitoring of the execution of the system, which significantly helped to solve problems on the system level, as well as to debug the individual components while working in the integrated system. In this way, the visualization and user interface server facilitated integration work and helped to increase the robustness of the system, as well as significantly improved the user experience for the final user of the system.

Since the design and implementation of the proposed framework are very general, the presented solution can be applied to a variety of problems that are modeled with heterogeneous distributed systems. It is relatively easy to integrate it with any application that is able to communicate through the Ice system. And if that isn't possible the File Monitor component can be used for visualization of data produced by the application using file-based communication.

VII. Acknowledgment

This work was supported by the EU FP7 IP "CogX: Cognitive Systems that Self-Understand and Self-Extend" (ICT–215181) and by the Research program P2-0214(RS).

References

[1] "WebGL," http://www.khronos.org/webgl.
[2] "ROS (Robot operating system)," http://www.ros.org.
[3] "Peekabot," http://www.peekabot.org.
[4] "Qt - Cross-platform application UI framework," http://qt.nokia.com.
[5] "The WebKit open source project," http://www.webkit.org.
[6] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, pp. 66–75, 2004.
[7] "ZeorC, the home of Ice," http://www.zeroc.com.
[8] "Introducing JSON," http://www.json.org.
[9] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The evolution of Lua," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007.
[10] "OpenGL binding for Lua 5.1 and 5.2," http://luagl.sourceforge.net.
[11] "CogX - cognitive systems that self-understand and self-extend," http://cogx.eu.
[12] N. Hawes and J. Wyatt, "Engineering intelligent information-processing systems with cast," *Advanced Engineering Informatics*, vol. 24, no. 1, pp. 27 – 39, 2010.
[13] D. Skočaj, M. Kristan, A. Vrečko, M. Mahnič, M. Janíček, G.-J. M. Kruijff, M. Hanheide, N. Hawes, T. Keller, M. Zillich, and K. Zhou, "A system for interactive learning in dialogue with a tutor," in *IEEE/RSJ IROS*, San Francisco, CA, USA, 2011.
[14] "Graphviz - Graph visualization software," http://www.graphviz.org.