OCOA: A Modular, Ontology Based, Autonomous Robotic Agent Architecture*

Feliciano Manzano Casas [1], Luis Amable Garcia Fernandez [2]

[1] Intelligent Control Systems research group. Universitat Jaume I
Campus Riu Sec, s/n -12071- Castellon (Spain) manzano@guest.uji.es
[2] Intelligent Control Systems research group . Universitat Jaume I
Campus Riu Sec, s/n -12071- Castellon (Spain) garcial@icc.uji.es

Abstract.Ontology based Component Oriented Architecture (OCOA) is a software architecture designed for autonomous robotic agents. It is comprised of four kinds of objects that manage and interchange information with each other on a distributed peer to peer basis. The central architectural information service in the agent is the Agent Information Manager (AIM), which is notified and notifies any capability added, updated, substracted, or failed in the agent. These capabilities are managed ontologically. The architectural knowledge base is built dynamically by the components of the agent, and all of them can be searched and found using ontology as resource and information retrieval mechanism. High level logical data processing services are performed by Common Framework objects (CFo). CFos also offer the infrastructure needed to interchange raw and ontological architectural information. The interface to physical devices is provided by Devide object Drivers (DoD). DoDs extend CFo features by incorporating device and platform dependent code wrapped in Device Input Output Drivers (DIOD). DIODs are Java Native Interface objects, which operate directly with physical devices. Therefore, OCOA uses these four kinds of objects (AIM, CFo, DoD and DIOD), giving (by replacing only DIODs) a scalable, modular, platform neutral, dynamic, ontology based agent architecture.

Keywords: distributed AI, multi-agent systems, robotic software architectures, ontologies, JADE.

* This work is partly supported by the Spanish CICYT project TAP1999-0590-c02-02

1 Introduction

The most succesful robotic software architectures developed can be classified into three categories [1]: hierarchical, deliverative and hybrid. The main feature of a hierarchical architecture [2] is to be guided to reach a high level plan by restricting low-level horizontal communications. This architecture has poor flexibility, so it is difficult to adapt to modern robots, which have to manage many sensors in reactive and reflex loops. The deliverative architecture [3] adopts the opposite approach. It comprises several modules known as behaviours which run concurrently through communication and through the environment. The design of high level goals is usually difficult to achieve. Hybrid architectures [4] are the most recent. They try to combine reactive and deliberative control. However, the connection between these two levels is generally a difficult task [1].There are several trends in the development of software that can help to the development of new and powerful robotic architectures.

From Software Engineering there is a recent approach for building software architectures that reuse off-the-shelf components. This approach is called component based architectures [5]. Also, there is a growing interest in using ontologies[1] as the main tool for the development of new and powerful knowledge based systems [9]. OCOA architecture follows these trends by integrating ontology into the core of the architecture. As far as we know, this is a new approach for designing autonomous robotic software architectures that may perform dynamic reconfiguration of robotic system software components. OCOA, writen in Java, is a hybrid robotic software architecture that uses component based features. This architecture proposes approaches to:

- Total portability of components among different robotic platforms.
- Dynamic plug/unplug of interdependent behavioral reactive, deliverative and physical-driver components, even among different physical agents, without loss of control over the agent.
- Ability to perform, structure and coordinate complex interdependent reactive and deliverative behaviors.

In order to be able to manage the dynamic adding and removing of components, the OCOA architecture uses an architectural knowledge base. This architectural knowledge base is dynamically built by the components of the OCOA architecture. All its components manages the same ontology which is stored in the AIM. This component provides the OCOA with a yellow and white pages server, thus any request for information about services available in OCOA is answered by the AIM. Logical data processing services are performed by Common Framework objects (CFo) whilst the interface to physical devices is provided by Device object Drivers (DoD). DoDs extend CFo features by incorporating device and platform dependent code wrapped in Device Input Output Drivers (DIOD). DIODs are Java Native Interface [6] objects, which operate directly with physical devices. Therefore, OCOA is a scalable (components can be added or removed), modular (component based), platform neutral (by replacing only DIOD components the rest of the OCOA

[1] In this context, an ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents [13]

architecture can be used in different robotic hardware architectures), dynamic and ontology based software agent architecture. Moreover, due to the use of Java Remote Machine Interface (Java/RMI) [7] [8], each component may be located in a different Java Virtual Machine, therefore it also has distributed characteristics.
This paper describes the OCOA architecture and it is organized as follows: section 2 sets out the robotic architecture ontology used by the OCOA architecture. Section 3 describes the components of OCOA: AIM, CFo and DoD. Section 4 show an overview of the coordination resources that OCOA provides. Section 5 show an example of a robotic agent that uses OCOA. Section 6 describes some details of OCOA implementation. Finally, the conclusions are drawn.

2 OCOA Robotic Ontology

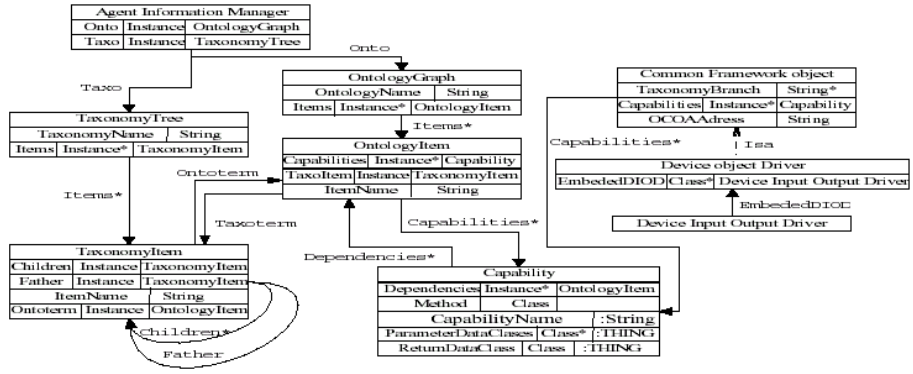

Fig. 1. OCOA architectural ontology

In this section it is set out the ontological representation of the architectural knowledge base of OCOA. The description includes the properties, features, attributes and restrictions of each concept. Figure (1) shows the class tree used for representing the architectural knowledge base.
The root of the class tree is the Agent Information Manager, which has an instance of the OntologyGraph and an instance of the TaxonomyTree.
The TaxonomyTree contains the name of the taxonomy, and a collection of TaxonomyItems. Each TaxonomyItem contains the name of the item, the Children that the item owns (which are instances of TaxonomyItem), the Father of the item itself (which is, again, an instance of TaxonomyItem), and a link to OntologyItem (which is the ontological correspondence of the TaxonomyItem in the knowledge base).
The OntologyGraph contains the name of the Ontology and a collection of OntologyItems. Each OntologyItem contains the name of the item, a link to TaxonomyItem (which is its taxonomical hierarchy correspondence), and a collection of instances of Capabilities.
Each Capability contains the CapabilityName, a collection of Dependencies (which are instances of OntologyItem), and information related to the concrete capability implementation made by the part of a concrete CFo (Method, ParameterClass and ReturnDataClass). The CFo is comprised of a collection of Capabilities, an

OCOAAddress (which univocally identifies the component in the OCOA agent), and a series of strings related to the taxonomy branch kept by this CFo in the TaxonomyTree. Device object Driver has a ”is-a” relationship with a Common Framework object. This relationship represents that a DoD is a ”kind of” CFo. Device object Driver contains an EmbebedDIOD, which links DoD to the system library that can be used to manage physical devices.

3 The structural components of OCOA Architecture

In this section the main structural components of the OCOA architecture are explained.

3.1 The Agent Information Manager (AIM)

The Agent Information Manager provides the agent with a white and yellow pages server. It manages available information about components of the agent by using the architectural knowledge base. Through the registration process, a component anounces its existence, capabilities, goals and dependencies to the AIM. Thus, the AIM incorpotes the component in its architectural knowledge base. Capabilities, goals and dependencies are specified by the component being registered using the common ontology of the architecture. The taxonomical information provided by the component can be non existent. Thus, the AIM must include this information as a new branch of the taxonomical tree.

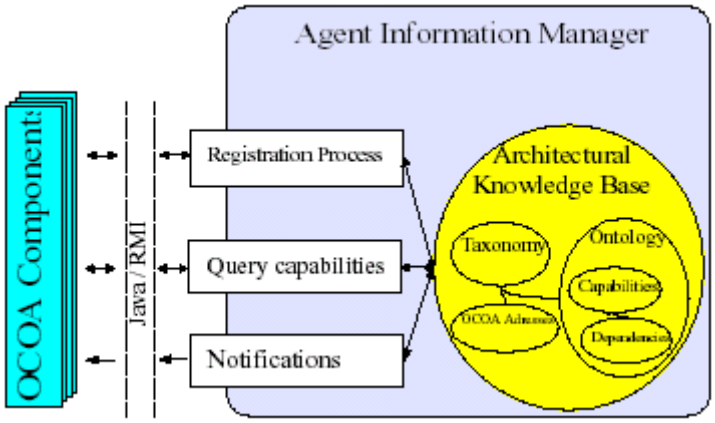

Fig. 2. Modular description of the Agent Information Manager (AIM)

As a result of this registration process, the component receives from the AIM its own OCOA address and all the addresses of its dependent components. If any of the component dependences are not available (i.e. not yet registered), the dependence OCOA addresses will be not provided. These addresses will be sent when the related components that provide these capabilities are registered in the AIM.

The modular structural description of the AIM is shown in Figure 2. It includes the architectural knowledge base (which includes taxo-ontological information and component addresses) and facilities to communicate with other components by providing methods for performing registration, notifications and requests of capability explanations.

3.2 The Common Framework Object (CFo)

Common Framework object provides facilities to interchange information with other agent components: methods to register and unregister to the AIM and methods to attach listeners and triggers to other OCOA components. Also, the CFo includes its own timers and watchdogs.
The CFo implements a *Listener Manager* which accepts and processes new listener registrations from other components and requests listener registrations to other components. This *Listener Manager* processing involves a complete ontological knowledge of the component to be registered. This knowledge is used to deal with conflicting external requests.
CFo also implements a *Dependence Manager* which manages all dependency information that this CFo has with other components in the agent.
The *Capability Manager* performs several tasks in the CFo: 1) it informs the *Listener Manager* about new registrations to be made to other components; 2) it provides all necesary information to perform the registration to the AIM; 3) it manages all communications needed by any capability method; and 4) it is informed of dependency modifications by the *Dependence Manager*.

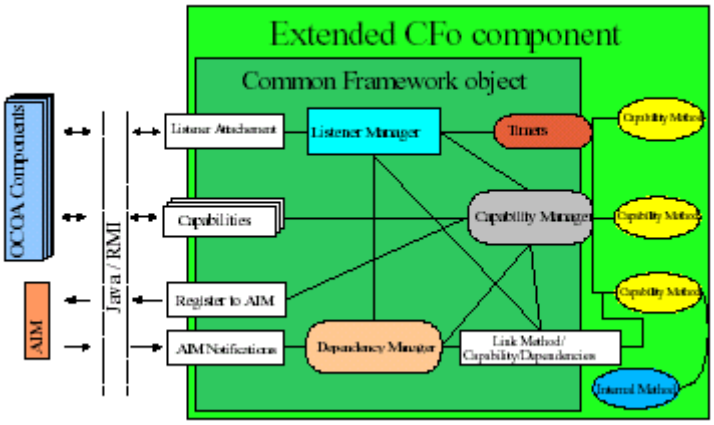

Fig. 3. Modular description of an Extended Common Framework object (ECFo).

The CFo must be extended with capability methods to perform the desired tasks and, afterwards, must be linked to an onto-taxonomical description of the capabilities that these methods perform. Figure 3 shows the modular structural description of an extended CFo component (ECFo).

When a CFo is incorporated into the architecture, the CFo communicate with the AIM to provide it with its capabilities and dependencies.
The AIM incorporates the CFo into the general ontology and, as result, the AIM sends the CFo OCOA address, and the addresses of the CFo dependence components to the CFo. With this information, the CFo determines how and when to use the dependences relating to the tasks to be executed. If any of its core dependences are not available, the CFo states inactive until the AIM notifies it of the availability of those dependences.

3.3 Device object Driver (DoD)

The Device object Driver can be shown as an abstraction layer to hide platform dependant device implementation issues. DoD, besides inheriting all the functionalities of its superclass (the CFo), adds a new object, the Device Input Output Driver (DIOD). This new object wraps a link to a plattform dependant driver (a system library program, usually written in C), which allows access to physical, plattform dependant devices. The DIOD links platform dependant code through Java Native Interface.

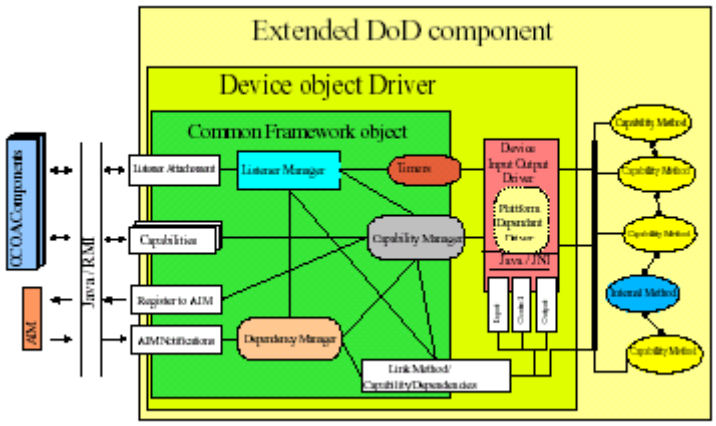

Fig. 4. Modular Description of an Extended Device object Driver (EDoD)

The possibilities offered by its superclass (the CFo) enable preprocessing of input data signals and revision of the execution of commands that interact with the external world environment. As a result, DoD can carry out explicit trapping of errors that occur within primitive action/sense tasks and the subsequent activation of an alternative or error-correction activity. These reactive control possibilities that DoD offers, allow prewired patterns of behavior. DoD may have either eager sensing (i.e. senses often to update the system's view of the world) or lazy sensing (i.e. senses by request of any other component of the agent). Both ways can be chosen. This gives OCOA agents the ability to selectively focus their attention on specific aspects of their environment. These considerations allow an OCOA agent to operate in real-time dynamic environments, due to the possibility of executing simple reaction strategies,

the lack of an explicit external world representation, and the reactive response to stimuli.
The DoD must be extended with: 1) a platform dependant driver linked to the DIOD, 2) capability methods to develop the desired tasks; and 3) link these methods to a taxo-ontological description of the capabilities. Figure 4 shows the modular structural description of an Extended DoD component (EDoD).

4 Component coordination[2]

In OCOA architecture coexists several components. All of them try to accomplish their job, and ocassionally will race to obtain necessary resources. These resources can be, i.e. complying with DoD sensors, efectors, CFos that express different levels of behavior, etc ... Coordination among them is reached by getting semanic knowledge of the tasks and goals asigned to each component. This knowledge is expressed at an ontological level, and it is stored in the AIM when the component is registered. Details of tasks to be accomplished are expressed by:

Precondition(s) to activate the behavior. These preconditions can be, i.e. a definite state of the environment.

Postcondition: State of the agent after the execution of a behavior. This can imply the interchange of messages among different components.

Execution priority: It has to be set off-line. Some behaviors will require a higher priority over remainder behaviors(obstale avoidance, panic behavior, etc...). Remainder behaviors may have an standard priority, and given a punctual situation, race for resources.

Execution deadtime: It can be an absolute or relative temporal definition, and in terms of available resources or state of the agent.

5 OCOA by example

In figure (5) can be seen an example of the use of OCOA in an autonomous robotic agent. In the lower side of the figure, we can see three physical devices managed by DIODs, which are embebed in Extended DoD (EDoD) components. Each EDoD manages a DIOD. All EDoDs are interconnected to allow data interchange needed to perform reactive behaviors. Above the EDoDs, a series of Extended CFos (ECFos) can be seen. These ECFos perform logical processing of data provided by the EDoDs; one of the ECFos uses an EDoD to deal with the movement of the robot. All ECFos perform deliverative processing of data (i.e.: map building, spatial and temporal reasoning, and navigational processing). The Agent Information Manager manages architectural knowledge data and, as a future work to be done, to perform communication with external agents.

[2] In this section, though a component can run several behaviors, the term “component” (structural), can be freely interchanged by the term “behavior” (functional).

Figure (6) shows a cronogram representing event registrations, notifications and capability requests during the execution of the OCOA implementation shown in figure (5). The first action each component performs is to register itself to the AIM.
After providing their capabilities and dependences, the AIM provides each component with its own OCOAAddress and the OCOAAddresses of their dependence components (if available).

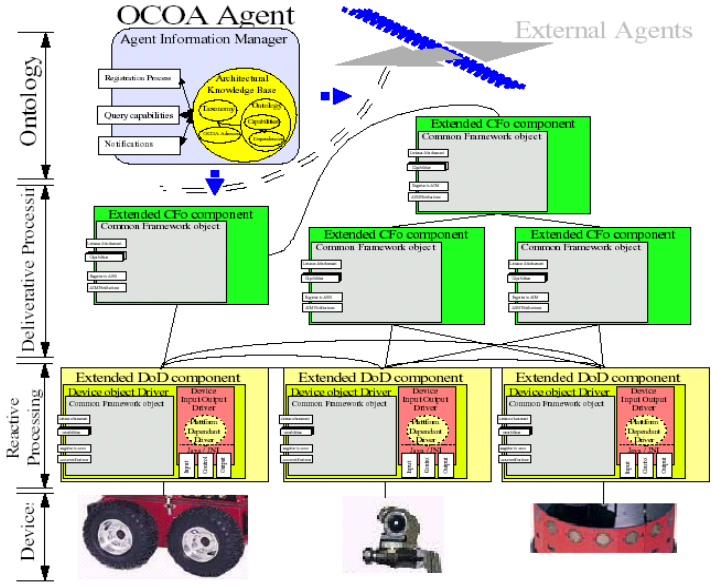

Fig. 5. Software components and hierarchical relationships in an example agent

Next, the components mutually perform a series of registration processes, in order to append listeners to achieve automatic event notifications. The last series of processes shown in figure (6) evidences the interaction between the active components in the agent.

6 OCOA implementation. Ongoing efforts.[3]

OCOA implementation is being done using JADE [14] and its framework. According with JADE philosophy, every agent runs an unique execution thread. By this reason, every OCOA component will be compound by several JADE agents. Every manager of OCOA components (Listener, Dependence and Capability managers), is being implemented using a different and separate execution thread; and every manager will be a different JADE agent. Also, every OCOA capability will be implemented by a JADE agent. All OCOA components will have common methods for initialize, register and cleanly exit from the system. As further work, an analisys and comparation with the most relevant robotic software architectures will be done.

[3] In this section, though a component can run several behaviors, the term “component” (structural), can be freely interchanged by the term “behavior” (functional).

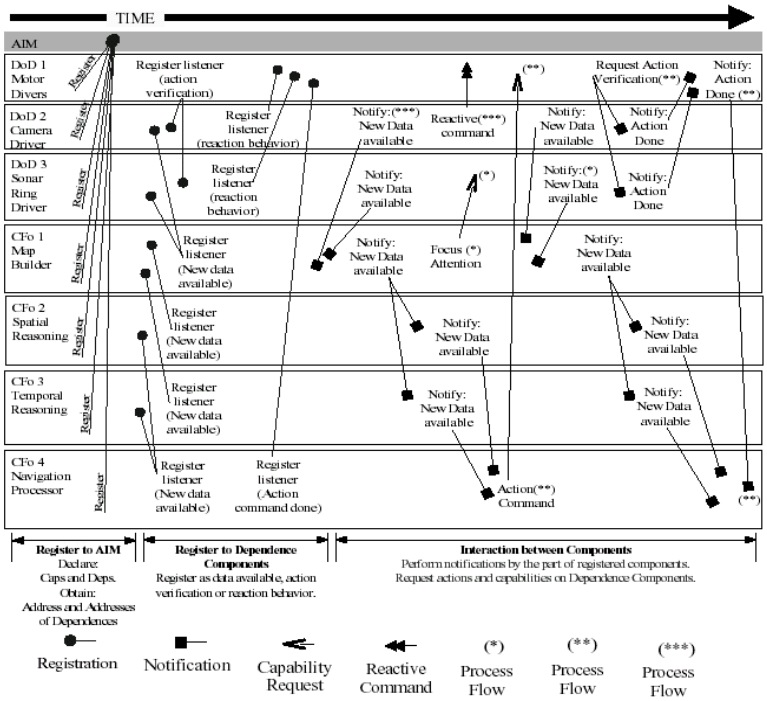

Fig. 6. Cronogram of the execution of the OCOA implementation shown in figure 5. There are three main stages: register to AIM, register to dependent components and interaction between components.

7 Conclusion

Several recently published architectures for robotic autonomous agents use component based theories (i.e. the works of [10] [11] [12]). OCOA main advantages among other architectures are:

- *Portability*: the choice of Java as the language to use in this architecture allows the implementation on a wide variety of target platforms, and OCOA is not tied to any specific operating system.
- *Reusability*: CFos perform high level logical processing. Therefore, they can be reused off-the-shelf in different robotic platforms.
- *Dynamic Component Plug-in*: through the registration process, capabilities can be added or substracted dynamically to the system.
- *Modularity*: The definition of the OCOA architecture is inherently modular.
- *Scalability*: through the Java/RMI distributed computation model, nodes can be attached to the system to add more compute power, and thanks to OCOA this is done transparently.
- *Fault Tolerance and Security*: due to the use of Java/RMI fault tolerance and security issues are provided.

- *Reactive Control Behaviour Pattern*s: DoD structure provides prewired patterns of behavior.
- *Ability to focus attention on specific aspects of the robot environment*: DoDs provide to perform lazy or eager sensing.
- *High level planning*: CFos provide the possibility of performing deliverative processing.
- *Architectural Knowledge Base*: The ontology is built dynamically with the components. Moreover, the use of ontology provides a way to perform real dynamic component plugin and resolution of all possible coordination and component dependences in the agent.
- *Ability to perform, structure and coordinate complex interdependent reactive and deliverative behaviors.*

References

1. Eve Coste-Maniere, Reid Simmons. Architecture, the Backbone of Robotic Systems. *Proceedings of the 2000 IEEE International Conference on Robotics & Automation.* San Francisco, CA, April 2000.
2. J. Albus, R. Lumia, H. McCain. Hierarchical control of intelligent machines applied to space station telerobots. *Transactions on Aerospace and Electronic Systems.* September 1988.
3. R.A. Brooks. A robust layered control system for mobile robot. *IEEE Journal of Robotics and Automation.* March 1986.
4. A. Stoytchev, R. C. Arkin. Combining Deliberation, Reactivity, and Motivation in the Context of a Behavior-Based Robot Architecture. *http://www.cc.gatech.edu/ai/robot-lab/publications.html.* 2000.
5. Jean-Guy Schneider, Oscar Nierstrasz. Components, Scripts and Glue. *Software Architectures - Advances and Applications.* Springer-Verlag 1999.
6. Sun Corporation: JNI-Java Native Interface. *http://www.javasoft.com/products/jdk/1.1/docs/guide/jni/index.html* Sun Microsystems 1999.
7. The Java Virtual Machine Specification: Release 1.1 *Sun Microsystems white paper.* Sun Microsystems 1997.
8. Sun Microsystems. User's Manual. Java Remote Method Invocation Specification, Revision 1.4, JDK 1.1. *Sun Microsystems.* Sun Microsystems 1997.
9. A. Gomez Perez, V. R. Benjamins. Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods. *Proceedings of the IJCAI-99 workshop on Ontologies and Problem-Solving Methods (KRR5).* Stockholm, Sweden, August 2, 1999.
10. R. Volpe et al. The CLARAty Architecture for Robotic Autonomy. *Proceedings of the 2001 IEEE Aerospace Conference.* Big Sky, Montana, March 2001.
11. K. Konolige et al. The Saphira Architecture: A Design for Autonomy. *Journal of Experimental and Theoretical Artificial Intelligence, 9(1): 215- 235.* 1997.
12. R. Alami et al. An Architecture for Autonomy. *International Journal of Robotics Research, 17(4).* April 1998.
12. T.R. Gruber. A Translation Approach to Portable Ontologies. *Knowledge Acquisition, 5(2):199-220.* 1993.
13. Fabio Bellifemine, Agostino Poggi, Giovanni Rimassa. JADE – A FIPA-compliant agent framework. *Proceedings of PAAM'99, pg.97-108*. London, April 1999.