LIRMM

UM II

Component based Software Architecture of Robot Controllers

R. Passama, D. Andreu, C. Dony, T. Libourel

Rapport de Recherche LIRMM n° 05059

Septembre 2005

Abstract - **The paper presents a methodology for the development of robot software controllers, based on actual software component approaches and robot control architectures. This methodology defines a process that guides developers from the analysis of a robot controller to its execution. A proposed generic software controller architecture, useful for analysis and integration, and a dedicated component-based language, focusing on modularity, reusability, scalability and upgradeability of controller architectures parts during design and implementation steps, are briefly presented.**

Keywords - **Software Components, Control Architecture, Integration, Reuse, Petri Nets with Objects.**

I. Introduction

Robots are complex systems whose complexity is continuously increasing as more and more intelligence (decisional and operational autonomies, human-machine interaction, robots cooperation, etc.) is embedded into their controllers. This complexity also depends, of course, on the mechanical portion of the robot that the controller has to deal with, ranging from simple vehicles to complex humanoid robots [15]. These two portions of a robot, its mechanical part (including its sensors and actuators) and its control logic, are intrinsically interdependent. Nevertheless, for reasons of modularity, reusability and upgradeability, the controller design should separate, as far as possible, these two aspects: the functionalities that are expected from the robot on the one hand, and, on the other, the representation of the mechanical and technological parts that implements them. This is necessary to attain platform independence, as well as to favor upgradeability according to the technological development [14]. Indeed, the software architecture (controller) should not be closely dependent on the technology (e.g. sensors suite), neither on a specific hardware implementation (computing technology); that would reduce the opportunity to take advantage of future technical advancements (Ultrasounds, laser, visual systems could be used for obstacle detection for example).

One current limitation in the development of robot software controllers is the difficulty of integrating different functionalities, potentially originating from different teams (laboratories), into a same controller, as they are often closely designed and developed for a given robot (i.e., for a given mechanical part in particular). Hence, reusability, as well as scalability and upgradeability, are aims that are currently almost impossible to achieve since both aspects of the robot (control and mechanical descriptions) are tightly merged.

Our goal is to provide a methodology [11] that rationalizes the development process of a robot software controller in order to overcome these limitations. We thus present the CoSARC (Component-based Software Architecture of Robot Controllers) development methodology based on actual component [12] and architecture descriptions [9], approaches in software engineering and control architectures in robotics. CoSARC defines a process that guides developers during analysis, design, implementation, deployment and operation of a robot controller. Its structure is based on two concepts: a generic software controller architecture, useful for analysis and integration, presented here in section II, and a component-based language, useful for design and implementation, presented in section III. This paper concludes by citing actual work on, and perspectives of, the CoSARC methodology.

II. GENERIC SOFTWARE CONTROL ARCHITECTURE

Robot control architecture is a widely studied domain. Three categories of architectures have so far emerged: hierarchical (deliberative) architectures [8], subsumption architectures [5] and mixed architectures. The proposed CoSARC generic architecture belongs to the mixed architectures category, like ORCCAD [4], CLARATy [13] and LAAS architectures [1].

The CoSARC approach aims to improve modularity, reusability and upgradeability within robot control architectures. The generic architecture of CoSARC improves these aspects in order to provide developers with abstractions that are useful for analyzing the control architecture structure, taking into account both the robot's physical part (operative part) and the robot-control part (the robot's intelligence, i.e. the part that exhibits its behavior). The central abstraction in the CoSARC generic architecture is the *Resource*. A *Resource* is a part of the robot's intelligence that is responsible for the control of a given set of independently controllable physical elements. For instance, consider a mobile manipulator robot consisting of a mechanical arm (manipulator) and a vehicle. It is possible to abstract at least two resources: the ManipulatorResource which controls the mechanical arm and the MobileResource which controls the vehicle. Depending on developer's choices or needs, a third resource can also be considered, coupling all the different physical elements of the robot, the MobileManipulatorResource. This resource is thus in charge of the control of all the degrees of freedom of the vehicle and the mechanical arm (the robot is thus considered as a whole). The breaking down of the robot's intelligence into resources mainly depends on three factors: the robot's physical elements, the functionalities that the robot must provide and the means developers have to implement those functionalities with this operative part.

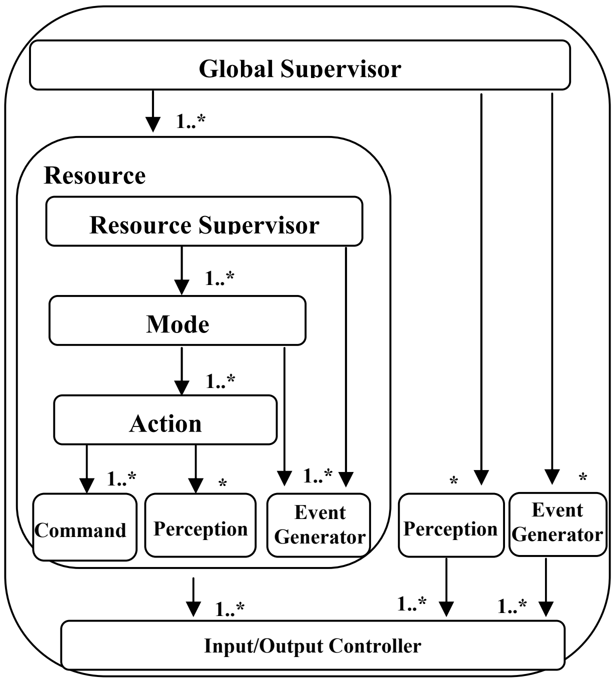

Figure 1: The CoSARC Generic Architecture

A resource (cf. Fig. 1) corresponds to a sub-architecture decomposed into a set of hierarchically organized interacting entities. Presented from bottom to top, they are:

- A set of *Commands*. A *command* is in charge of the periodical generation of command data to actuators, according to given higher-level instructions (often setup points) and sensor data; *commands* encapsulate control laws. The actuators concerned belong to the set of physical elements controlled by this *resource*. An example of a *command* of the ManipulatorResource is the JointSpacePositionCommand (based on a joint space-position control law that is not sensible to singularities, i.e., singular positions linked to the lining up of some axis of the arm).
- A set of *Perceptions*. A *perception* is responsible for the periodical transformation of sensor data into, potentially, more abstract data. An example of a *perception* of the ManipulatorResource is the ArmConfigurationPerception that generates the data representing the configuration of the mechanical arm in the task space from joint space data (by means of the direct geometrical model of the arm).
- A set of *Event Generators*. An *event generator* ensures the detection of predefined events (exteroceptive or proprioceptive phenomena) and their notification to higher-level entities. An example of an *event generator* of the ManipulatorResource is the SingularityGenerator; it is able to detect, for instance, the singularity vicinity (by means of a 'singularity model', i.e., a set of equations describing the singular configurations).
- A set of *Actions*. An *action* represents an (atomic) activity that the resource can carry out. An *action* is in charge of commutations and reconfigurations of commands. An example of an *action* of the ManipulatorResource is the ManipulatorContactSearch-Action, which uses a set of commands to which belongs the ManipulatorImpedance-Command. This command is based on an impedance control law (allowing a spring-damper like behavior).
- A set of *Modes*. Each *Mode* describes one resource behavior and defines the set of orders the resource is able to perform. For example, the MobileResource has two modes: the MobileTeleoperationMode using which the human operator can directly control the vehicle (low-level teleoperation, for which obstacle avoidance is ensured), and the MobileAutonomousMode in which the resource is able to accomplish high-level orders (e.g., 'go to position'). A *mode* is responsible for the breaking down of orders into a sequence of actions, as well as the scheduling and synchronization of these actions.
- A *Resource Supervisor* is the entity in charge of the modes commutation strategy, which depends on the current context of execution, the context being defined by the resource state, the environment state and the orders to be performed.

A control architecture consists of a set of resources (cf. Fig. 1).

The *Global Supervisor* of a robot controller is responsible for the management of resources according to orders sent by the operator, and events and data respectively produced by event generators and perceptions. Event generators and perceptions not belonging to a resource thus refer to physical elements not contained in any resource. In the given example, we use such resource-independent event generators to notify, for instance, 'low battery level' and 'loss of WiFi connection' events to some resources as well as to the global supervisor. The lower level of the hierarchical decomposition of a robot controller is composed of a set of *Input/Output controllers*. These *I/O controllers* are in charge of periodical sensor- and actuator-data updating. *Commands*, *event generators* and *perceptions* interact with *I/O controllers* in order to obtain sensor data, and commands use them to set actuator values. *I/O controllers* contribute to the abstraction of the technology which is used (set of sensors and actuators, with their specific "drivers").

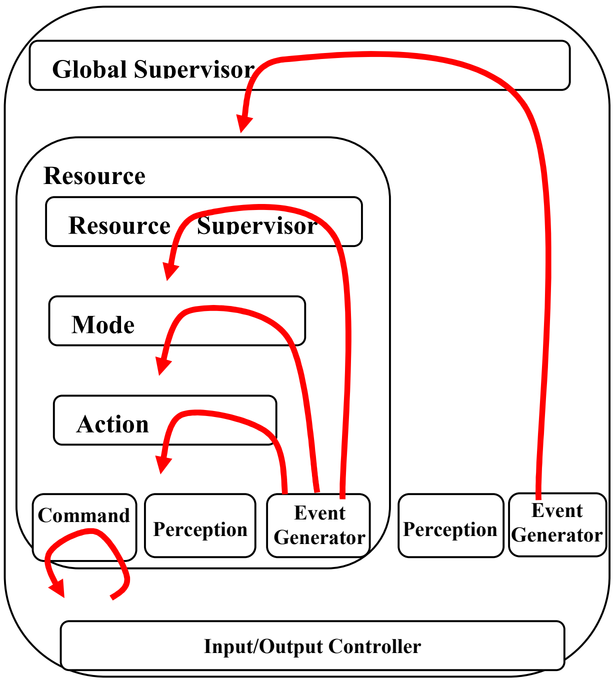

Figure 2: Reactivity loops within the CoSARC Generic Architecture

Event generators are dynamically configured, i.e. the set of events to be monitored is dynamically determined according to the context of execution (or can be permanently monitored if necessary). It allows reactivity loops to be dynamically installed according to the context, and at different levels within the controller (cf. Fig. 2). A given event can be simultaneously notified to different components, even at different levels of the hierarchy (if they "subscribed" to this event). Reactions performed further to an event occurrence depend on the behavior specified in each concerned component: command switching (e.g. for obstacle avoidance), mode commutation (e.g. from the remote control mode to the autonomous one in case of communication link break), resource change (e.g. teleoperation of the MobileManipulator-Resource instead of the MobileResource), etc.

The analysis of the controller architecture is an important stage because it allows outlining of all the entities involved in the actions/reactions of the controller (i.e. the robot behavior) and the interactions between them. To this end, we propose a dedicated design language; we will not deal with standards of data representation (position, velocity, orientation parameters, etc.) neither message formats, etc.

III. Component-based Language

A. General concepts

The CoSARC language is devoted to the design and the implementation of robot controller architectures. It proposes a set of structures to describe an architecture in terms of a composition of cooperating software components. A component is a software entity that encapsulates behavior and data, provides and requires functionalities by means of ports, and which is subject to composition. This language draws from existing software component technologies such as Fractal [6] and Architecture Description Languages such as Meta-H [3].

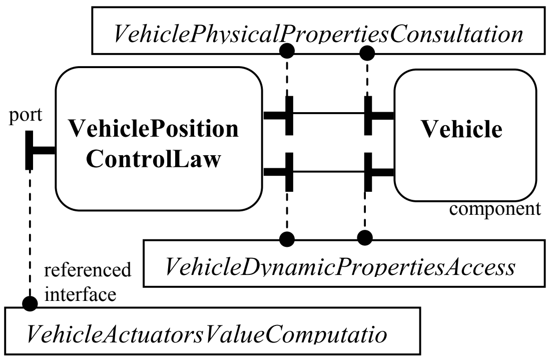
VehiclePhysicalPropertiesConsultation
port
VehiclePosition ControlLaw
Vehicle
component
referenced interface
VehicleDynamicPropertiesAccess
VehicleActuatorsValueComputatio

Figure 3: Example of two connected representation components

The main features of components in the CoSARC language are *ports*, *interfaces*, *properties* and *connections*. A component's port is a point of connection between the component and other components. A port references an interface which is a contract containing the declaration of a set of services. If a port is 'required', the component needs one or more services declared in the interface that the port references. If a port is 'provided', the component implements the services declared in the interface that the port references. Required ports must always be connected while provided ones not necessarily. The properties of components define their internal behavior (e.g., operations) and data (e.g., attributes). Their behavior implements services declared in the interfaces that are referenced by their provided ports and call some services declared in the interface that are referenced by their required ports. Connections are entities that can be used by developers to connect ports. A connection is used to connect a required port with a provided one. When a connection is established, the conformity of interfaces referenced by the provided and the required ports is checked, to ensure consistency.

In the CoSARC language, there are four types of components: *Representation Components*, *Control Components*, *Connectors* and *Configurations*. Each of them is used to deal with a specific aspect of controller architecture design. We present the specificities of these types of components in the following sub-sections.

B. Representation Components

This component type is used to describe a robot's knowledge of its environment, its mission and its physical elements. Representation components can represent abstract entities, such as events, sensor/actuator data, orders, control laws (a law in this context is a model that describes how to compute a set of outputs based on a given set of inputs), etc. They can also represent concrete entities, such as those relating to the robot's physical elements or elements of its environment.

Representation components are 'passive' entities. Their ports allow only synchronous connections, and interfaces that are referenced by their ports declare a set of synchronous services. Internally, *representation components* consist of attributes (state) and operations (behavior) that use these attributes. Operations are the implementation of the services declared in provided ports; they can use services declared in required ports. Representation components can thus be composed between themselves when they require services of each-other. Indeed, a *representation component* consists of a set of provided ports that allows other *representation components* to obtain some of its static physical properties (wheel diameter, frame width, etc.) and set or obtain its dynamical properties (velocity and orientation of wheels, etc.). Figure 3 shows a simple example of composition. The *representation component* called VehiclePositionControlLaw consists of:

- one provided port, named VehicleActuatorsValue-Computation, using which another component, a control component for instance, can ask for a computation (of the value to apply to the actuator),
- and two required ports. One of them references the VehiclePhysicalPropertiesConsultation interface, the other references the VehicleDynamicProperties interface. These interfaces are necessary for the computation as some parameters of the model depend on the vehicle on which the corresponding law is applied. The corresponding ports are provided by the *representation component* Vehicle. VehiclePositionControlLaw and Vehicle are so composed by connecting the two required ports of VehiclePositionControlLaw with the two corresponding provided ports of Vehicle.

Representation components are used by components of other types, such as control components and connectors.

C. Control Components

A *Control Component* is used to describe and program entities in charge of the robot control activities. These entities, such as Event Generators, Commands, Actions, etc., are described in the CoSARC generic architecture (cf. section 2, Fig. 1).

The behavior of a *control component* determines the decision/reaction of the robot, according to its knowledge and the effective context. So each *control component* encapsulates a set of representation components that represent this knowledge. These representation components can be formal parameters of its services or of its attributes.

Control components are 'active' entities. Their ports permit asynchronous communications, and interfaces referenced by their ports declare a set of asynchronous services. Internal properties of a *control component* are attributes, operations and a Petri net with objects that describes its reactive asynchronous behavior [10].

The asynchronous behavior of a *control component* describes the way its operations are executed (synchronizations, parallelism, concurrent access to its attributes, etc.). Tokens inside the Petri net refer to representation components (the knowledge used by the *control component*), and the structure of the Petri net describes temporal and logical control flows applied from/to this knowledge (when computing its reaction). The reactive behavior also describes the way each *control component* synchronizes its internal activities with activities of others *control components*.

The use of Petri nets with objects is justified by the need of formalism to describe precisely not only synchronizations, concurrent access to data and parallelism within *control components*, but also interactions between them. Petri nets' formal analysis capabilities, which have been widely studied [7], provide developers with a way of verifying the controller model (its logical part). Moreover, Petri nets with objects can be directly executed by means of a token player.

Figure 4 presents an example of a *control component* that represents a command entity (cf. Fig. 1), named MobilePositionCommand. It has three attributes: its periodicity, the Vehicle being controlled and the applied VehiclePositionControlLaw. The Vehicle and the VehiclePositionControlLaw are connected in the same way as described in Figure 3, meaning that the MobilePositionCommand will apply the VehiclePositionControlLaw to the Vehicle at a given periodicity. Such a decomposition allows the adaptation of the MobilePositionCommand to the Vehicle and the VehiclePositionControlLaw used. It is thus possible to reuse this *control component* in different controller architectures (for vehicles of the same type). This control component's provided port references an interface named MobilePositionControl that declares services offered to other *control components* in order to be activated/deactivated/configured. Its required ports reference one interface each: MobileMotorsActuatorsAccess which declares services used to fix the value of the vehicle's motors and MobileWheelVelocityandOrientationAccess which declares services used to obtain the values of the orientation and velocity of the vehicle's wheels. These two interfaces are implemented by one or two I/O controllers (cf. Fig.1), depending on the developer's choices and/or hardware architecture constraints.

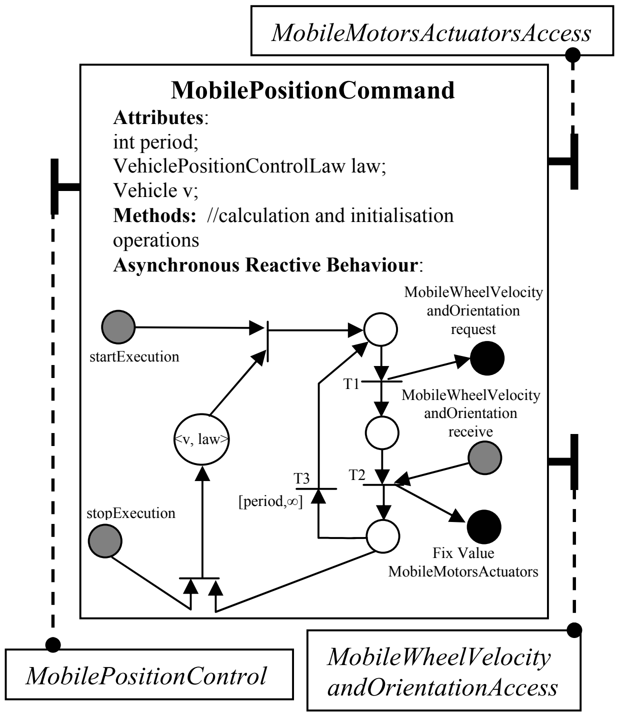
MobileMotorsActuatorsAccess
MobilePositionCommand
Attributes:
int period;
VehiclePositionControlLaw law;
Vehicle v;
Methods: //calculation and initialisation operations
Asynchronous Reactive Behaviour:
MobileWheelVelocity andOrientation request
startExecution
T1
MobileWheelVelocity andOrientation receive
<v, law>
T3
T2
[period,∞]
stopExecution
Fix Value MobileMotorsActuators
MobilePositionControl
MobileWheelVelocity andOrientationAccess

Figure 4: Simple example of a control component

The (simplified) Petri net with objects representing the asynchronous reactive behavior of MobilePositionCommand is shown in Figure 4. It describes the periodic control loop performed by this control component. This loop is composed of three steps: the first one (firing of transition T1) consists of requests for sensors data, the second one (firing of transition T2) consists of the computation of the reaction by executing VehicleActuatorsValueComputation services (cf. Fig. 3) and then by fixing the values of the vehicle motors, and the third one (firing of transition T3) consists of a wait for the next period before a new iteration (loop). Grey and black Petri net places represent, respectively, reception and transmission of messages corresponding to service calls. For example, places startExecution and stopExecution correspond to a service declared in the MobilePositionControl interface whereas places MobileWheelVelocityandOrientationRequest and MobileWheel-VelocityandOrientationReceive correspond to a service declared in the MobileWheelVelocityandOrientationAccess interface.

D. Connectors

Interactions between composed control components, involving a large number of synchronizations and constraints, are modeled in the CoSARC language by protocols carried by components named *connectors*. Such interaction protocols then become reusable entities, as in many component-based approaches like [2]. Moreover, developers need not mix interactions and reactive behavior – which relate to different aspects – within control components.

Connectors are also components, used to connect ports of control components. A *connector* has at least two ports to connect at least two control components; each port defines the role of the control component within the given interaction. A role is specified by an interface that describes the contract that must be respected by the control component to be connected (to the port of this role). Like control components, a *connector* has several attributes and operations, and a Petri net with objects describing the asynchronous interaction it is responsible for.

An example of a (simple) *connector* used to connect two control components is shown in Figure 5. This *connector*, named Request/ReplyConnector, describes a simple interaction protocol between a Requester and a Replier. It consists of two ports: one provided port referencing the Requester interface and one required port referencing the Replier interface. The control component assuming the Requester role sends a request message to the control component assuming the Replier role, which then sends the reply message to the Requester. Constraints described in the Petri net with objects ensure (for example) that only one request will be sent by the Requester until it receives a reply, and that the Replier will process only one request until it sends the reply to the Requester. This *connector* can be used to establish connections between different control components, and each time the interaction to be described corresponds to this protocol.

Just like control or representation components, *connectors* are not only modeling entities but also programming ones, i.e., entities that exist at runtime, ensuring communications between control components.

Connectors, being also modeled by Petri nets, allow the building of the global Petri net, i.e., one resulting from the composition of control components. Thanks to this property, developers can analyze inter-component synchronizations, allowing then to check, for example, that the interconnection does not introduce any dead-lock.

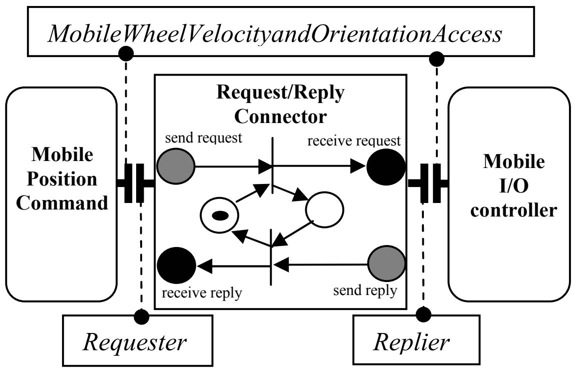

Figure 5: Simple example of connector

E. *Configurations*

When a resource has been completely modeled, the result is a graph of the composition of control components by means of connectors (i.e., a software architecture). The CoSARC language provides another type of component, named *Configuration*, that contains this graph. It allows developers to encapsulate a software (sub-)architecture into a reusable entity. A *configuration*, at the design phase, can be considered (and so composed) as a control component.

Like all components, they have provided and required ports. These ports allow asynchronous communications and reference interfaces that declare asynchronous services. Ports of a *configuration* export (dotted lines) ports of control components that the *configuration* contains. At runtime, any connection to those ports is directly replaced by a connection to the initial port, i.e. to that of the concerned control component. Figure 6 shows an example of a configuration: the MobileResource, corresponding to a sub-architecture. This *configuration* exports the provided port of the MobileSupervisor and the required ports of MobilePositionCommand and MobileObstacleEventGenerator.

A *configuration* results from interconnections of control components, according to the pattern provided in the CoSARC generic architecture. For instance, the MobileResource is constituted by the MobileSupervisor, the MobileAutonomousMode, the MobileActionMoveToPosition, which interacts with the MobilePositionCommand, and the MobileObstacleEventGenerator.

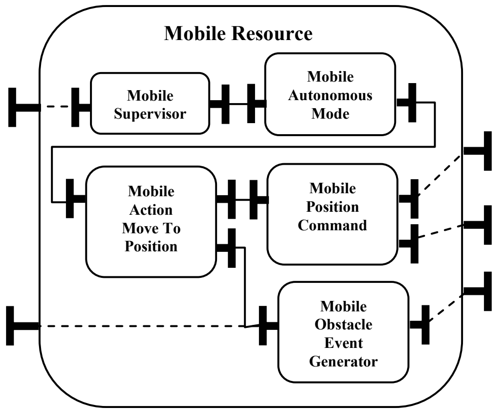

Figure 6: Simple example of a configuration

Since a *configuration* can contain others *configurations*, this structure allows developers to hierarchically model the controller architecture. When an architecture is built following the pattern provided by the CoSARC generic architecture, the 'highest' *configuration* is the *Robot*

Controller. In the given example, the MobileManipulatorController *configuration* contains as many *configurations* as resources, i.e., the ManipulatorResource and the MobileResource.

The CoSARC language also provides structures to describe the hardware part of a controller (graphs of nodes, for instance), the different processes (containers) executing one or more control components and the scheduling of these processes on each node (its system manager). At the deployment stage, *configurations* are used to install/uninstall components on containers and to set the parameters of system managers.

IV. Conclusion

We have briefly presented the CoSARC methodology, which is devoted to improving modularity, reusability and the upgradeability of control architectures. It is specifically dedicated to the integration of different aspects concerning robot control (control laws, physical descriptions, action scheduling, etc.), and can be seen as a framework into which any standard can be used by developers to represent their data, messages, services, etc. Moreover, the CoSARC language has the added benefit of relying on a formal approach based on Petri nets with objects formalism. This allows analysis to be performed at the design stage itself. After all, analysis cannot be ignored when designing the control of complex systems.

We are currently implementing the example of the *Mobile Manipulator Robot Controller* architecture. Future work will concern the CoSARC language execution engine and the CoSARC software engineering environment.

References

[1] Alami, R. & Chatila, R. & Fleury, S. & Ghallab, M. & Ingrand, F. (1998). An architecture for autonomy, International Journal of Robotics Research, vol. 17, no. 4 (April 1998), p.315-337.

[2] Aldrich, J. & Sazawal, V. & Chambers, C. & Notkin, D. (2003). Language support for connector abstraction, In Proceedings of ECOOP'2003, pp.74-102, Darmstadt, Germany, July 2003.

[3] Binns, P. & Engelhart, M. & Jackson, M. & Vestal, S. (1996). Domain Specific Architectures for Guidance, Navigation and Control, International Journal of Software Engineering and Knowledge Engineering, vol. 6, no. 2 (June 1996), pp.201-227, World Scientific Publishing Company.

[4] Borrely J.J. et al. (1998). The Orccad Architecture, International Journal of Robotics Research, Special issues on Integrated Architectures for Robot Control and Porgramming, vol. 17, no. 4 (April 1998), pp.338-359.

[5] Brooks, R. et al. (1998). Alternative Essences of Intelligence, in Proceedings of American Association of Artificial Intelligence (AAAI), pp. 89-97, July 1998, Madison, Wisconsin, USA.

[6] Bruneton, E. & Coupaye, T. & Stefani, J.B. (2002). Recursive and Dynamic Software Composition with Sharing, In Proceedings of the 7th International Workshop on Component-Oriented Programming (WCOP02) at ECOOP 2002, June 2002, Malaga, Spain.

[7] David, R. & Alla, H. (2004). Discrete, Continuous and Hybrid Petri Nets, Ed. Springer, ISBN 3-540-22480-7, 2004.

[8] Gat, E. (1997). On three-layer Architectures, A.I. and mobile robots, D. Korten Kamp et al. Eds. MIT/AAAI Press, RR. N°3552, 1997.

[9] Medvidovic, N. & Taylor, R.N. (1997). A framework for Classifying and Comparing Software Architecture Description Languages, in Proceedings of the 6th European Software Engineering Conference together with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Springer-Verlag, pp. 60-76, 1997, Zurich, Switzerland.

[10] Sibertin-Blanc, C. (1985). High-level Petri Nets with Data Structure, in proceedings of the 6th European workshop on Application and Theory of Petri Nets, pp.141-170, Espoo, Finland, June 1985.

[11] Stewart, D. B. (1996). The Chimera Methodology: Designing Dynamically Reconfigurable and Reusable Real-Time Software Using Port-Based Objects, International Journal of Software Engineering and Knowledge Engineering, vol. 6, no. 2, pp.249-277, June 1996.

[12] Szyperski, C. (1999). Component Software: Beyond Object Oriented Programming, Addison-Wesley publishing.

[13] Volpe, R. et al. (2001). The CLARATy Architecture for Robotic Autonomy, in Proceedings of the IEEE Aerospace Conference (IAC-2001), vol. 1, pp.121-132, Big Sky, Montana, USA, March 2001.

[14] Joint Architecture for Unmanned Systems. http://www.jauswg.org/

[15] Climbing and Walking Robots (CLAWAR) Network. http://www.clawar.com/home.htm