Overview of a new Robot Controller Development Methodology

R. Passama[1,2], D. Andreu[1], C. Dony[2], T. Libourel[2]
[1]Robotics Department
[2]Computer sciences Department
LIRMM, 161 rue Ada 34392 Montpellier, France
E-mail : {passama, andreu, dony, libourel}@lirmm.fr

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 control architecture pattern and a dedicated component-based language, focusing on modularity, reusability, scalability and upgradeability of controller architectures parts during design and implementation steps, are presented. Finally, language implementation issues are shown.*

Keywords: *Software Components, Control Architecture, Integration, Reuse, Object Petri Nets.*

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. Robot controllers development platforms and their underlying methodologies are of great importance for laboratories and IT societies, because there is an increasing interest in future service robotics. Such platforms help developers in many of their activities (modelling, programming, model analysis, test and simulation) and should take into account preoccupations like the reuse of software pieces and the modularity of control architectures as they correspond to two major issues.

The goal of our team is to provide a robot controller development methodology and its dedicated tools, in order to help developers overcoming problems during all steps of the design process. So, we investigate on the creation of a software paradigm that specifically deals with controller development preoccupations. From the study of robot control architectures presented in the literature, we identified four main different practices in control architecture design approaches that must be considered.

The first practice is the structuring of the control activities. There are different approaches. One approach consists in decomposing the control architecture into hierarchical layers, like in LAAS architecture [ALA, 98], 4D/RDC [ALB, 02], ORCCAD [BOR, 98] and some others. Each layer within the robot controller has a "decision-making system", as each layer only ensures part of the control (from low level control to planning). Such a decomposition impacts on the reactivity of the robot controller. The lower the layer is, the higher is the time constraint of its execution. The upper the layer is, the higher is the priority of its reaction. This hierarchical approach has been extended to hybrid architectures. For instance, AURA [ARK, 97] proposes to mix it with a behavioural approach in order to improve reactivity. In doing so, the interaction scheme is not limited to interactions between adjacent layers (for reactivity purposes): some data can be simultaneously available to several layers, or an event, can be directly notified to the upper layers (without passing through intermediary ones), etc. In behavioural approaches, like the subsumption architecture [BRO, 86] or similar ones (AURA for example), interactions between basic behaviours are complex, even if those interactions are not explicit and that they are taken into account by means of an "external" entity (for instance, the entity that computes the weighting of commands generated by individual low level behaviours).

The second practice is the decomposition of the control architecture into sub-systems that incorporate the control of specific parts of a robotic system. This practice is reified in IDEA agents architectures [MUS, 02] and Chimera development methodology [STE, 96]. This organizational view is orthogonal to the hierarchical one: each sub-system can incorporate both reactive and 'long term' decision-making activities and so can be "layered" itself.

The third practice is to separate, in the architecture description, the "robot operative portion" description from the "control and decision-making" one. This practice is often adopted at implementation phase, except in specific architectures like CLARATY [VOL,01], in which the "real world" description is made by means of objects hierarchies. These two portions of a robot, its mechanical portion (including its sensors and actuators) and its control one, are intrinsically interdependent. Nevertheless, for reasons of reusability and upgradeability, the controller design should separate, as far as possible, two aspects: the functionalities that are expected from the robot on the one hand, and, on the other, both the representation of the mechanical part that implements them and that of the environment with which it interacts. 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). Hence, upgradeability and reusability are aims that are currently almost impossible to achieve since both aspects of the robot (control and mechanical descriptions) are tightly merged. The reuse of decision-making/control systems parts is also a big challenge, because of the different approaches (behavioural or hierarchical) that can be used to design it.

Finally, the fourth practice is to use notations to describe the controller's parts and to formalize their interactions. Model-based specifications are coupled with formal analysis techniques in order to follow a "quality-oriented" design process. The verification of properties like invariants or the research of "dead-lock free" interactions are examples of benefits of such a process.

A robotic development methodology and its platform should propose a way to develop a control architecture using all these practices, as they correspond to complementary preoccupations. We identified five different preoccupations: description of the real world, description of the control (in the following this term will include decision-making, action, perception, etc.), description of interactions, description of the layers of the hierarchy, description of subsystems. The software component paradigm [SZY,99] helps dealing with these preoccupations in many ways (separation of protocols description from computation description, deployment management, etc.). Component based approaches propose techniques to support easy reuse and integration and they sometimes rely on formal languages to describe complex behaviour and interactions (aiming at improving quality of the design), like architecture description languages [MED, 97] for example.

In the following sections, we present the CoSARC (Component-based Software Architecture of Robot Controllers) development methodology, based on actual component models. It defines a process that guides developers during analysis, design, implementation and deployment phases. It is based on two concepts: a control architecture pattern for analysis, presented in section 2, and a component-based language, presented in section 3. It integrates robot controller preoccupation management and takes into account actual practices by promoting the use of Objects Petri Nets. The component execution and deployment model is shown in section 4. This paper concludes by citing actual work on, and perspectives of, the CoSARC methodology.

II. CONTROL ARCHITECTURE PATTERN

The CoSARC methodology provides a generic view on robot control architecture design by means of an architecture pattern. The proposed pattern is adaptable to a large set of hybrid architectures. It provides a conceptual framework to the developers, useful for controller analysis. The analysis phase is an important stage because it allows outlining of all the entities involved in the actions/reactions of the controller (i.e. the robot behaviour) and the interactions between them. It is made by following concepts and organization described in the pattern. It takes into account robot controller description depending on robot's physical portion (operative portion), to make the analysis more intuitive. The pattern also deals with design subjects, by defining the properties of layers of the hierarchy and the matching between layers and entities.

The central abstraction in the architecture pattern is the *Resource*. A r*esource* 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 Mobile-ManipulatorResource. 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.

A *resource* (cf. Fig. 1) corresponds to a sub-architecture decomposed into a set of hierarchically organised 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. C*ommands* encapsulate control laws. The actuators which are 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 action of the ManipulatorResource is the ManipulatorContactSearchAction, which uses a set of commands to which belongs the ManipulatorImpedanceCommand. This *command* is based on an impedance control law (allowing a spring-damper like behaviour). In a more "behavioural oriented" design an *action* could activate and deactivate sets of *commands* and manage the summing and the weighting of the command data they send to *I/O controllers*.

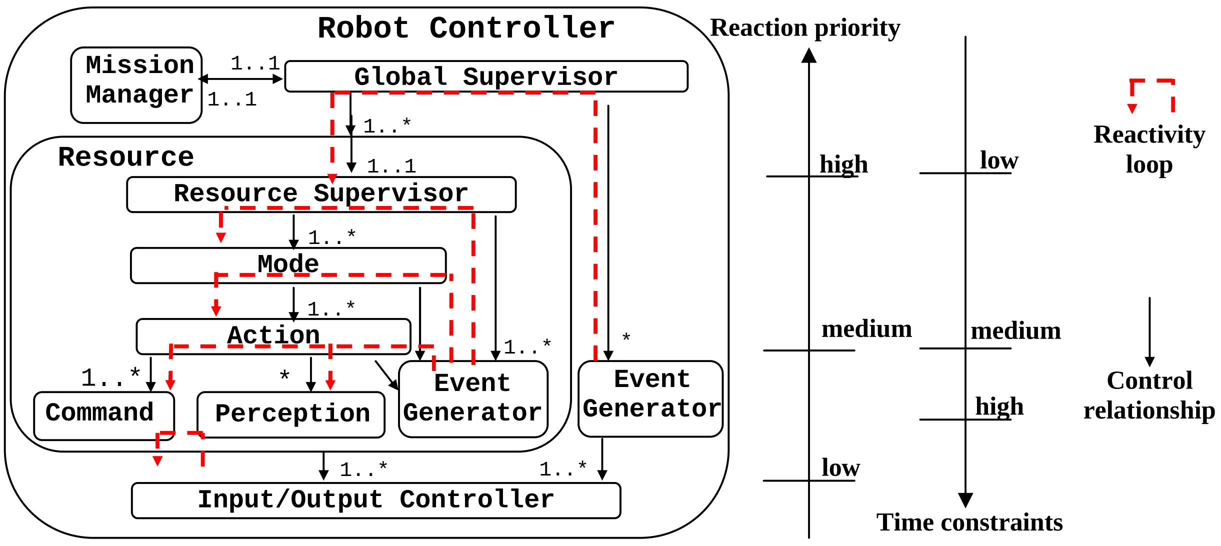

Figure 1: Control architecture pattern (UML-like syntax), and properties of the layers.

- A set of *Modes*. Each *Mode* describes the behaviour of a resource and defines the set of *orders* the resource is able to perform in the given mode. 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 corresponding operative portion state, the environment state and the orders to be performed. A robot control architecture consists of a set of resources (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 lowest 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. Other upper layer entities, like *actions* for instance, can directly interact with *I/O controllers* to configure their activities (if necessary).

Organization inside *resources* and *robot controller* follow a hierarchical approach. Each layer represents a ”level of control and decision” in the controller activities. The upper layer incorporates entities embedding complex decision-making mechanisms like *modes*, *supervisors* and *mission managers*. The intermediate layer incorporates entities like control schemas (*commands*), observers modules (*event generators*, *perceptions*) and reflex adaptation activities (inside *actions*). The lowest layer (*I/O controllers*) interfaces upper layers with sensor, actuators and external communication peripherals, and helps standardizing data exchanges. The semantic of layers hierarchy is based on the “control” relationship: a given layer controls the activities of lower layers. Two design properties emerge from this hierarchical organization. The first one is that upper layers must have a higher priority of reaction than lower layers, because their decision is more important for the system at a global scope. The second one is that lower layers have greater temporal constraints to respect, because they contain reflex and periodic activities. Managing these properties together is very important for the “real-time” aspect of the control architecture and has to be considered in our proposition.

III. COMPONENT-BASED LANGUAGE

A. General concepts

The CoSARC language is devoted to the design and implementation of robot controller architectures. This language draws from existing software component technologies such as Fractal [BRU, 02] or CCM [OMG, 01] and Architecture Description Languages such as Meta-H [BIN, 96] or ArchJava [ALD, 03]. It proposes a set of structures to describe the architecture in terms of a composition of cooperating software components. A software component is a reusable entity subject to “late composition”: the assembly of components is not defined at ‘component development time’ but at ‘architecture description time’.

The main features of components in the CoSARC language are *internal properties*, *ports*, *interfaces*, and *connections*. A component encapsulates internal properties (such as operations and data) that define the component implementation. A component’s *port* is a point of connection with other components. A port is typed by an interface, which is a contract containing the declaration of a set of services. If a port is ‘required’, the component uses one or more services declared in the interface typing the port. If a port is ‘provided’, the component offers the services declared in the interface typing the port. All required ports must always be connected whereas it is unnecessary for provided ones. The internal properties of a component implement services and service calls, all being defined in the interfaces typing each port of a component. Connections are explicit architecture description entities, used to connect ports. A connection is used to connect ‘required’ *ports* with ‘provided’ ones. When a connection is established, the compatibility of interfaces is checked, to ensure ports connection consistency.

Components composition mechanism (by means of *connections* between their *ports*) supports the “late composition” paradigm. The step when using a component-based language is to separate the definition of components from software architecture description (i.e. their composition). Components are independently defined/programmed and are made available in a ‘shelf of components’. According to the software architecture to be described, components are used and composed (i.e. their ports are connected by means of connections). The advantages of such a composition paradigm is to improve the reusability of components (because they are more independent from each other than objects), and the modularity of architectures (possibility to change components and/or connections). Obviously, the reuse of components is influenced by the standardization of interfaces typing their ports (which define the compatibility and so, the composability of components), but this is out of the scope of this paper.

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 preoccupation of controller architecture design and implementation. We present the specificities of these types in the following sub-sections.

B. Representation Components

This type of component is used to describe a robot's "knowledge" as regards on its operative part, its mission and its environment. Representation components are used to satisfy the "real-world modelling" preoccupation, but their use can be extended to whatever developers considers as the knowledge of the robot. They can represent concrete entities, such as those relating to the robot's physical elements (e.g. chassis, and wheels of a vehicle) or elements of its environment. They can also represent abstract entities, such as events, sensor/actuator data, mission orders, control or perception computational models (in this context, those models are mathematical models that describe how to compute a set of outputs based on a given set of inputs, like for instance control laws and observers), etc. When a developer wants to represent the fact that a specific model is applied on a specific (operative) part of the robot, it just has to connect those two representation components: that corresponding to the computational model with that related to the operative part. For example, Fig. 2 illustrates how to apply a control law to a given vehicle.

Representation components are 'passive' entities that only act when one of their provided services is called. They only interact according to a synchronous communication model. Internally, representation components consist of object-like attributes and operations. Operations implement the services declared in provided ports and they use services declared in interfaces of required ports. *Representation components* are incorporated and/or exchanged by components of other types, such as control components and connectors. Representation components can also 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 get the value of its "static" physical properties (wheel diameter, frame width, etc.) and/or to set/get the current value of its "dynamic" properties (velocity and orientation of wheels, etc.).

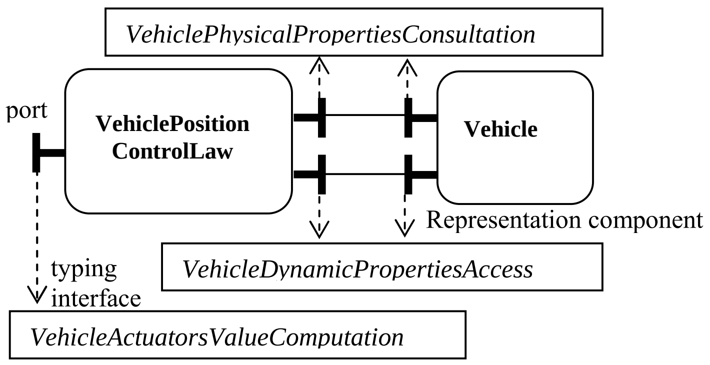
VehiclePhysicalPropertiesConsultation
port
VehiclePosition ControlLaw
Vehicle
Representation component
typing interface
VehicleDynamicPropertiesAccess
VehicleActuatorsValueComputation

Figure 2: Example of two connected representation components

Fig. 2 shows a simple example of composition. The representation component called VehiclePositionControlLaw consists of:

- a provided port, typed by the VehicleActuatorsValueComputation interface, through which another component (a representation or a control component) can ask for a computation of the value to be applied to the actuator.
- and two required ports. The first one is typed by the VehiclePhysicalPropertiesConsultation interface, the second one by 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.

C. Control Components

A *Control Component* describes a part of the control activities of a robot controller. It can represent several entities of the controller, as we decompose the controller into a set of interconnected entities (all being components), like for example: Command (i.e. entity that executes a control law or a control sequence), Perception (i.e. entity in charge of sensor signal analysis, estimation, etc.), Event Generator (i.e. entity that monitors event occurrences), Mode Supervisor (i.e. entity that pilots the use of a physical resource in a given mode as teleoperation, autonomous, cooperation), Mission Manager (i.e. entity that manages the execution of a given mission), etc. A control component incorporates and manages a set of representation components which define the knowledge it uses to determine the contextual state and to make its decisions.

Control components are 'active' entities. They can have one or more (potentially parallel) activities, and they can send messages to other control components (the communication being further detailed). Internal properties of a control component are attributes, operations and an asynchronous behaviour. Representation components are incorporated as attributes (representing the knowledge used by the component) and as formal parameters of its operations. Each operation of a control component represents a context change during its execution. The asynchronous behaviour of the control component is described by an Object Petri Net (OPN) [SIB, 85], that models its 'control logic' (i.e. the event-based control-flow). Tokens inside the OPN refer to representation components used by the control component. The OPN structure describes the logical and temporal way the operations of a control component are managed (synchronizations, parallelism, concurrent access to its attributes, etc.). Operations of the control component are executed when firing OPN transitions. This OPN based behaviour also describes the exchanges (message reception and emission) performed by the control component, as well as the way it synchronizes its internal activities according to these messages. Thus the OPN corresponds to the reaction of the control component according to the context evolution (received message, occurring events, etc.).

We chose OPN both for modelling and implementation purposes. The use of Petri nets with objects is justified by the need of formalism to describe precisely synchronizations, concurrent access to data and parallelism (unlike finite state machines) within control components, but also interactions between them. The use of Petri nets is common, for specification and analysis purposes, in the automation and robotic communities. Petri nets formal analysis has been widely studied, and provides algorithms [DAV, 04] for verifying the controller event-based model (its logical part). Moreover, Petri nets with objects can be executed by means of a token player, which extends its use to programming purposes (cf. section 4).

Fig. 3 shows a simplified example of a control component behaviour that corresponds to a command entity, named VehiclePositionCommand. 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 Fig.3, meaning that the VehiclePositionCommand will apply the VehiclePositionControlLaw to the Vehicle at a given periodicity. Such decomposition allows the adaptation of the control component VehiclePositionCommand to the Vehicle and VehiclePositionControlLaw used (i.e. the representation components it incorporates). It is thus possible to reuse this control component in different control architectures (for vehicles of the same type).

This control component's provided port (Fig. 3) is typed by the interface named VehiclePositionControl that declares services offered (to other control components) in order to be activated/deactivated/configured. Its required ports are typed by one interface each: VehicleMotorsAccess 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 provided by ports of one or more other control components (depending on the decomposition of the control architecture).

The (simplified) OPN representing the asynchronous behaviour of VehiclePositionCommand shown in Fig. 3, describes the periodic control loop it performs. This loop is composed of three steps:
- the first one (firing of transition `T1`) consists in requesting sensors data,
- the second one (firing of transition `T2`) consists in computing the reaction by executing MotorData computeVehicleMotorControl(Velocity,Orientation) operation (cf. Fig. 1) and then by fixing the values of the vehicle motors (token put in FixMotorValue black place),
- and the third one (firing of transition `T3`) consists in waiting for the next period before a new iteration (loop). Grey and black Petri net places both represent, respectively, the reception and transmission of messages corresponding to service calls. For example, grey places startExecution and stopExecution correspond to a service declared in the VehiclePositionControl interface, whereas the black place RequestVelAndOrient and the grey place ReceiveVelAndOrient correspond to a service declared in the VehicleWheelVelocityandOrientationAccess interface.

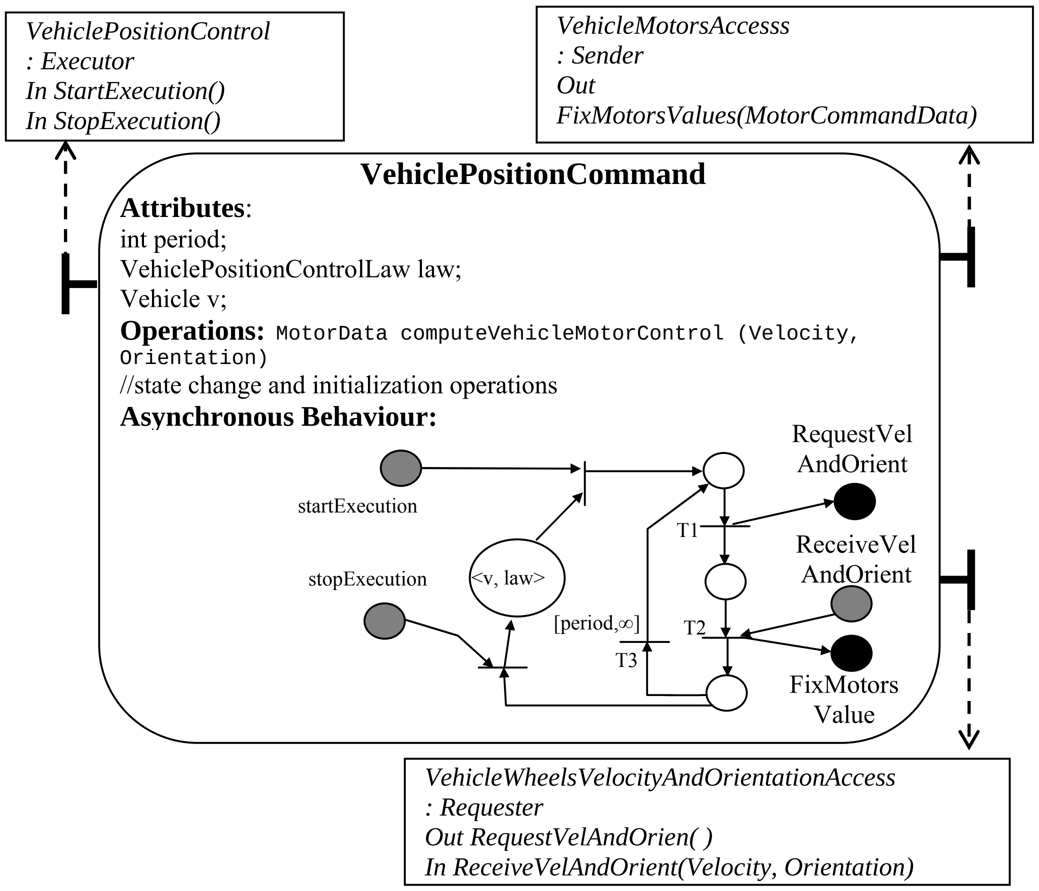

Figure 3: Simple example of a control component

D. Connectors

Connections of control components are reified into components named *connectors* (that allow the assembly). Connectors contain the protocol according to which connected control components interact. Being a component, a connector is an entity definable and reusable by a user. It implements a protocol that potentially involves a large number of message exchanges, synchronizations and constraints. Once defined, connectors can be reused for different connections into the control architecture. This separation of the interaction aspect from the control one appears to be very important in order to create generic protocols adapted to domain specific architectures. One good practical aspect of this separation is that it leads to distinguish interactions description with control activities description, whereas describing both aspects inside the same entity type would reduce the reusability.

A connector incorporates sub-components named *roles* (as attributes). Each role defines the behaviour's part that a control component adopts when interacting through the protocol defined by the connector. We then say that a control component "plays" or "assumes" a role. For example, the connector of Fig. 4 describes a simple interaction between a RequesterRole and a ReplierRole. 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 (once the reply has been computed). For each role it incorporates, a connector associates one of its required or provided ports. A connector's port is typed by an interface that defines the message exchanges allowed between the connector on one side and the control component to be connected on the other side. Fig. 4 shows that the connector has one provided port (left) typed by the Requester interface and one required port (right) typed by the Replier interface. The Replier interface defines the message exchanges between the connector and the VehicleIOController control component. VehicleIOController receives a request from the connector, computes it internally, and then sends the reply. The connection between the control components and the connector has been possible because of the compatibility of ports: an interface typing a connector's port (provided or required) must be referenced by the interface of the control component's port to which it is connected. Fig. 2 shows that VehicleWheelsVelocityAndOrientationAccess interface references the Requester interface which allows the connection of VehiclePositionCommand's port; VelocityAndOrientationAccess interface references the Replier interface which allows the connection of VehicleIOController's port (cf. Fig. 4). Finally, compatibility of control components ports is verified according to interface names. Fig. 4 shows that the connection has been possible because VehicleWheelsVelocityAndOrientationAccess service is required and provided by the two control components ports connected (i.e. each interface has the same name).

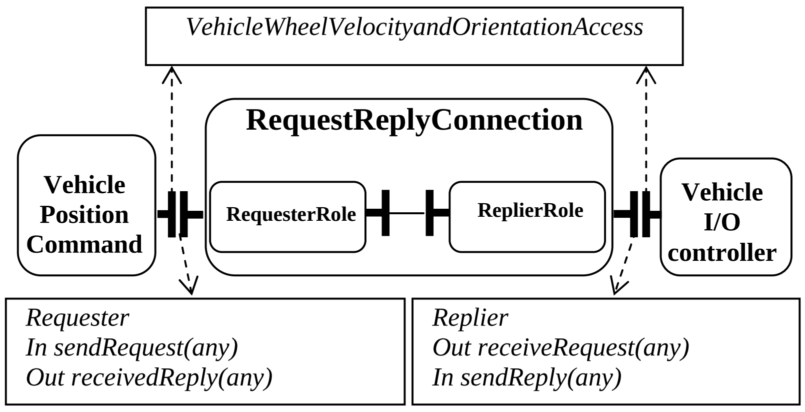

Figure 4: Simple connector example, connecting two control components

A connector can be a very adaptive entity. First, the number of roles played by components can be parameterized. Connector's initialisation operation is useful to manage the number of roles played, according to the number of control components ports to be connected by the connector and according to their interfaces. A cardinality is associated with each role to define constraints on the number of role instances. For example, the ReplierRole has to be instantiated exactly one time, and the RequesterRole can be instantiated one or more time. The second adaptive capacity of connector is the ability to define generic (templates-like) parameters that allow to parameterize the connector with types. This is particularly important to abstract, as far as possible, the connector description from data types used in message exchanges. In Fig.2, the connector has two generic parameters: anyReq, representing the list of the types of the parameters transmitted with the request and anyRep, representing the list of types of the parameters transmitted with the reply. RequestReplyConnection is parameterized as follows: anyReq is valued to void, because no data is transmitted with the message; anyRep is valued with the Velocity and Orientation types pair, because these are the two pieces of information returned by the reply. Protocols being describes into a composition of roles, roles are parameterized entities too.

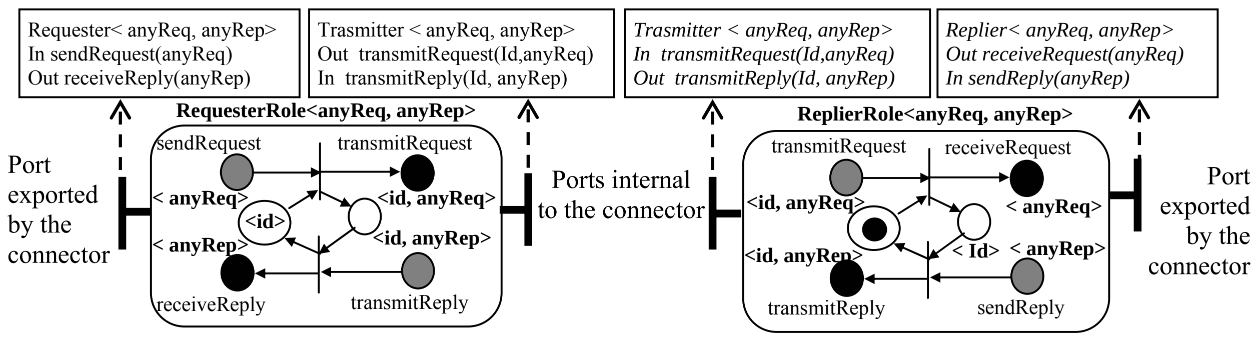
Requester< anyReq, anyRep>
In sendRequest(anyReq)
Out receiveReply(anyRep)
Trasmitter < anyReq, anyRep>
Out transmitRequest(Id,anyReq)
In transmitReply(Id, anyRep)
Trasmitter < anyReq, anyRep>
In transmitRequest(Id,anyReq)
Out transmitReply(Id, anyRep)
Replier< anyReq, anyRep>
Out receiveRequest(anyReq)
In sendReply(anyRep)
RequesterRole<anyReq, anyRep>
ReplierRole<anyReq, anyRep>
Port exported by the connector
Ports internal to the connector
Port exported by the connector
sendRequest
transmitRequest
< anyReq>
<id>
<id, anyReq>
< anyRep>
<id, anyRep>
receiveReply
transmitReply
transmitRequest
receiveRequest
<id, anyReq>
< anyReq>
<id, anyRep>
< Id>
< anyRep>
transmitReply
sendReply

Figure 5: RequesterRole and ReplierRoles

A role is a sub-component, part of a connector, that itself has ports, attributes, operations and an asynchronous behaviour, like control components (Fig. 5). But unlike, control components, roles description is completely bounded to connectors one. A role has a provided or a required port *exported* by the connector to make it "visible" outside the connector (and then, connectable with control component's ports). Other ports of roles are internal to the connector (Fig. 4) and are connected by the connector's initialization operation. A role implements the message exchange between the port of the connected control component and its (own) associated port, as well as the message exchange with the other role(s) of the connector (i.e. exchanges inside the connector). Constraints described in the OPN of the ReplierRole (Fig. 5) ensure 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. The OPN of ReplierRole ensures that only one request will be proceed at a time by the component assuming this role. It also describes the way it identifies and memorizes the requester in order to send it the reply. A specific object of type Id, that contains all necessary configuration information to this end, can be transmitted during messages exchanges. RequesterRole sends its own identifier object to the ReplierRole, with transmitRequest message (the state of the Id is "informing"). The ReplierRole uses this Id to identify its clients and then sends it the reply computed by the control component behaviour. In this case, the Id is used to configure communications (its state is "routing"), and not as registering data. When more than one RequesterRole exists, each has a port typed by the Transmitter interface that is connected to the corresponding provided port of the unique ReplierRole. Then their Id are used by the replier to select the receiver of the computed reply. The initialization of role Id is made by the initialization operation of the connector.

The RequestReplyConnection connector can be used to establish connections between different control components, if the interaction to be described corresponds to this protocol, and if ports are compatible. To design a mobile robot architecture, we defined (and used several times) different types of connectors supporting protocols like EventNotification or DataPublishing.

Connectors being also modelled by Petri nets, it allows to build the OPN resulting from the composition of control components (i.e. the model resulting from the composition of all their asynchronous behaviours). Thanks to this property, developers can analyze inter-component synchronizations, allowing then to check, for example, that those interconnections do not introduce any dead-lock.

E. Configurations

Once the control architecture (or part of it) has been completely modelled, the result is a graph of the composition of control components (composition done by means of connectors). The CoSARC language provides another type of component, named *Configuration*, that contains this graph. It allows developers to incorporate a software (sub-)architecture into a reusable entity. Configurations can be used to separate the description of sub-systems, each one corresponding to a *resource* of the robot. The global control architecture can be represented by configuration. At design phase, a configuration can be considered as a control component because it has ports that are connectable via connectors. Ports of a configuration export ports of control components that the configuration contains (dotted lines, Fig. 6). At runtime, any connection to those ports is replaced by a connection to the initial port, i.e. to that of the concerned control component. Fig. 6 shows an example of a configuration: the MobileSubSystem, corresponding to the sub-architecture controlling the vehicle part of a mobile robot. It exports the provided port of the MobileSupervisor and the required ports of VehiclePositionCommand and VehicleObstacleEventGenerator. Since a configuration can contain others configurations, it allows developers to describe the controller architecture at different levels of granularity.

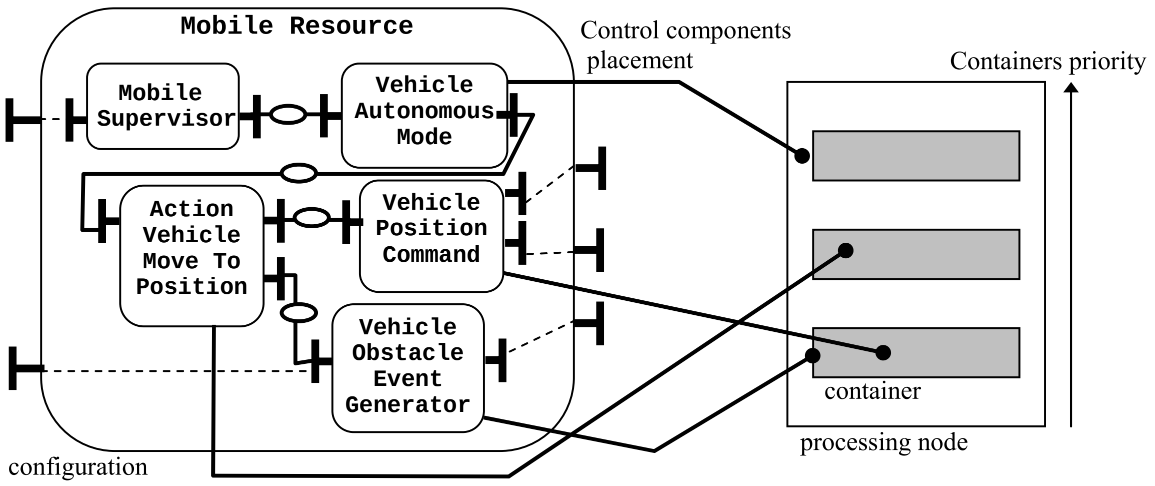

Figure 6: Managing architecture organization: description of the MobileResource by means of a configuration and description of its deployment.

The CoSARC language provides structures to describe the deployment of a configuration. This description is made of two phases:

- the hardware description phase (graphs of nodes, communication networks) allows to define operating system (OS) resources available to execute components,
- the component placement phase allows to define the different OS processes (named *containers*) executing one or more control components and to define the scheduling of these processes on each node. At the deployment stage, configurations incorporate the description used to install and configure components.

This mechanism allows to treat the deployment of an architecture independently of the control behaviour it defines. We chose to treat the organization of a control architecture into layers (hierarchy) during the deployment phase. A container is the unity that is useful to (parts of) layer's description. The relationships between layers are translated into container execution configuration: container's process execution priorities are set depending on layer's relationship (the upper is the layer represented by the container, the higher is its priority of reaction). One future research is to find a multi-criteria scheduling algorithm (dealing with temporal constraints in addition to containers priorities) which will be more adapted to the management of layers hierarchy, in order to ensure maximal reactivity of low layers without sacrificing pre-emption of upper layers.

IV. DEPLOYMENT & EXECUTION MODEL

The CoSARC language is not only a design but also a programming language. Then, it needs a framework to execute components. This framework is a middleware application that runs on top of an OS. It provides standardized access to OS functionalities, and a set of functionalities specific to the CoSARC components execution. It is also in charge of configuration deployment (i.e. the deployment of control components and connectors corresponding to the description made) by creating containers and by managing their scheduling on each processing node.

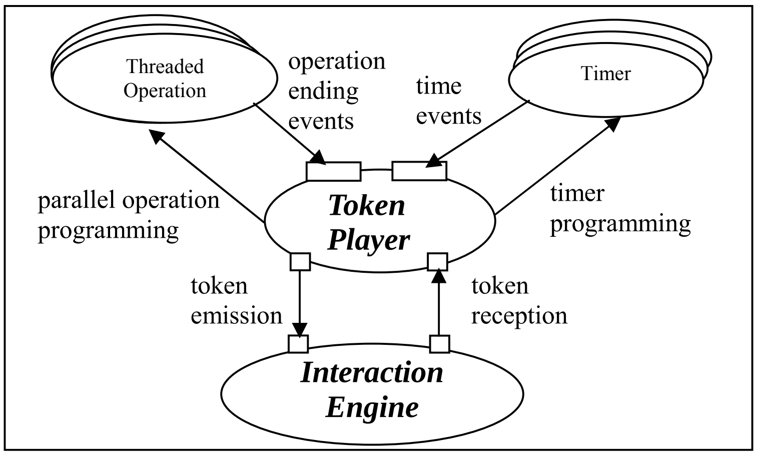

Figure 7: Container's internal structure

A container is in charge of the execution of a set of control components and the set of all the roles played by these components. Any number of control components and roles can be placed into one container, and many containers can be deployed on the same processing node. It supports OPN execution and roles communications by the mean of two software processes that interact with an asynchronous communication model (Fig. 7): the *Token Player* (TP) and the *Interaction*

Engine (IE). The TP is a kind of OPN inference engine that executes the byte code in which is compiled the OPN resulting from the composition of all the asynchronous behaviours of roles and control components contained in the container. During its execution, it can program threads for parallel operation execution and timers for timed transition management. During execution, the TP communicates with the *Interaction Engine* (IE) for emission and reception of external messages. The IE manages run-time communications of control components that are contained in the corresponding container. These communications between control components are configured depending on the connection of the roles they play. At run-time the IE of a given container exchanges messages with IE of others containers according to roles connection information. Given a container, its IE unpacks received messages to give corresponding tokens to the Token Player. And vice-versa its IE packs up tokens arriving from its TP to send the corresponding messages (Fig. 8).

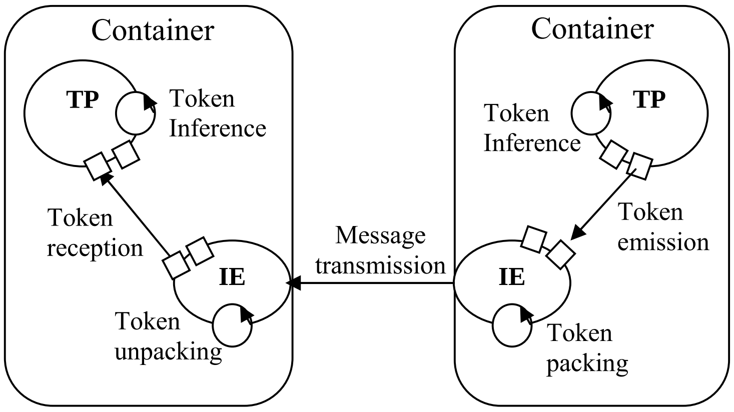

Figure 8: container communications

In the next subsections we first present components deployment model and we focus on OPN execution mechanism in the second one.

A. Deployment Model Overview

The execution of CoSARC components is completely configured by their deployment. The description of configuration deployment is useful to describe precisely components execution issues. It is used to configure containers priorities, but it also helps determining where and how OPN are executed. The deployment is realized by the following steps:

- The *component placement step* consists in creating each container on nodes and placing components code inside containers, corresponding to deployment description (Fig.6).

- The *role assignment step* consists in defining where roles are executed. This is automatically deduced from the preceding step by applying the following rule: when a control component plays a role, this role is executed in the same container as the control component. Fig.9 shows that VehiclePositionCommand and RequesterRole are placed inside the same container. A connector can be then distributed among different containers.

- The *behaviour execution model definition step* consists in producing a global OPN that will be executed by the TP. A container OPN model can be made of the disjoined union of the *complete control component behaviour*. A complete behaviour is an OPN model of the fusion of a control component's and role's asynchronous behaviours. This fusion is deduced from each connection between a control component and a role it plays: each place concerned with message exchanges, of a control component's OPN, is merged with the corresponding place of a role's OPN, according to port connection and interface matching. The resulting OPN, executed by a container, is thus made of as many "complete behaviours" as control component it has to execute. These behaviours can communicate between each others (if connections between their roles exist) and they can communicate with behaviours contained in other containers. Fig. 9 gives an example of this step: it shows that, for example, places P1 and P3 are merged, because they are bind to the same sendRequest service of the Requester Interface.

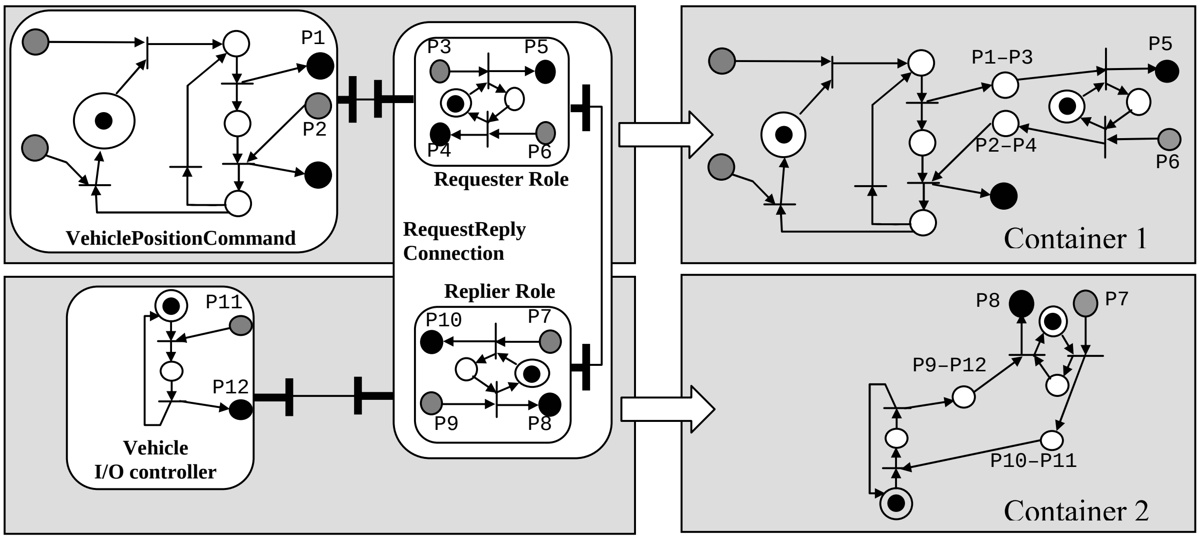

Figure 9: deployment of containers - OPN byte code compilation.

- The *container communication configuration step* consists in defining communications between (and inside) Interaction Engines of containers, according to connections between ports of roles (Fig. 4). For example, the RequesterRole r1 and the ReplierRole r2 are connected by their ports typed by the Transmitter interface. The interaction described in the two Transmitter interfaces (Figs. 4, 5) implies that r1 sends transmitRequest message to r2 and that r2 sends transmitReply message to r1 (once their ports are connected).

Configuring communications supported by Interactions Engines is made in different steps. First it requires to determine relations between Interactions Engines. This is deduced by applying the following operation:

```
foreach port p of a role r executed by a container c
   foreach port p′ of a role r′ executed in container c′
      if p connected with p′
         configure message reception and emission between c and c′
         with information of p and p′ interfaces.
   end
end
```

Second, it consists in defining the system communication supports (pipe, TCP/IP, etc.) used to make IE communication effective. This is done in accordance with connector deployment. If a connector is deployed on one container, the communication is local to the IE, so the IE directly route tokens without using OS communication support. If it is deployed on two or more containers placed on the same processing node, the communication relies on OS process communication procedures (e.g. Mailbox). If the connector is deployed then a network communication protocol has to be used (e.g. TCP/IP). For the moment, we don't consider network distribution problems.

Finally, it consists, for each IE, in doing the matching of the message reception and emission points on one hand, and respectively input and output places of OPN played by the TP on the other hand. This information is directly extracted from ports description (ports reference input and output places associated to message transmission). The IE can then, packs tokens arriving from the token-player into emitted messages, and unpacks token from message arrivals.

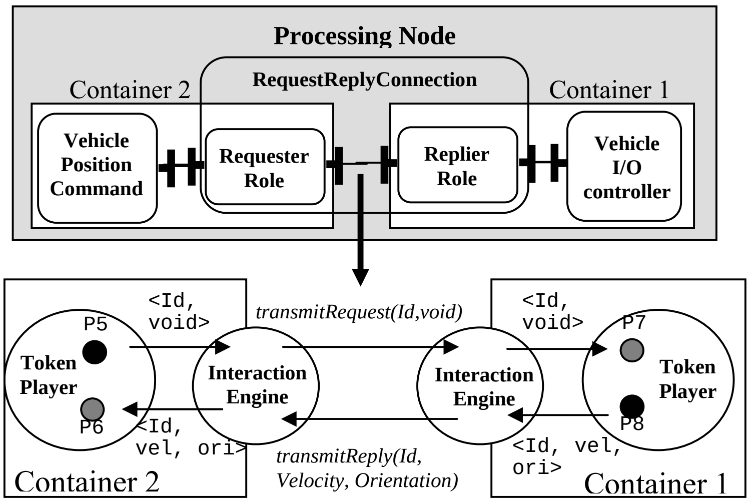

Figure 10: Configuring containers communications with connector and deployment information.

Fig. 10 shows an example of the configuration of the interaction engines of container 1 and 2 according to the connector used and its deployment. We can see that IE of container 1 packs tokens coming from the place P5 into a transmitRequest message and sends the message to container 2. The IE of container 2 unpacks the token from the message and puts it to the place P7.

B. OPN Execution Model Overview

Basically there are two main approaches for implementing a discrete-event controller specified by means of a Petri net [VAL, 95]. For the first one, by means of a procedural language, a collection of tasks are coded in such a way that their overall behaviour emulates the dynamics of the Petri net. For the second one, the Petri net is considered as a set of declarative rules. Then an inference engine which does not depend on the particular Petri net to be implemented operates on the data structure representing the net. This inference engine is the Token player that propagates tokens with respect to the semantic of OPN formalism. The TP also executes functional calls contained in condition and action parts of OPN's transitions. Operations can be threaded when their execution is too long and may block the inference for a too long time. The TP also programs timers when it needs time events to be monitored (time-out, periodic, delay events) to deal with timing notations on transitions. When timer or thread execution is finished, they send corresponding internal events to the TP. The argument to use a TP instead of a direct compilation into a programming language, is that the state of the OPN during execution is reified, allowing then to put in place introspection (dynamic study of OPN state) and reflexive (dynamic change of the OPN state) mechanisms. Introspection is useful to reason, at run-time, about sequences of events, in order to detect a problem and elaborate a diagnosis. Reflexivity is useful to correct (or modify) the OPN state, one diagnosis is done.

The Token Player inference mechanism [PAS, 02] is event based: PNO tokens are propagated in the PN control structure, as far as possible according to OPN propagation rules, and then stands for new events to reactivate the inference mechanism. In order to optimise the OPN inference, its structure is compiled into an equivalent executable structure (Fig. 11). The principle is to decompose transitions into an optimised graph of transition nodes (test, joint, time and action nodes are only used if required); the propagation mechanism is applied on this resulting graph.

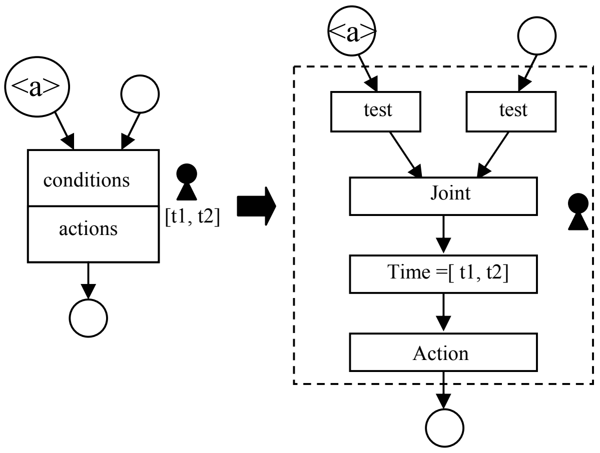

Figure 11: compilation of a transition into an executable structure

The propagation mechanism propagates tokens as far as possible within this resulting graph. The TP starts from new marked places of the PNO and propagates tokens of each new marked place through transitions, i.e. the deepest as possible within the equivalent graph. For example, in Fig. 11, the token will be propagated to the first node ("test"). In the case of a successful test, the token will be blocked before the "Joint" node until a token arrives in the adjacent "test" node and satisfies this test. When done, the two tokens are used to verify test associated to the "Joint" node. If this step is passed, tokens continue their propagation to the "Time" node and stands for the time event to occur (if specified). Once the time event has occurred, the propagation will continue to the "Action" node, where operations will be executed (as well as eventually new token creations). When new tokens have been created and put into one or more post-places, these places are considered to be newly marked ones.

When the propagation is not possible anymore (i.e. there is no newly marked place), the OPN inference mechanism is in a “stable state”. A “stable state” is a state in which token propagation is waiting for event occurrences to be pursued (Fig. 12). So, in a stable state, the token player stands for internal events (time event or parallel operation ending) and/or external events (message arrival) that will reactivate token propagation. For instance, when an external event occurs a new token is created and put into the corresponding Input Place; such newly marked place leads the propagation to start again.

The right part of Fig. 12 depicts the propagation of tokens from newly marked place. When all tokens have been propagated from such place to its post-transitions, this place is no more considered as a newly marked place. If a transition is fired (action is executed) then new tokens are created and put down into post-places which become then newly marked places. When all the post-transition of a given place have been treated, then the token player checks if new internal events occurred. If none, the propagation pursues with another newly marked place. On the other hand, if an internal event has occurred, it is treated before pursuing with the newly marked places. When all the newly marked places have been considered and in lack of internal event occurrences, the token player reaches a "stable state".

In this inference mechanism, we distinguish internal and external events: internal events are monitored more frequently than external ones (Fig. 12). This distinction is made because of the nature (meaning) of the events. For reactivity purposes, internal events, and particularly time events, must be handled as quickly as possible. Indeed, such events can correspond to watchdogs for example. External events result from message arrivals and represent communications between component instances. Internal reactivity (reaction and propagation) is considered as having priority over external request. When tokens are put into output places, these tokens are given to the Interaction Engine in charge of sending them as parameters of messages.

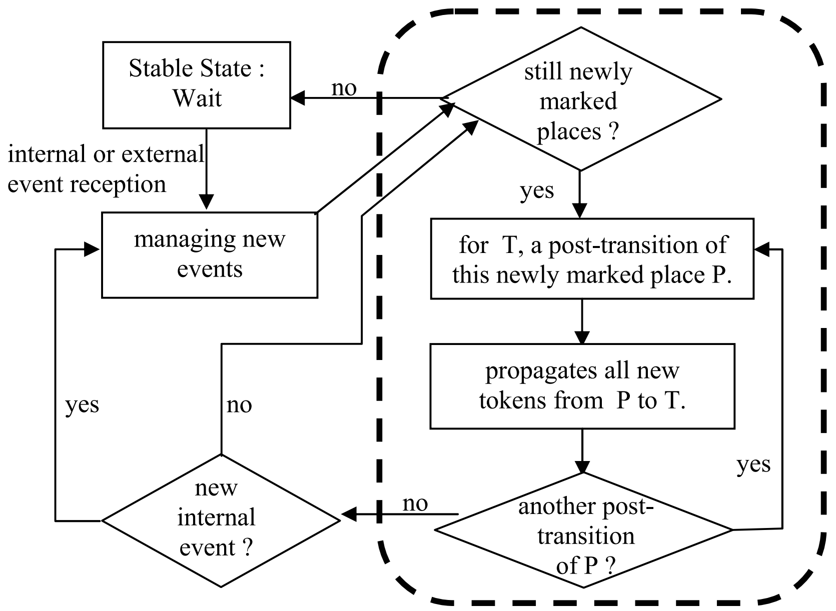
Stable State : Wait
no
still newly marked places ?
internal or external event reception
yes
managing new events
for T, a post-transition of this newly marked place P.
propagates all new tokens from P to T.
yes
no
yes
new internal event ?
no
another post-transition of P ?

Figure 12: Simplified Token Player inference mechanism

For determinism and performance purposes, the token player relies on:
- A real time operating system which allows to manage time events thanks to real time clocks, and to define precisely component instances process execution priorities.
- The determinism of the executed structure that is possible thanks to OPN transition firing priorities.
- The efficiency of the propagation mechanism. It allows to avoid polling of the OPN (cyclic execution) and consequently optimises its execution (and processor use).
- The robustness of the inference mechanism which guarantees that no evolution will take place if an incoherent or a non-pertinent event occurs.

Actually, the TP has been developed and tested in order to validate inference mechanism. A complete and real-time version of container's execution mechanism is under development. Representation components are translated into objects and their types are translated into classes. Each of these classes implements the object interfaces corresponding to the interfaces typing the provided ports. Each required port is translated into a specific attribute that references the object interface corresponding to the interfaces typing the required ports. Connections of representation component ports are also manageable by means of a specific connection object.

V. CONCLUSION

We have presented the CoSARC methodology, which is devoted to improving quality, modularity, reusability and the upgradeability of robot control architectures, along all the architecture life cycle, from analysis to execution. To this end, the methodology relies on two aspects: an architecture pattern and a component-based language. The proposed architecture pattern helps in many way for the organization of control activities, by synthesizing main organization principles. It also gives a way for identifying control activities and their interactions with respect to material elements of the robot. It is also specifically dedicated to the reification and the integration of human expertise (control laws, physical descriptions, modes management, observers, action scheduling, etc.). The CoSARC language deals with design, implementation and deployment of control software architecture. It supports four categories of components, each one dealing with a specific aspect during control architecture description. Moreover, it has the added benefit of relying on a formal approach based on Object Petri Nets formalism. This allows analysis to be performed at the design stage which is a great advantage when designing the control of complex systems.

Current works concern the development of the CoSARC execution environment and of the CoSARC language development toolkit.

REFERENCES

[ALA, 98] Alami, R. & Chatila, R. & Fleury, S. & Ghallab, M. & Ingrand, F. *An architecture for autonomy*, International Journal of Robotics Research, vol. 17, no. 4 (April 1998), p.315-337.

[ALB, 02] Albus, J.S. & al.*, 4D/RDC: A reference model architecture for unmanned vehicle systems*. Technical report, NISTIR 6910, 2002.

[ALD, 03] Aldrich, J. & Sazawal, V. & Chambers, C. & Notkin, D. *Language support for connector abstraction*, in Proceedings of ECOOP'2003, pp.74-102, Darmstadt, Germany, July 2003.

[ARK, 97] Arkin, R.C. & Balch, T. *Aura : principles and practice in review*. Technical report, College of Computing, Georgia Institute of Technology, 1997.

[BIN, 96] Binns, P. & Engelhart, M. & Jackson, M. & Vestal, S. *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.

[BOR,98] Borrely, J.J. & al. *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.

[BRO,98] Brooks, R. & al.. *Alternative Essences of Intelligence*. in Proceedings of American Association of Artificial Intelligence (AAAI), pp. 89-97, July 1998, Madison, Wisconsin, USA.

[BRO, 86] Brooks, R.A. *A robust layered control system for a mobile robot*. IEEE journal of Robotics and Automation, vol. 2, no. 1, pp.14-23, 1986.

[BRU, 02] Bruneton, E. & Coupaye, T. & Stefani, J.B. *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.

[DAV, 04] David, R. & Alla, H. *Discrete, Continuous and Hybrid Petri Nets*. Ed. Springer, ISBN 3-540-22480-7, 2004.

[MED, 97] Medvidovic, N. & Taylor, R.N. *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.

[MUS, 02] Muscettola, N. & al.. *Idea : Planning at the core of autonomous reactive agents*. In Proceedings of the 3rd Int. NASA Workshop on Planning and Scheduling for Space, 2002.

[OMG, 01] OMG. *Corba 3.0 new components chapters*. OMG TC Document formal 2001-11-03, Object Management Group, December 2001.

[PAS, 02] Passama, R & Andreu, D. & Raclot, F. & Libourel, T. *J-NetObject :Un Noyau d'Exécution de Réseaux de Petri à Objets Temporels*. Research report LIRMM n°02182, version 1.0, LIRMM, France, December 2002.

[SIB, 85] Sibertin-Blanc, C. 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.

[STE, 96] Stewart, D. B. 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.

[SZY, 99] Szyperski, C. Component Software: Beyond Object Oriented Programming, Addison-Wesley publishing.

[VAL, 95] R. Valette. *Petri nets for control and monitoring: specification, verification, implementation.* In workshop « Analysis and Design of Event-Driven Operations in Process Systems, Imperial College, Centre for Process System Engineering, London, 10-11 April 1995.

[VOL, 01] Volpe, R. & al. 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.