Formalizing, Implementing and Reusing Controller's Behaviors and Interactions

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 – This paper presents a formal component-based language used to design and to implement control architectures. This language aims at providing concepts and notations to favor reusability of software components and quality during their design. It is based on an Object Petri Nets notation for behaviors and interactions descriptions, but also for execution purposes.

Keywords: Control Architecture, Components, Interactions, Object Petri Nets.

I. INTRODUCTION

Nowadays, the complexity of industrial and robotic control systems design is continuously increasing. Such systems are now often made of more or less 'autonomous' control sub-systems, that put in place interactions between each-other in order to obtain a global task realization (production plan, robotic missions, etc.). Such systems have also to be more evolutive, allowing to easily change or modify some parts of their hardware and software architectures. In this field, there are many challenges to overcome, some of them are related to software architecture design and implementation. A software control architecture describes the different software components and interaction protocols that respectively impose a behavior to sub-systems and describe their interactions (communications, synchronizations). Of course, each control system unity can be itself composed of many components and interactions (that describe the local behavior of the sub-system). Managing control activities (system behavior's parts) and interactions in these systems is a very complex task. It requires both quality of design, implementation reuse, and modularity of the architecture.

One major trend in software engineering techniques, is to consider that, during software architecture description, interactions are also as important as computations, like in architecture description languages (ADLs) [MED, 97] [ALD, 03]. Computations and protocols descriptions are made separately, and are put together at architecture description time. This property is really interesting when dealing with interactions and control activities reuse, and with modularity of the architecture. Another trend, that appears for example in Wright ADL [ALL, 97], is to formalize architecture descriptions by the means of specific notations to describe component's behavior and connections (interaction protocols). This is a great advance for the 'quality of design' management, because these notations allow for the use of formal analysis techniques. Concurrent object oriented languages [SIB, 98] also uses Object Petri Nets [SIB, 86] to model and to program complex object's behaviors. All these modeling and implementation techniques could be combined in order to satisfy needs during the controller design process.

The paper presents a component-based modeling and implementation language that specifically deals with the management of a 'quality-based' design, of software reuse and modularity of architectures. Section 2 gives an overview of the language. Section 3 and section 4 presents language notations used to describe respectively control activities and interaction protocols. CoSARC Language formal aspects are discussed in section 5. Section 6 shows the deployment influence on formal notation compilation and presents components execution and communication issues. Finally section 7 concludes this paper by citing actual works and perspectives.

II. COMPONENT-BASED LANGUAGE OVERVIEW

The CoSARC (Component-based Software Architecture of Robot Controllers) language is devoted to the design and implementation of control architectures. This language draws from existing software component technologies [SZY,99] such as Fractal [BRU, 02] and Architecture Description Languages such as Meta-H [BIN, 96]. It proposes a set of structures to describe the software 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.

The CoSARC language provides different types of components but only two of them are presented in this paper: c*ontrol components* and c*onnectors*. Control components allow for control activities description and connectors allow for interaction description.

III. Control Activities description

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 control components, like for example: Commands (i.e. entity that executes a control law or a control sequence), Observers (i.e. entity in charge of sensor signal analysis, estimation, etc.), Event Generators (i.e. entity that monitors event occurrences), Supervisor, etc. A control component incorporates and manages a set of *representation components* which define the data it manages. In the following parts of this paper, representation components will be considered (to simplify) as objects.

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, plus an asynchronous behavior. Each operation of a control component represents a context change during its execution. The asynchronous behavior of the control component is described by an Object Petri Net (OPN) [SIB, 86], that models its 'control logic' (i.e. the event-based control-flow). Tokens inside the OPN refer to objects 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 behavior 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. Message arrival is represented as grey PN places and message emission as black PN places. 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 modeling 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, 05] 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.

Fig. 1 shows a simplified example of a control component behavior that corresponds to an entity that applies a control law to a vehicle, we named it VehiclePositionCommand. It has three attributes: its periodicity, the Vehicle being controlled and the control law to be applied VehiclePositionControlLaw. The Vehicle and the VehiclePositionControlLaw are connected in the same way as described in Fig. 1, meaning that the VehiclePosition Command will apply the VehiclePosition ControlLaw to the Vehicle at a given periodicity.

This control component's provided port (cf. Fig. 1) 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 behavior of VehiclePositionCommand shown in Fig. 1, 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.

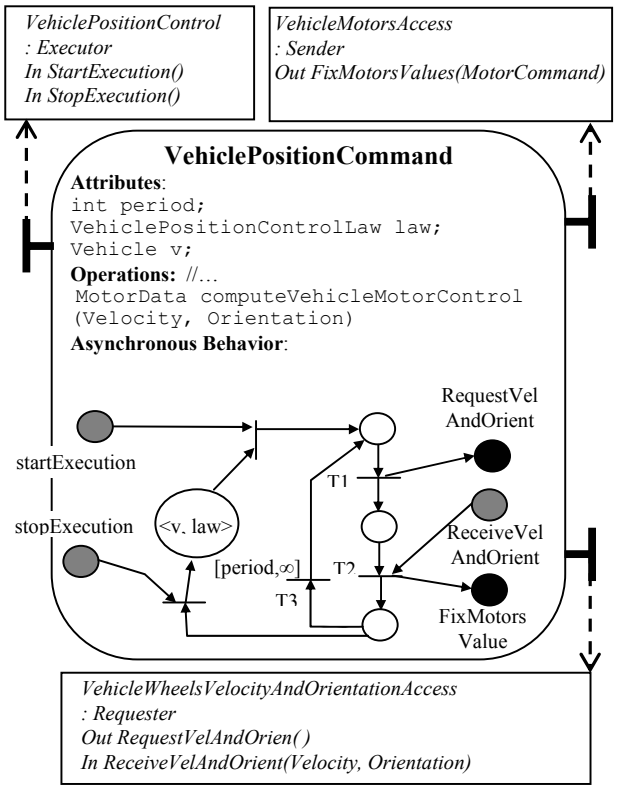

Fig. 1: Simple example of a control component

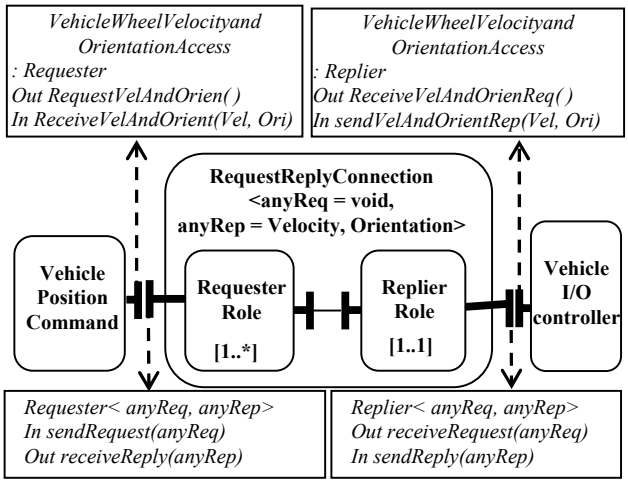

Fig. 2: Simple connector example, connecting two control components

IV. INTERACTIONS DESCRIPTION

Connections of control components are reified into components named *connectors.* An example of a (simple) connector, named RequestReplyConnection, is shown in Fig.2. It is used to connect the control component of Fig.1 with a control component that provides the VehicleWheel-VelocityandOrientationAccess service. 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 that can implement any (application level) interaction protocols, even those 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 their independent reuse.

A connector incorporates sub-components named *roles* (as attributes). Each role defines the behavior'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. 2 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. 2 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. 2). Finally, compatibility of control components ports is verified according to interface names. Fig. 2 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).

A connector can be a very adaptative 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 adaptative capacity of connector is the ability to define generic (templates-like) parameters that allow parameterizing 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. RequestReply Connection is parameterized as follow : 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.

A role is a sub-component, part of a connector, that itself has ports, attributes, operations and an asynchronous behavior, like control components (Fig. 3). 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* (Fig. 2) to the connector and are connected by 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. 3) 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 identify 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 behavior. In this case, the Id is used to configure communications (its state is "routing"), and not as registering data. When more than one RequesterRole exist, 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 Ids is made by the initialization operation of the connector.

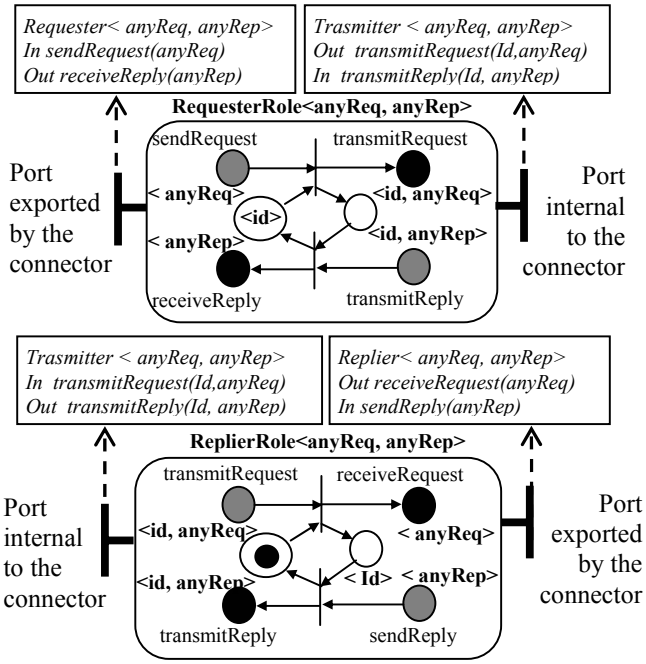

Fig. 3: Description of Requester and Replier Roles

The RequestReply Connection 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.

V. FORMAL ASPECTS

The CoSARC language is based on an extension of Object Petri Nets (OPN) to provide developers a mean to create models of controller's behaviors and their interactions. OPN are supported by a set of techniques to analyze formal properties of a model, which allows to check for its validity (e.g. liveness of behaviors, dead-lock free interactions, verification of invariants). This is a very important aspect for bringing quality during controllers design. For analysis purposes, OPN can be translated into beavirally equivalent Coloured Petri Nets [LAK, 95]. OPN modeling is at the basis of all formal aspects of the CoSARC language. First, it is used for independent reasoning on behaviors and protocols. Second it is used to formalize the composition of control components by means of connectors. To this end, we define two possible Communicating OPN (COPN) management operations: *elimination of unused Input/Output PN places* and *fusion of Input and Output PN places*. These operations are used to transform COPN models, used during control component's behaviors and connector's protocols description, into an OPN (that could be then analyzed, after translation, using analysis techniques of Coloured Petri nets).

Formally, we define a COPN by the following 5-uplet:

$$\mathbf{COPN = < P, T, A, Class, V >}$$

- **P** is a finite state of places, constituted by the union of three sets: $\mathbf{P^i}$ the set of internal places, $\mathbf{P^e}$ the set of input places and of $\mathbf{P^s}$ the set of output places. A place is an 2-uplet **Pla=<Stp, M>**, where **Stp** is a set of n-uplets containing elements of **Class** (i.e. types admitted bu Pla) and **M** the initial marking of the place – i.e. a set of n-uplet containing objects, instances of classes of **Class**. **M** is null for places of $\mathbf{P^e}$ and of $\mathbf{P^s}$.
- **T** is a finite set of transitions. Each transition is a 2-uplet **Tra=<Atc, Ata>**, where **Atc** is a condition that takes for arguments one or more variables of **V**, and **Ata** an action (operation) that takes for arguments one or more variables of **V**. To simplify we don't consider **Atc** and **Ata** properties of transitions, because they are no relevant for analysis purpose.
- **A** is a finite set of arcs. Each arc is a 3-uplet **Arc=<Prec, Foll, Vars>**, where **Prec** and **Foll** are respectively the preceding and the following nodes (a place or a transition) of the arc, and **Vars** is a formal sum of n-uplets containing elements of **V**.
- **Class** is a finite set of object classes, organized in a hierarchy. Classes of **Class** can eventually be shared among different COPN.
- **V** is a set of variables, each being typed by a class of **Class**. Each variable has an unique identifier composed of its own name plus the COPN identifier.

An OPN is defined by a similar 5-uplet : $\mathbf{OPN = < P, T, A, Class, V >}$, where **P** is only made of the $\mathbf{P^i}$ set (no input neither output place in the OPN). The two COPN management operations are:

- ***elim:*** $COPN \rightarrow OPN$, the operation that produces an OPN by eliminating Input and Output PN places from a given COPN. Formally, given a COPN *copn* and an OPN *opn*, it is equivalent to apply sequentially the following statements:
  - $\mathbf{P}(opn) \leftarrow \mathbf{P^i}(copn)$
  - $\mathbf{T}(opn) \leftarrow \mathbf{T}(copn)$
  - $\mathbf{A}(opn) \leftarrow \cup\{a \in \mathbf{A}(copn) \mid (\mathbf{Prec}(a) \notin \mathbf{P^e}(copn)) \wedge (\mathbf{Foll}(a) \notin \mathbf{P^s}(copn))\}$
  - $\mathbf{Class}(opn) \leftarrow \mathbf{Class}(copn)$
  - $\mathbf{V}(opn) \leftarrow \cup\{v \in \mathbf{V}(copn) \mid (\exists\ a \in \mathbf{A}(opn) \mid v \in \mathbf{Vars}(a))\}$

- ***fuse:*** $\mathbf{set}\{COPN\} \otimes REG \rightarrow COPN$, the operation that produces a COPN from the composition of a set of COPN. This operation consists in merging output and input places of a set of COPN, in order to create new internal places. The domain REG contains a set of 'place relations' (*reg*) in which each relation ***r*** determines a fusion between a set of input places and a set of output places. An element ***r*** of *reg* is a set of input and output places. The ***Pfusion*** function take an element ***r*** for argument to produce an internal place by fusion of input and output places of ***r***. Formally, given a set of COPN S*Copn*, a resulting COPN *RCopn* and a set of 'place relations' *reg*, the ***fuse*** function is equivalent to the following statement:
  - $\mathbf{P^i}(RCopn) \leftarrow \cup_k\{\mathbf{P^i}(\mathrm{SC}opn_k)\} + \cup_j\{\mathbf{Pfusion}(reg_j)\}$
  - $\mathbf{P^e}(RCopn) \leftarrow \cup_i\{p \in \mathbf{P^e}(\mathrm{SC}opn_i) \mid \forall r \in reg,\ p \notin r\}$
  - $\mathbf{P^s}(RCopn) \leftarrow \cup_i\{p \in \mathbf{P^s}(\mathrm{SC}opn_i) \mid \forall r \in reg,\ p \notin r\}$
  - $\mathbf{T}(RCopn) \leftarrow \cup_i\{\mathbf{T}(\mathrm{SC}opn_i)\}$
  - $\mathbf{A}(RCopn) \leftarrow \cup_i\{a \in \mathbf{A}(\mathrm{SC}opn_i) \mid \forall r \in reg, (\mathbf{Prec}(a) \notin r) \wedge (\mathbf{Foll}(a) \notin r\} + \cup_i\{a \in \mathbf{A}(\mathrm{SC}opn_i) \mid (\mathbf{Prec}(a) \leftarrow \mathbf{Pfusion}(r)\ if\ \exists\ r \in reg \mid \mathbf{Prec}(a) \in r) \vee (\mathbf{Foll}(a) \leftarrow \mathbf{Pfusion}(r)\ if\ \exists\ r \in reg \mid \mathbf{Foll}(a) \in r)\}$
  - $\mathbf{Class}(RCopn) \leftarrow \cup_i\{\mathbf{Class}(\mathrm{SC}opn_i)\}$
  - $\mathbf{V}(RCopn) \leftarrow \cup_i\{\mathbf{V}(\mathrm{SC}opn_i)\}$

with ***Pfusion: set{place}*** → *place* the function that creates an internal place by the fusion of places passed for arguments. Formally, given a set of places *Spla* and a resulting place *Rpla*, it is equivalent to the following statements:

- ***Stp***(*Rpla*) ← $\cup_i${***Stp***(Spla$_i$)}
- ***M*** (*Rpla*) ← ∅

Mechanisms allowing to build analyzable OPNs from COPN of components are all based of the ***elim*** and ***fuse*** operations; they are used for behaviors, protocols, and composition analysis.

A. Behavioral analysis model

The behavioral analysis consists in independently analyzing the asynchronous behaviors of control components and roles. In order to get the analyzable OPN of a control component behavior, or a role behavior, the ***elim*** operation has to be applied on the COPN description of their asynchronous behaviors. This provides a behavior without any input or output place (messages calls and reception are of no relevance).

The main interest of this mechanism is that it allows the verification of properties on each behavior.

B. Protocol analysis model

The protocol analysis consists in analyzing the protocol incorporated inside a connector. In order to get the analyzable OPN of a connector protocol, different steps have to be done.

The first one consists in deciding roles composition which leads to determine the set of 'place relations' between the role's input and output places. To this end we have to consider that:

- Role's ports refer to input and output places of their behaviors.
- Connector's internal connections are established between ports of roles.

So, the set of 'place relations' between input and output places can be easily found thanks to ports interfaces matching. For example, the assembly between the RequesterRole and the ReplierRole (Fig. 3) is defined by connecting their ports typed by the Transmitter interface. This results in identifying the input and output places that are responsible of transmitRequest and transmitReply message exchanges, corresponding respectively to the fusion of places P5 (RequesterRole) and P7 (ReplierRole) and to the fusion of places P6 (RequesterRole) and P8 (ReplierRole) (Fig. 4).

Once the 'place relations' have been established, the second step consists in applying the ***fuse*** operation to all the roles of the connector. This leads to the building of the COPN model of the connector protocol.

The third step consists in applying the ***elim*** function to this COPN model in order to obtain a global OPN analyzable model (without message emission or reception between roles and control components).

The protocol analysis model is useful, for example, to detect connector's internal dead-locks or verify the respect of invariants.

C. Composition analysis model

The last mechanism consists in analysing the composition of control – i.e. analysing the global behavior issued from the composition of behaviors of control components and protocols of connectors.

The first step to create the analyzable OPN model of the composition is to build the connector's protocol OPN model, by following the two first steps of protocol analysis model creation mechanism.

The second step consists in identifying 'place relations' between the role's input and output places on one hand, and control component's ones, on the other hand. To this end we have to consider that:

- Role's ports refer to input and output places of their behaviors.
- Control component's ports refer to input output places of their behaviors.
- When a control component plays a role, one of its ports is connected to the corresponding port of a connector's role.

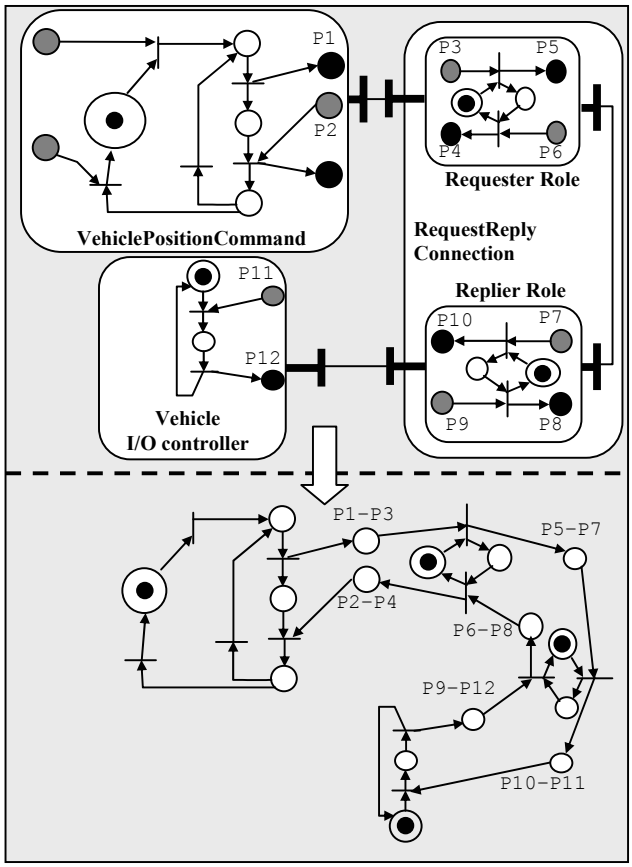

Fig. 4: formal composition model - COPN place fusion and elimination

So, the set of 'place relations' between input and output places can be easily found thanks to port's interfaces matching. For example, the RequesterRole's provided port is type by the Requester interface and VehiclePosition Command's port is typed by the VehicleWheelsVelocity AndOrientationAccess interface, which is compatible with the Requester interface (Fig.1 , Fig.2). This results in identifying the input and output places that are responsible of sendRequest and receiveReply message exchanges, corresponding respectively to the fusion of places `P1` and `P3` and to the fusion of places `P2` and `P4` (Fig. 4).

Once the 'place relations' have been established, the third step consists in applying the ***fuse*** operation on connector's and control component's behaviors. This leads to the building of the COPN model of the composition. The last step consists in applying the ***elim*** function to this COPN model in order to obtain a global analysable OPN model of the composition (cf. down of Fig.4).

Thanks to this model, developers can analyze inter-component synchronizations, allowing then to check, for example, that those interconnections do not introduce any dead-lock, or that the global behavior respects invariants.

VI. Deployment Description & Execution issues

Control components and connectors are not only modeling entities but also programming ones. The execution model of the CoSARC language is configured by the deployment of components assemblies (i.e. the software architecture). The unit of deployment is the container, which is a system process deployed on a processing node (Fig. 5). A container is able to execute control components and the roles they play. It incorporates a COPN execution mechanism, named token player, that deals with tokens inference, operations execution (even threaded execution is possible) and data objects exchanges. It also incorporates an Interaction Engine that interfaces the Token-player activities with the activities of other containers, by allowing inter-container communications. Any number of control components and roles can be placed into one container, and many containers can be deployed on the same processing node.

The description of software architecture deployment is useful, in our context, to describe precisely components execution issues, particularly it helps determining where and how COPN are executed. The deployment is realized by the following steps:

- The *component placement step* consists in defining in which container each control component is executed (Fig.5). When a control component is placed in a container, it executes its behavior – i.e. its COPN code.
- The *role assignment step* consists in defining where roles are executed. This can be 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.

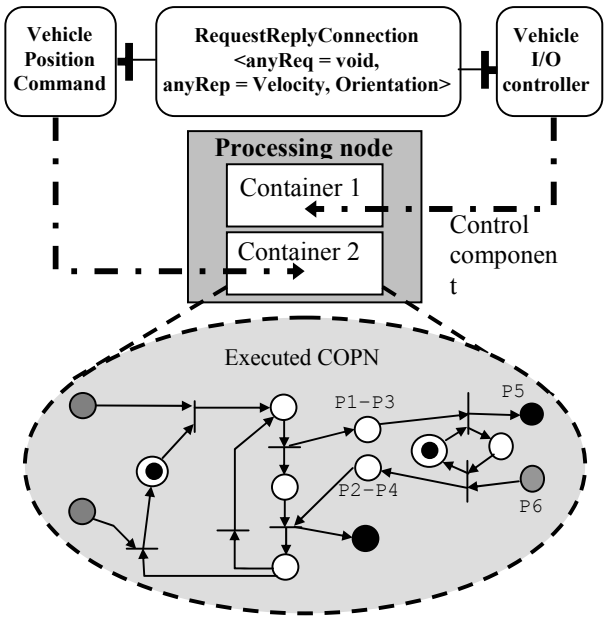

Fig. 5 : Component deployment in containers and COPN execution.

- The *behavior execution model definition step* consists in producing a global COPN that will be executed by the token-player of a container (Fig.5). A container COPN model can be made of the disjoined union of the *complete control component behavior*. A complete behavior is a COPN model of the fusion of a control component's and role's asynchronous behaviors. This fusion is deduced from preceding steps, by applying the following statements :

```
foreach role r played by control component c
    add to reg the place relations between
    the places of r and the places of c,
    deduced from the connection of their
    ports.
end
Apply the fuse operation with the set made
of the behavior of c and behaviors of all
roles it plays, as first argument, and with
reg as second argument.
```

A global COPN executed by a container is thus made of as many complete behaviors as control component it has to execute. These behaviors can communicate between each others (if connections between their roles exist) and they can communicate with behaviors contained in other containers.

- The *container communication configuration step* consists in defining communications between (and inside) Interaction Engines of containers, according to connectors internal connections between ports of roles (Fig. 6). For example, the RequesterRole r1 and the ReplierRole r2 are

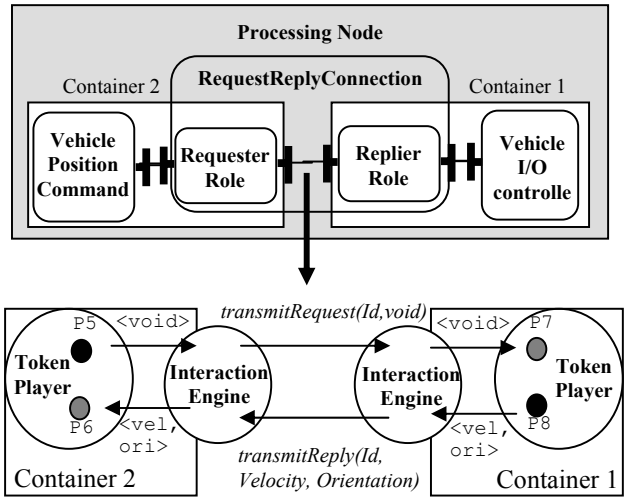

Fig. 6 : Configuring containers communications with connector and deployment information.

connected by their ports typed by the Transmitter interface. The interaction described in the two Transmitter interfaces (Figs. 2 and 3) implies that r1 sends transmitRequest messages to r2 and that r2 sends transmitReply messages to r1 (once their ports are connected). Configuring communications supported by an Interaction Engine is completely deduced from the two first steps 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
```

An Interaction Engine is completely configured when the matching is done between the message reception and emission points on one hand, and respectively input and output places of COPN played by the token player on the other hand. This information is directly extracted from ports description (ports reference input and output places associated to message transmission). The Interaction Engine can then, packs tokens arriving from the token-player into emitted messages, and unpacks token from message arrivals.

Once all containers have been configured by following these steps, the controller can be executed.

VII. Conclusion

This paper has presented a development methodology based on a formal component-based language. The use of this language is to bring quality and reusability of complex behaviors and interactions inside control architectures. The reuse principle relies on the software component paradigm ones: behaviors and interactions description are separated and only merged at architecture description time. Relying on an Object Petri Net Model (OPN) notation for behaviors and interactions description, we show in which way this formalism allow to bring quality of the design process, by allowing the use of PN analysis techniques for diverse aspects of architectures (e.g. components assemblies). This quality centred process is reinforced by the fact that OPN models are directly translated into an equivalent semantics structure that is executable by a token-player. Finally the architecture deployment description allows to configure execution and aspects of components.

A prototype of the language execution environment has been created to validate some middleware mechanisms and a complete version is under development. The language specification is complete and a dedicated development environment is under development.

References

[ALD, 03] J. Aldrich and *al*. “Language support for connector abstraction”. In Proceedings of ECOOP'2003, pp.74-102, 2003.

[ALL, 97] R. Allen. " A Formal Approach to Software Architecture", PhD thesis, Cannergie Mellon University, May 1997.

[BIN, 96] P. Binns and *al*. “Domain Specific Architectures for Guidance, Navigation and Control”. Int. Journal of Software Engineering and Knowledge Engineering, vol. 6, no. 2, pp.201-227, World Scientific Publishing Company, June 1996.

[BRU, 02] E. Bruneton, T. Coupaye, and J.B. Stefani. “Recursive and dynamic software composition with sharing”. In 7th Int. Workshop on Component-Oriented Programming (WCOP02) at ECOOP 2002, Malaga, Espagne, June 2002.

[DAV, 05] R. David and H. Alla. “Discrete, Continuous, and Hybrid Petri nets”. Springer-Verlag, 2005.

[LAK, 95] C. Lakos. “From Coloured Petri nets to Object Petri nets”. Proc. Of the 16th Int. Conf. on Application and Theroy of Petri nets. Lecture Notes in Computer Science 935, pp. 278-297, Torino, Italy, 1995.

[MED, 97]] N. Medvidovic and R.N. Taylor. A framework for Classifying and Comparing Software Architecture Description Languages, in Proceedings of the 6th European Software Engineering Conference, 5th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE), Springer-Verlag, pp. 60-76, 1997, Zurich, Switzerland.

[PAS, 03]. R. Passama, D. Andreu, C. Dony, T.Libourel. “Control System Design using PNO Based Programmable Components”. In proceedings of IMACS Multiconference in Computational Engineering in Systems Applications , 2003, p. 6

[SIB, 85] C. Sibertin-Blanc. “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.

[SIB, 98] C. Sibertin-Blanc. *CoOperative Objects : Principles, Use and Implementation* . Concurrent Object-Oriented Programming and Petri Nets, G. Agha & F. de Cindio Eds, Computer Science XX, 1998.

[SZY,99] C. Szyperski (1999). Component Software: Beyond Object Oriented Programming, Addison-Wesley publishing,1999.