ContrACT: a software environment for developing control architecture.

Robin Passama, David Andreu

ContrACT: a software environment for developing control architecture.

R. Passama, D. Andreu
LIRMM, 161 rue Ada, 34392 Montpellier, France
passama@lirmm.fr, andreu@lirmm.fr

I. Introduction

Robotic software is now one of essential part of robotic system development, therefore software control architecture design methods and concepts, often inspired by engineering software field, are necessary within a robotic project to enhance evolution, modularity and re-usability, and to avoid redesign costs [1]. Control software architecture design approaches are usually classified into three main categories:

- Reactive architectures are built by gathering several modules called behaviors. Each behavior reacts continuously to the situation sensed by the perception system [4]. Actuators control values are obtained from a weighted summation of all commands generated by these modules. The complexity of this method lies in weights adjustment allowing a good reaction and keeping global objective [5][8].
- Deliberative architectures are built in several levels, usually three. Decisions are taken in the higher level; the intermediate level is in charge of controlling and supervising. The low level deals with all periodical treatment related to the instrumentation, such actuator control or measuring instrument management [3].
- Hybrid architectures are a mixed of two previous ones. Usually these are structured in three layers: the deliberative layer, based on planning, the control execution layer and a functional reactive layer [6][7][9][11].

There is no commonly agreed architectures, methods, models, formats... even if some attempts have been proposed by international initiatives like RETF [2], or military defense departments like JAUS [12]. EUROP (European Robotic Platform) also pleads for standardization, notably to provide interoperability between proposed control solutions and to enable rapid technology insertion [14]. Standardization still is an important issue, at least for interoperability purpose. However standardizing architecture implies that all robotics actors agree on what to standardize. That is to say at least to share common definitions, that must not be ambiguous and be comprehensive for any one. But behind this also raises the question of formalism, methods, etc. and to efficiently exploit modeling and formalization efforts to tackle architectural and dependability issues.

Whatever are the organization of the control architecture and the formalism they rely on, most of them are based on modular approaches: objects [10], components [13] or modules [9] are different names given to the software entities according to which is built the control architecture. Depending on the software entity types, the control architecture is obviously built using different mechanisms, but they all aim at:

- describing a control architecture using software entities (pieces of software) that encapsulate algorithms (“algorithm” refers to the functionality to be executed by the robot controller, e.g. a control law),
- allowing the different actors involved to focus only on their concern, like control algorithms and/or others specific aspects,
- avoiding the need to design, every time, a new structure to execute and test it,
- favoring: fast integration of these software entities to build the control architecture, easy replacement of modules, sharing and reuse of modules, extensibility of the architecture, etc.

Let us consider only some examples of architectures or frameworks that illustrate different approaches of “algorithm integration” and “algorithm interaction”. The architecture is based on:

- A set of functions and procedures. The PROCOSA architecture [15] belongs to this approach where a function encapsulates an algorithm, and a Petri net based-procedure specifies the control flow, i.e. the planning and synchronization of functions. A data server ensures control and data flows with the low level.
- A set of objects. CLARAty belongs to this approach where an object encapsulates algorithms that can be generic or specialized ones (methods dedicated to a given robot) [16]. The planner ensures the scheduling of objects' functions execution, taking into account the management of resources.
- A set of modules. In the GENOM approach algorithms (called codels) are encapsulated within modules, according to a generic model of module, some of them being dependent on the instrumentation [17]. Modules offer services (requests), execute activities according to a state automaton, generate reports and posters to exchange data. Procedures of the decisional level specify the sequence, the parallelism and the synchronization of modules (following a decomposition into tasks, actions and modules): the execution of modules being ensured by the execution control level.

  The ORCCAD framework is also based on modules [18]. A Robot-Task (RT) is a control-oriented (basic action), it embeds a port-based composition of modules. The RT’s logical behavior is based on a synchronous state automaton describing the reactive behavior of the RT and an asynchronous execution manager (of modules). Robot-Procedures describe the hierarchical (logical and temporal) composition of RTs, composition done by means of the RTs' interface.
- A set of agents. The HARPIC architecture is based on different agents, interacting by means of messages [19]. “High-level” agents for behavior selection, “low-level” agents for perception and action, and a communication-dedicated agent to support the communication with other robots; all of them being managed by an administrator agent (registration of agents and data exchange).

  The ECA architecture also belongs to this approach [20]. High-level agents ensure the supervisory control and low-level agents are in charge of the control execution; all agents of the application execute activities according to a given state automaton. The communication between agents is ensured by a mediator which establishes a data exchange system based on the publisher/subscriber model.

  Another example of agent based architecture is Robotics4.net in which a main agent is similar to a brain that controls a set of agents considered as body organs [21]. These later, called roblets, are associated to basic robot elements. The body map of the main agent is like a blackboard. The control flow between the main agent and the roblets relies on periodic messages to/from the body map. Roblets interact by means of direct messages (asymmetric pair friendship relation). The data flow is supported by periodic exchanges (messages) of data to/from BodyMap.

  A last example of agent-based architecture is IDEA [22]. IDEA is composed by a set of agents, their composition being independent from any type of architecture. An architecture contains agents for diagnosis, for planning, or agents like functional modules. Local planning and execution controller are embedded within each “control” agent; a plan defines its reactive behavior and its interactions with other agents. Planning is based on a model of relations (causality, temporal dependencies), constraints, compatibility with other agents. Communication between agents is performed by means of messages.
- A set of components. A component implements well defined interfaces and provides access to a set of functionalities; it is a unit of deployment, and sometimes a unit of execution. The composition is based on the contractual nature of the interface (services offered vs. required); two connected components may be assembled with mutual ignorance of internal code. The architecture results from the compositions that are chosen and the way the

components are composed. The CoolBot framework belongs to such approach [23]. The component is an independent execution unit that interacts via ports for data exchange, control and monitoring of its activity. A component embeds two state automatons, one default component's state automaton managing the phases of its life, and a specific state automaton corresponding to its functionalities. The architecture relies on a data flow driven processing (asynchronous communication), using different types of port connections: queue based input port, shared memory based output port, request/answer messages, etc.

The COSARC language [25], whose aim is to provide architectural abstractions to decompose the application according to different dimensions: description of knowledge with representation components (e.g. robot mechanical parts, maps of the environment, geometric and cinematic models, control laws, etc.); description of behaviors using control components and connectors ; description of the deployment of (parts of) the software using configuration components. The composition according to which is built the configuration fixes the architecture type (hierarchical, reactive, hybrid).

A last example is ROS (Robot Operating System), a framework that provides a component-based programming framework that relies on the use of robotic dedicated abstract concepts, named Topics, to specify and enable communication between components.

To summarize, modularity is may be the only "property" commonly and effectively integrated from the design since it already favors within all these different frameworks: adaptability and flexibility (customization depending on the actual mission), durability (maintenance and updates facilitated), competencies sharing and so on. Another common point of most of these approaches is the concurrent nature of the resulting software architecture: applications are structured according to a set of tasks, each task embedding robotic algorithms. But according to the composition mechanism used, the way these tasks are defined can vary. We can notice that a lot of approaches, for instance ROS, Genom, COSARC, ORCCAD, IDEA, bind the composition unit (software programming abstraction) with the execution unit (task scheduled on the OS), simplifying the management of concurrency by opposition to approaches like CLARAty that hides the concurrency into software programming code (of objects). Of course, the ContrACT approach we introduce in this paper also adheres to these core properties; it is based on software entities we call "modules", that are in the same time composition and execution units from which we build the system.

Another absolutely essential issue is the need for software environments to efficiently support methodological concepts, all the above approaches being provided with a suite of tools (compilers, debuggers, verification/validation tools, editors, etc.) and software code (middleware, libraries). These software environments, ease the development process by automating some parts of the code generation process, by providing predefined software component libraries and by automating procedures to check, test or debug applications. To make it usable in practice a control architecture description paradigm needs a powerful software environment, that is why we develop one for ContrACT.

From a technological point of view, the current challenge is still to preserve the independence from hardware architecture (even a distributed one) and operating system without giving up with real-time aspects and to provide solutions allowing the deployment over any target (hardware and OS). ROS is a recent tool to help software developers create robot applications providing interesting hardware abstraction, but dealing with real-time constraints (scheduling) at this level of abstraction remains unsolved. Generally speaking, contrary to decisional and high-level supervision processes management, real-time scheduling is not the main concern of most of the approaches, while it remains a critical aspect. At best, these approaches use basic services of real-time OS (periodical scheduling) to make their system real-time, but they do not specifically focus on real-time algorithms composition, managing real-time tasks at low granularity level. ContrACT is not yet independent from OS, does not either rely on a strict formalism and high-level of abstraction, and

many work still to be done, for example to get it closer to our COSARC [25] approach through which we focused on formalization and abstraction. ContrACT comes from the “robotic ground”, from requirements of actors in terms of possibility to play with real-time preoccupation together with robotic algorithms composition, etc.

The paper firstly presents software abstractions like the programming model and the architectural one. The programming model, based on module, is exposed as well as its port-based composition of modules on which relies the architectural model. The architectural model exploits different types of modules, among which a specific module dedicated to scheduling and a generic module supporting supervisory control. The middleware of ContrACT is then presented in section III, as well as its two specific modules which are the scheduler and supervisor ones. Section IV is devoted to a brief presentation of the software development environment of ContrACT.

II. Software Abstractions

II.1 Programming model

The programming model of ContrACT relies on the concept of module. A module is an independent (with it own context) real-time software task that reacts to a set of predefined requests and communicates with other modules by the means of ports (fig. 1).

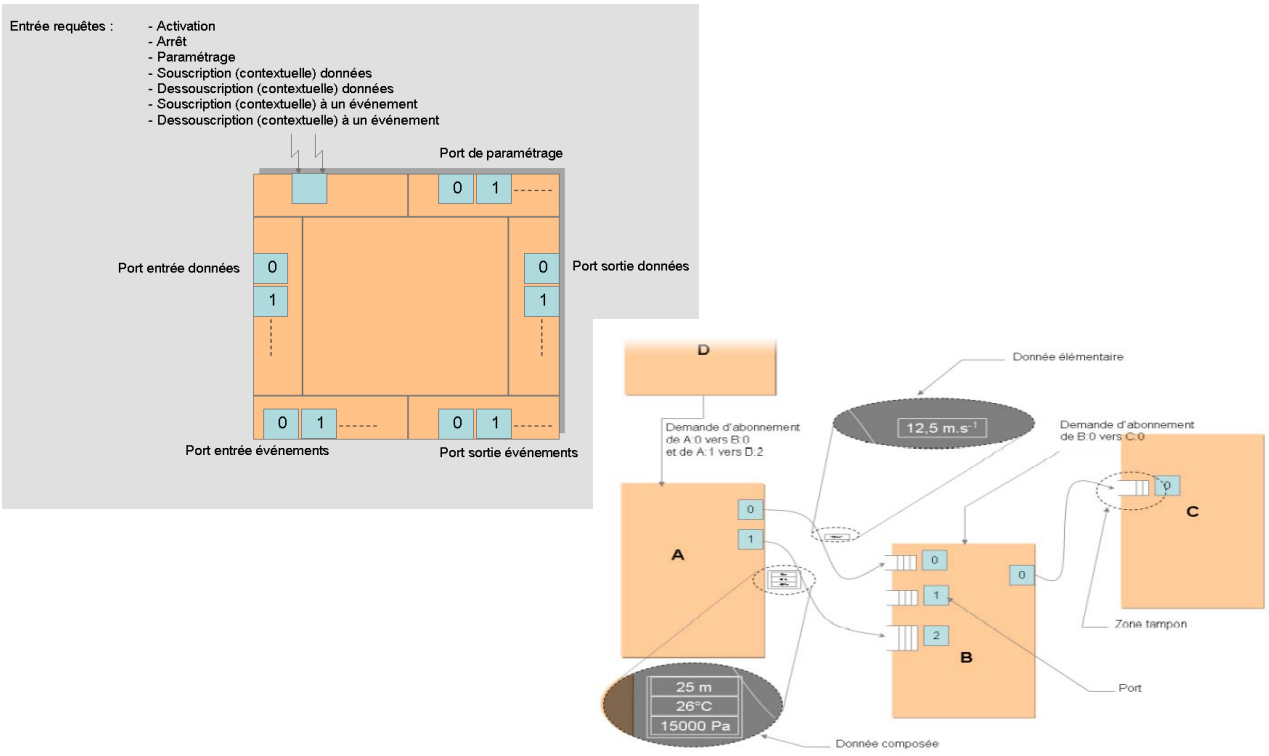

Figure 1: Schematic description of modules and their interaction

Ports are communication points of different types:

- Input and output data ports are used by modules to periodically exchange produced data. When a module publishes data on an output port, this data is copied into the input port of all subscriber modules, so this interaction pattern is based on a publish / subscribe protocol. All data exchanged are timestamped regarding their production date, but to date there is no global timestamping (for a network of PC) but only a local one (single PC).
- Input and output event ports are used by modules to spontaneously exchange generated events. When a module publishes an event on an output port, this event is notified to each of its subscriber modules, so this interaction pattern is also based on a publish / subscribe

protocol. All events exchanged are timestamped regarding their production date.

- Configuration ports (input only) defined by a module are used to set properties and parameters of this module. A module can so directly send a request that contains the adequate parameters to the module to be configured, so this interaction pattern is based on a classical request/reply protocol.
- Consultation ports (output only) defined by a module are used to consult visible properties of this module. A module can so directly send a request to the consulted module and wait for its answer, so this interaction pattern is also based on a classical request/reply protocol.

Data ports and event ports can be dynamically or statically bind according to their input/output status by the means of dedicated subscription requests. Some other requests allow for the activation (start) or the deactivation (stop) of the nominal behavior of the module. All these requests, as well as event notification, configuration and consultation requests all arrive in the unique request-dedicated port of the module, causing the activation of the module which results in the execution of the receiver module as soon as it receives the right to execute by the real-time scheduler. Indeed, the module is continuously waiting on requests arrival and reacts appropriately (fig. 2). Some requests (subscription) are automatically managed internally by the module, while reactions to some others can (quick reactions to an event arrival, to consultation or configuration requests) or must (nominal behavior activated by start requests) be partially customized by the user. During nominal behavior execution, a non blocking lookup of requests arrival takes place to enable quick reaction of the module (typical case is the stop of the execution). Communication with data ports does not cause the activation of receiver modules, neither its blocks the sender module execution: these ports will be read only when the code of the module requires this operation.

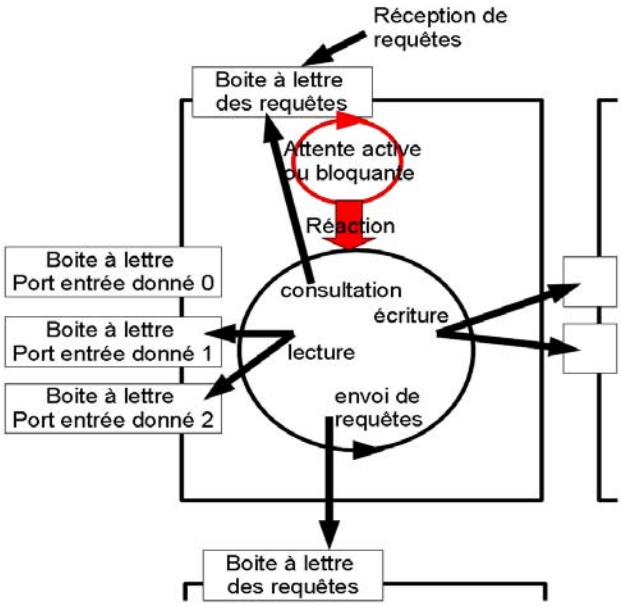

Figure 2: Internal behavior of modules

II.2 Architectural model

From composition possibilities offered by the programming model we derive a control architecture model. In a ContrACT architecture all modules are software tasks, but different classes of modules, and even some specific ones, derived from the previous module model are defined (fig. 3):

- **Asynchronous modules** are used to implement some spontaneous reaction to events, as for instance arrival of messages on a network link. They can also be used to implement long term computation (with no critical time constraint), as planning for instance. They

communicate between each other only using event-based communication.

- **Synchronous modules** are used to implement periodically computed algorithms and periodically performed sampling of sensors and command of actuators. They communicate between each other only using data-based communication.
- **The scheduler module** is the unique module implementing the applicative scheduling algorithms. Its role is to schedule synchronous modules (scheduled modules are dynamically configured with adequate configuration requests) according to a set of constraints defined on the module itself (duration of the execution) and on compositions of modules, named schemes (precedence constraints, critical delay, period). To achieve this scheduling it plays on OS priorities and activation requests sent to modules.
- **Supervision modules** are implementing reactive supervisory control in the architecture. There may be many supervisors but only their configuration change, not their implementation. Supervisors can be seen as specific asynchronous modules that control the execution (asynchronous or scheduled execution), the assembling (data or event communication) and the parametrization/consultation of modules according to events they receive. They can so receive events from synchronous and asynchronous modules but also produce events to other supervisors.

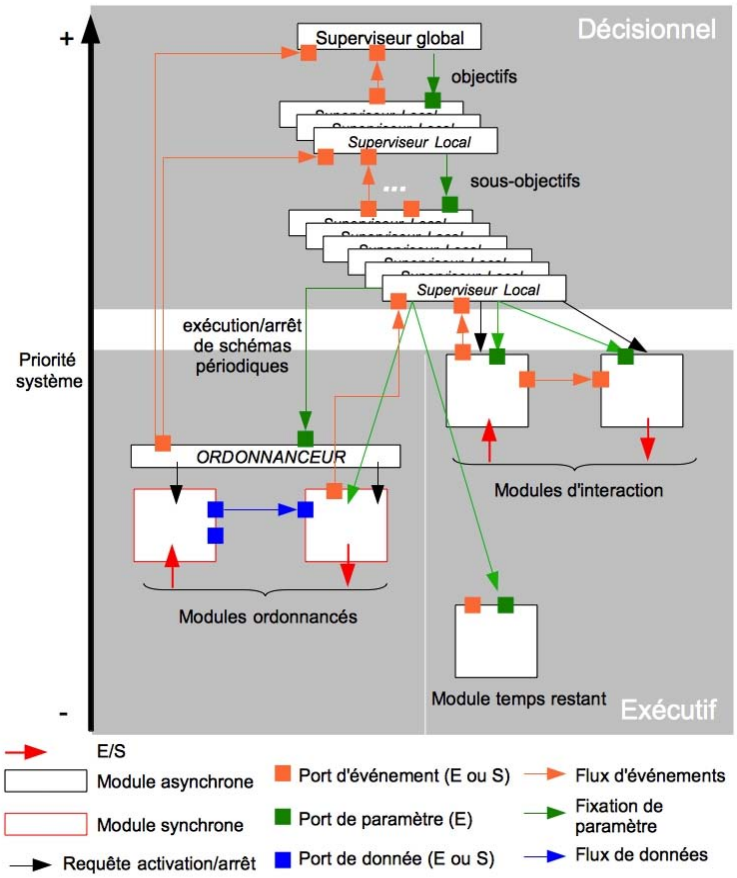

Figure 3: ContrACT architectural model

Modules can be conceptually separated into two interacting layers (fig.3): the decisional layer that contains for now only supervision modules and the executive layer that contains synchronous modules, the scheduler module and asynchronous modules. At a finer grain description, the hierarchical decomposition of architecture is respected defining adequate real-time OS priorities: the priority associated to the task corresponding to a given module determines the importance of this module in the decisional process, from the most important (top level supervisor) to the less

important (remaining-time asynchronous execution). This way we are sure that important reaction (e.g. commutation of control laws) will always take place before less important ones (e.g. execution of the control law). Of course this could cause real-time problem but we assume that supervisors are **reactive** entities: they are executed each time they receive an event by quickly computing the adequate reaction (if any) and then performing this reaction (reassembling, re-parametrization and re-scheduling of modules; generating events to other supervisors). If some modules require a long term computation (e.g. trajectory planning) that could induce real-time scheduling perturbation, they have to be defined as remaining-time asynchronous modules (lowest OS priority).

With this organization, all synchronous modules share the same OS priority range. Their priority is permanently greater than the priority of remaining-time asynchronous modules and lower than that of the scheduler module. This later modifies the priority of synchronous modules from greater to lower value of the given range to respectively allow or withdraw their right to execute.

III. ContrACT Middleware

The Contract middleware is constituted by:

- a library implementing all mechanisms used to program modules according to the programming model, notably mechanisms supporting interactions between modules (requests, data and event exchanges). Its interface is a generic API while its implementation is dependent on the real-time OS used (to date only linux RTAI/LXRT). A module is implemented as a real-time LXRT task, to allow modules to potentially call drivers running only in the linux space (even if it should be avoided depending on the module type).
- code skeleton for each kind of module. These skeletons implement main reaction loops of modules as proposed in fig. 2, but specialized for each kind of module to respect their respective programming constraints.
- an implementation of algorithms used in specific modules of the architecture (scheduling, supervision).

III.1 The ContrACT library and code skeletons

The library providing module programming facilities is the basis of the proposal, from which every other part of the ContrACT middleware has been designed. Code skeletons of modules use the “internal” part of the API (i.e. system configuration functions) provided by the library to implement generic parts of modules, while a programmer uses the “external” part of the API to fill the content of modules regarding a specific applicative context. A schematic representation of the articulation between the generic and the user parts of a module is given figure 4. Internal data structures to be used by the module, like ports, requests, etc. are automatically generated when the module is created, as a real-time task. The system aspects (left side of fig. 4) corresponds to code skeletons using internal functions and structures of the API while the user aspects (right side of fig. 4) is the code using external structures and functions of the API.

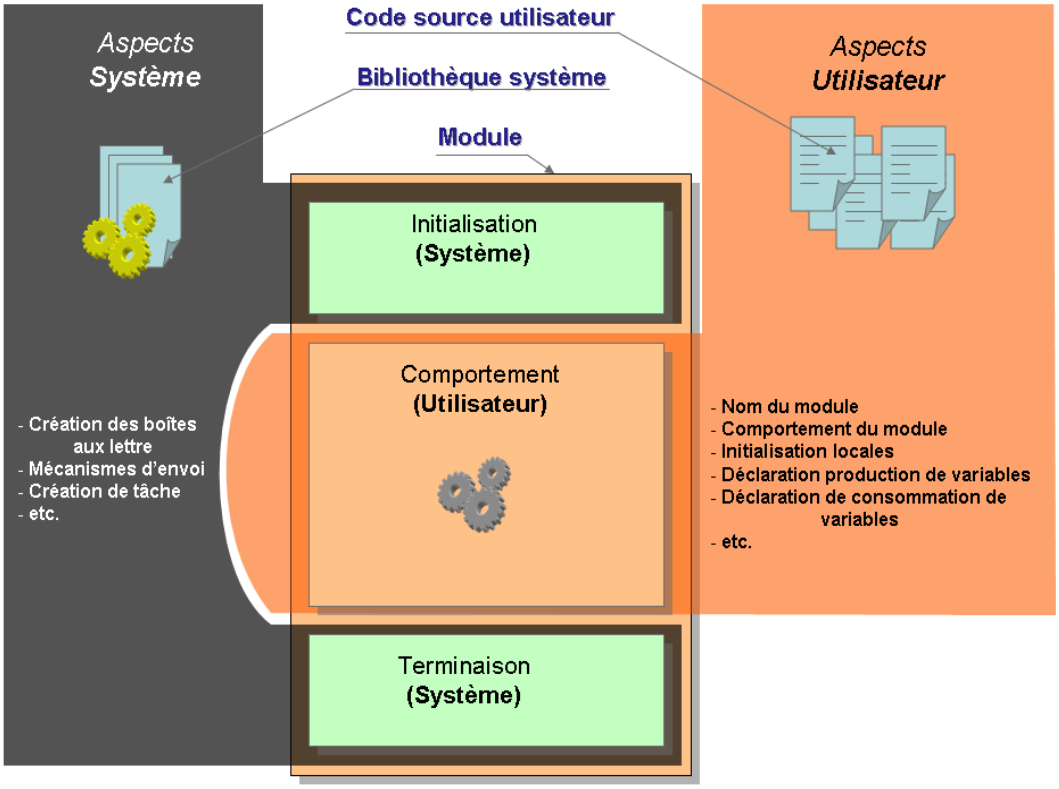

Figure 4: Implementation of a ContrACT module

The behavior of the module is specified by the user through the dedicated function called `ModuleMain`; all system aspects are hidden to the user. He only has to specify (fig. 5):

- The module name, which will be the identifier of the module within the architecture. This name is notably used by the communications primitives (provided by the library) to identify the target mailbox (implemented as a RTAI named mailbox) for a given request/event notification/data publishing.
- The module priority. This priority is the default one but not fixed : somes modules' priorities can be changed dynamically by other modules. For instance, priority of a synchronous module is modified by the scheduler at runtime, or priority of asynchronous modules can change according to their criticity (critical, remaining-time) set by a supervisor.
- The author name and description, only useful for project management (traceability).
- The module input/output data ports (resp. `IUSE` and `IPRODUCE`, on fig. 5) with the associated variables (in which received data will be stored and from which data to be sent are taken). For input ports, subscription to the producer and consumption rate can be statically specified, even if this is usually done dynamically.
- The module input/output event ports (resp. `IREACT` and `IDETECT`, on fig. 5) with the associated variables. The event generation rule can be specified (one-shot or continuous, respectively meaning that the event will be produced only once before automatic unsubscribing or will be generated each time until explicit unsubscribing request).
- The module parameter tables. A table (`PARAM_SIZE_IN`) specifies parameters that can be externally modified (by means of a configuration request). Another table (`PARAM_SIZE_OUT`) specifies parameters (of other modules) that the module can modify through configuration requests.
- The module's visible state tables define all the state variables that can be consulted by other modules (not shown in fig. 5). It follows the same logic than parameter tables.

```
...

// information on module

MODULE_DESCRIPTION("say what the module does");
MODULE_AUTHOR("Your name");

//Exchanges Management Variables
int input, output;
float notify, react;

// module name
char MODULE_NAME[] = "MOD";
//module priority
int MODULE_PRIORITY = 37;

// input data flows : example = { &Variable, PortReception, Size,
Periodicity (important if module name is set), Name of emitter module(can
be unused by setting value to "---"), PortModuleEmetteur} ou { &Variable,
PortReception, Size, 0, "---", 0}
ModuleUse IUSE[] = { {&input, 0, sizeof(int), 0, "---", 0}, IUSE_TERM };

// output data flows : example = { &Variable, PortEmision, Size}
ModuleProduce IPRODUCE[] = { {&output, 0, sizeof(int)}, IPRODUCE_TERM };

//input event flows : example = { &Variable, PortReception, size, module
name (can be unused by setting value to "---"), PortEmision, oneshot}
ModuleReact IREACT [] = { {&react, 0, sizeof(float), "---", 0, 0},
IREACT_TERM };

//output event flows : example = { &variable, PortEmision, size }
ModuleDetect IDETECT [] = { {&notify, 0, sizeof(float)}, IDETECT_TERM };

// size of parameters that can be received in the request mailbox:
size_t PARAM_SIZE_IN[] = { sizeof(double), PARAM_SIZE_TERM };

// size of parameters or events that can be sent in request mailboxes of
other modules :

size_t PARAM_SIZE_OUT[] = {sizeof(char)*256, PARAM_SIZE_TERM };

// description of the module behavior

...
int ModuleMain(int argc, const char * argv[]) {
...
}
```

Figure 5: Implementation of the interface of a module using ContrACT modules library

The request port is a generic one (generated without any specification of the user), it corresponds to the port from which the module activity can be externally controlled (start/stop/kill, with or without ack), configured (by means of specified parameters) and subscribed/unsubscribed to event and data flows. Modules are concerned by different types of requests depending on their type. For example, supervision modules are only reactive to request relative to events and configuration, since it is the way to activate supervisory functions. Asynchronous modules can react to all requests except subscribe/unsubscribe requests since they cannot be implied, to date, in regular data flows. This later restriction is mainly due to technical reasons, to avoid deadlocks due to full data reception mailboxes : contrary to synchronous modules, asynchronous ones cannot produce data perdiodically since their production/consumption rate is variable, so it is impossible to be sure their production does not overflow reception mailboxes capacity (which can cause a deadlock). Nevertheless, this restriction will be removed in the future after a modification of the ContrACT modules library (using non blocking data emission primitives and mailboxes errors reporting/handling at applicative scheduler and supervisors level). Reversely synchronous modules can react to all requests, including subscribe/unsubscribe event request since they can generate event, except event notification request (since they cannot be activated by event reception). The scheduler module only reacts to configuration requests as it is the way to indicate it the scheme to be performed, and some acknowledgments since it controls the activity of synchronous module (an thus can get back acknowledgments to its requests).

III.2 The applicative real-time scheduler

The applicative real-time scheduler is embedded in a specific module of the middleware, dedicated to real-time aspects management. This scheduler manages the execution of synchronous modules, dealing with periodical schemes. A periodical scheme describes an ordered sequence (composition) of synchronous modules to be executed; modules are ordered to respect causal relationship between them, according to the usual “perception-decision-action” process. The applicative real-time scheduler controls the execution of synchronous modules (their corresponding tasks) above the scheduler of the real-time OS. To do so it sends activation request to considered modules and plays with priorities it dynamically associates to the corresponding tasks according to the selected scheduling policy [26]. Two efficient scheduling policies have been used: Earliest Deadline (ED) or Least Laxity (LL). ED algorithm gives the highest priority to the task which has the earliest deadline. LL algorithm gives the highest priority to the task which has the least laxity. The laxity is the time remaining before the task deadline. ED algorithm has been preferred since LL algorithm induced too many context commutations. The scheduling problem has been formalized taking into account two kinds of tasks: treatment tasks which contain an algorithm that will be executed on the processor, and acquisition tasks that are two-part algorithm which first sends a request to a device (e.g. sensor, communication link) and then waits for the response (to get the data). Constraints between tasks are described in terms of task ordering within a given scheme and tasks mutual exclusion (e.g. tasks of different schemes using common or exclusive resources, like interfering sensors for example) [27]. There is no pre-running validation of the schedulinng setup to anticipate CPU saturation, the behavior of the scheduler consist in launching a scheduling setup and doing an online reporting to supervisors each time realtime constraints are violated. This way, problems such as CPU saturation will be indirectly detected.

This real-time scheduler enables a fine grain decomposition of complex robotic algorithms (typically control loops) into individual real-time modules. Doing so, users obtain a better reusability for these algorithms because they can be composed differently according to the global algorithm to put in place. Furthermore, this approach allows to manage precisely real-time constraints execution and reaction to constraints violation, what is not possible when the global control loop is implemented as a periodic task of the real-time OS.

III.3 Reactive and real-time supervisors

Supervisors are specific asynchronous modules in the architecture but there may be several supervisors depending on the hierarchical decomposition of the decisional layer the application requires. Supervisors' interfaces so vary depending both on the event they produce/receive and on the supervision procedure they embed. From user's point of view each supervisor provides a set of specific supervisory control procedures (that can be call through a specific configuration port), each procedure being defined by a specific set of rules. Internally, all supervisors share the same “engine” dedicated to the execution of rules.

The engine is working in a fully asynchronous way: the supervisor waits for event or requests arrival. When it receives an objective execution request, it selects the corresponding supervision function, initializes its parameters according to the request's ones and enables all the rule defining the function (rules can be evaluated by the engine). Once rules are enabled, the engine evaluates them sequentially according to their ordering, only each time a context modification takes place (event reception, internal time event or rules' state modification event). Each rule is made of a precondition (activation constraints), a set of actions (executing, parameterizing, assembling modules; intermediate computations; event notification; calling a supervisor function, etc.) and a post condition (deactivation constraints). The evaluation of an inactive rule consists in evaluating its precondition and the evaluation of an active rule consists in evaluating its postcondition. If a

precondition becomes true, the rule is set as “active” and all actions of the rule are executed sequentially. Reversely, if a postcondition becomes true, the rule is set as “not active” and all actions of the rule are stopped sequentially (if possible, depending on the action nature).

With this approach ContrACT supervisors can describe simply very complex supervisory controllers' behaviors, well coupled with real time aspects. For the description of actions, the engine provides a large set of basic functionalities to dynamically manage the architecture (as a whole or parts of it), considering as well its structure (reassembling of modules) and its parameters (re-parameterization of modules) as its real-time aspects (rescheduling of modules). Furthermore, rules evaluation engine allows to easily implement different kind of state-transition models (state machines, Petri nets, etc.).

IV. Software Environment

The ContrACT middleware provides a simple library, code skeletons, supervision and real-time scheduling modules, to program a control architecture according to the ContrACT architectural model. Nevertheless, its use is not simple in practice if the user has to describe modules' interfaces, supervision rules and real-time scheduling properties “by hand” in the C source code of the modules. That is why we developed a dedicated software development environment (generated with Eclipse modeling and language tools) that automates, as far as possible, the development process and makes it as reliable as possible.

The global approach is based on the use of a domain specific language (DSL) dedicated for the description of real-time control architectures based on the ContrACT architectural model, from which the tools generates RTAI C source code of modules. A ContrACT project is a collection of DSL files describing all parts of the architecture (data exchanged, modules, supervisors) and a folder tree containing the source code (C sources, compilation files, bash command source file to control and test modules) generated by the tool and customized by the user. The ContrACT software environment provides a set of editors, views and actions to edit DSL files, to generate C source code and to navigate across the project (fig. 6). Of course editors of DSL files also provide classical functionalities like syntax coloring, automatic parsing of DSL code, semantic analysis and error reporting.

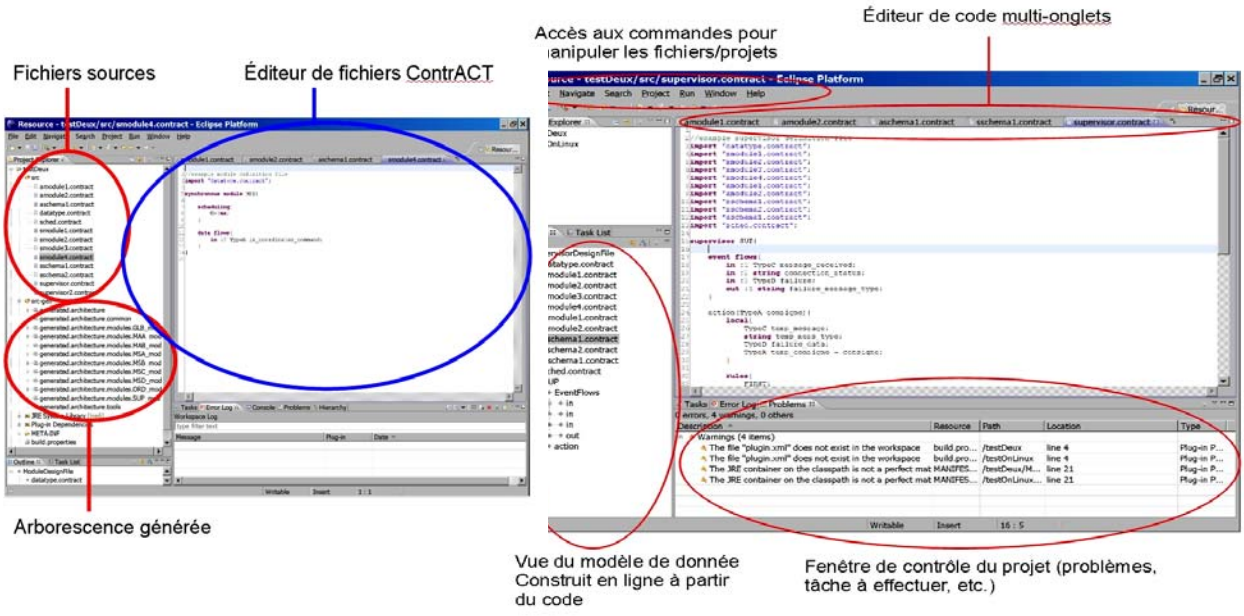

Figure 6: ContrACT projects editing environment

ContrACT language is used to describe all elements of a ContrACT architecture: user defined datatypes and functions, modules' interfaces and real-time properties, schemes and supervisors rules:

- Datatypes files define the types of data shared by modules. Defined data types are used to define modules ports. These files can also (or alternatively) contain the prototype of user functions that are used inside supervision functions. At C source code generation time, each datatype is converted into an equivalent C structure.
- Module files define interfaces and real-time properties of modules. Modules are considered synchronous or asynchronous depending on the key word used (cf. Example 1). All modules can define a set of parameters (**`parameters`** block), each parameter being defined by a data type, a name and optionally a default initialization value. For asynchronous modules, the user can define the set of event ports (**`event flows`** block) by specifying their nature (input – **`in`** - or output – **`out`**), their unique identifier, the data type and the name of the variable associated to the port. For synchronous modules, the user has also to define the duration of the execution for the module (**`scheduling`** block) and the set of data ports (**`data flows`** block) the same way as for event ports. Synchronous modules are not allowed to define input event ports since they can spontaneously react only to scheduler module requests. The result of code generation process is the skeleton code of the module, including a complete description of the interface, module main activation loop complete implementation and some empty procedures that the user must fill. For synchronous modules, the user has only to customize the nominal behavior (empty by default) of the module. For asynchronous ones, it has also to customize quick reaction functions (reaction to configuration, consultation and event notification requests).

```
//example module definition file
import "datatype.contract";
asynchronous module MAA{
parameters{
TypeA param = {22.12,6.98,897.34} :0;
}
event flows{
in :0 string controlevent;
out :0 TypeC message_type_notification;
out :1 TypeE received_message;
out :2 string connection_status;
}
}
```

```
//example module definition file
import "datatype.contract";
synchronous module MSC{
parameters{
TypeA param = {2.1,6.98,5.34} :0;
}
scheduling{
C=1ms;
}
data flows{
in :0 TypeA in_coordinates;
out :1 TypeB out_coordinates_commmand;
}
event flows{
out :1 TypeD failing_data;
}
}
```

Example 1: asynchronous (left) and synchronous (right) modules' interfaces

- Scheme files contain predefined composition of modules. Schemes are considered as periodic or event-based ones depending on the key word used (cf. Example 2). Generally, a scheme is a selection of a set of modules (**`modules`** block) parameterized according to scheme own parameters and to the definition of a set of communications allowed between these modules. Periodic schemes are made only of synchronous modules communicating by data flows while event scheme are made only of asynchronous modules communicating by event flows. Periodic schemes also have properties related to real-time scheduling: they define the precedence graph (**`scheduling graph`** keyword) to constraint the order of execution of modules and real-time scheduling properties (**`PERIOD`** and **`CRITICAL_DELAY`** for the execution of a valid sequence of modules). At C source code generation time, scheme information is used to generate configuration code of the scheduler module (realtime properties and precedence graph) but also to generate part of the configuration code of supervisors (communications and parameterizing of modules). So this information is distributed between different software entities.

- Supervisor files define both interface and behavior of supervisor modules (cf. Example 3). Interface of supervisors is made of their name, event ports (**`event flows`** block) and supervision functions prototypes (e.g. function1, top right corner of Example 3). The behavior of a supervisor is defined by its supervision functions: each function implementation defines a set of local variables (**`local`** block) used to store values, and a set of rules (**`rules`** block) . The rules set is arranged according to the priority of evaluation of rules, from first evaluated rule to last evaluated rule. Each rule defines a label that uniquely identifies it (e.g. label **`FIRST`** for first rule), a precondition (first condition between **`[]`**), a sequence of actions ordered with the "**`;`**" operator" and a postcondition (first condition between **`[]`**).

```
import "datatype.contract";
import "smodule1.contract";
import "smodule3.contract";
import "smodule4.contract";
periodic schema schemaCommande1(
       TypeA consigne = {0.0, 0.0, 0.0}){
realtime{
PERIOD=1s;
CRITICAL_DELAY=1s;
}
modules{
MSA ();
MSC (consigne:0);
MSD ();
}
scheduling graph{
MSA -> MSC;
MSC -> MSD;
}
communications{
MSA:0 -> MSC:0 * 1;//data flow communication
MSC:1 -> MSD :2 * 2;//data flow communication
}
}
```

```
import "datatype.contract";
import "amodule1.contract";
import "amodule2.contract";

event schema schemaReceptionMessage(
       TypeA param = {27.12,0.0,0.04}){
modules{
MAA (param :0);
MAB ();
}
communications{
MAA:0 =* MAB:0;//continuous event flow
}
}
```

Example 2: periodic (left) and event (right) schemes

The description language of supervisor modules is certainly the most complex in the environment since it allows to define dynamically set (sub-)architectures, with potentially sophisticated reconfigurations and re-parametrization. But its conciseness allows to define such reconfiguration in a quite easy way. For instance, the activation of a periodic scheme when a rule becomes active (precondition is true) is simply expressed with `activate schema schemaCommande(consigne=temp_consigne)` (cf. Example 3, rule **`FOURTH`**). This will result in: the parameterization of modules contained in the scheme according to the local variable `temp_consigne`, the subscription of synchronous modules data flows according to the communication defined in the scheme and the call to the scheduler module to schedule the scheme. The deactivation of this scheme is never expressed explicitly since it will occurs once the post condition of the rule **`FOURTH`** occurs. The same logic underlies all possible actions, except that some actions cannot be deactivated (parametrizing a module, calling a function, notifying an event) since they are spontaneous and not reversible.

Reactivity of the supervision process itself is mainly achieved by the way events are managed in the supervisor rule interpreter. The evaluation of conditions is done only when necessary, to avoid any busy wait, in other word each time an event received by the supervisor can change its execution context. If a condition is satisfied it immediately results in the activation or deactivation of rules, which means activation or deactivation of actions associated to the rules. All these actions being natively managed (and so completely controlled) in the rules interpreter it avoids most of programming bias that could negatively impact the reactivity (e.g. using mission planning algorithms).

```
action(TypeA consigne){
local{
TypeC temp_message;
string temp_mess_type;
TypeD failure_data;
TypeA temp_consigne = consigne;
}

rules{
FIRST:
      [elapsed=1ms]
            subscribe MAA:2 =* :2;subscribe MAA:0 =* :1;
            activate schema schemaReceptionMessage()
      [MAA:2<connection_status == "disconnected">]

SECOND:
      [(MAA:2<(connection_status=="disconnected" & test("dedede",temp_consigne))>
      and started(FIRST){infinite})]
            subscribe MSC:1=>:3;
            activate schema schemaCommande1(consigne = temp_consigne)
      [MSC:1(failure_data = failure, temp_mess_type = "commutation")]

THIRD:
      [endof(SECOND)]
            notify :1(failure_message_type=temp_mess_type)
      [started(THIRD)]

FOURTH:
      [endof(SECOND)]
            compute calcul("1.24",temp_consigne.k);
            activate schema schemaCommande2(consigne = temp_consigne)
      [endof(FIRST)]

RECEPTION:
      [MAA:0(temp_message=message_received)]
            parameterize MSC(temp_message.value :1);
            activate resttime module MAD
      [started(RECEPTION)]

}
}
```

```
supervisor SUP{

event flows{
in :1 TypeC message_received;
in :2 string connection_status;
out :1 string failure_message;
}

fonction1(TypeA consigne){
...
}

fonction2(TypeB param){
...
}
```

Example 3: behavior (left) and interface (top-right corner) of a supervisor

Real-time supervision mainly relies on pre and post conditions management. Events can be related to i/ absolute time constraints (i.e. a date is reached, using key-work **`elapsed`**), ii/ events generated by modules (including supervisor) or iii/ rules execution constraints (rules has just started, has just stopped or is active since a given amount of time). With last type of events, the user can easily describe sequential and/or concurrent execution of rules, while first ones are mainly used to manage initialization and termination of a supervision function and second ones are used to manage synchronization with executive modules. One important aspect is that complex synchronization of events (using keyword **`and`** in condition) as well as alternatives (using keyword **`or`** in condition) can be put in place together with a management of the persistency of events: the duration of the relevance of an event is specified, and out of this time-interval this event will be forgotten by the supervisor (considered has “not true anymore”). All these characteristics allow for a very precised temporal synchronization of all processes at the supervision level.

Module events are also managed specifically since they require subscription of the supervisor (achieved by **`subscribe`** keyword) and because they are associated to data. These data can be stored into local variables once the event containing this condition has been satisfied, and reused elsewhere (in actions of rules), but they can also be used to test the relevance associated to events (e.g. `MAA:2<connection_status == “disconnected”>` tests if the event produced by the event output port `2` of module named `MAA` relative to a change of the `connection_status` has the string value `disconnected`). These tests are used to verify the relevance of an event not only according to its notification but also considering its value.

IV. Conclusion

The ContrACT model allows describing different kind of control architectures, while taking into account real-time preoccupations. One point we may emphasize is that having a module dedicated to modules scheduling and execution gives the user a lever on which he can act to deal with real-time problems. Several scheduling algorithms can for instance be tested without modifying the control architecture in terms of constituent modules. This fine grain management of real time aspects at architecture description level, considering periodic treatments as well as asynchronous supervision processes is the main originality of this proposal regarding existing architectures. One lack of the proposal is the management of real-time planning, compared to many other proposals, for instance Genom [17] or IDEA [22] architectures.

Thanks to its associated software environment, the ContrACT framework has already been used in several ANR and local projects (underwater and land robots). Two of them, implying different actors from different laboratories and companies, are the ANR Prosit project which aims at developing a tele-echography platform and the ANR Assist project which deals with mobile two-arm manipulators for clinical environment.

As previously mentioned, many works on ContrACT must be carried on. One of on going works is related to this scheduler module; mechanisms to deal with scheduling problems should be added like the possibility to specify how to manage effective task duration and its potential fluctuation (eventually inducing overspending of deadline), how to react to scheduling errors, etc. This will have an impact on the ContrACT language to allow dealing with these aspects directly within supervision rules; to do so the description of interfaces could have to be modified and the set of rules to be completed. Moreover, such mechanisms could also allow to specify at low-level and to automatically, dynamically and efficiently perform commutation of schemes according to real-time considerations like for instance to change the subset of modules to be used for a given objective implying less time-consuming modules (even if embedded algorithm are less precise for example) in case of processor overload. All these possibilities will contribute to the robustness and the efficiency of the architecture.

Other future development will concern for instance the intergration of real-time mission planning since, as we said earlier, this is an important aspect of autnomous robot architectures that is currently missing in our proposal. The general idea is to propose at supervisor level, some mechanisms to build (parts of) the behavior (rule set) of a supervisor according from a set of possible rules and contraints (objective to achieve, spatial, causal and temporal constraints), which requires a dedicated planner (or at least a modification of existing planners). The planner engine will have to execute as a remaining time asynchronous module to avoid any perturbation of real-time loops and to allow its interruption when online replanning is required by a supervisor (e.g. when robot state evolution leads to an invalidation of selected constraints and rules). Validation of supervisor is also another direction to investigate and, concerning this point, we think that a Petri net representation of supervisors' behaviors could allow for some interesting validation processes.

Of course other more “technology-centered” evolutions of the middleware and the IDE are continuously taking place, for instance to ease project development (adding programming/modelling tools), to simplify the management of hardware (reifying hardware units) or to favor code reuse (adding component-based paradygm artifacts to the programming model).

References

[1] D. Brugali, M. Reggiani. “A roadmap to crafting modular and interoprable robotic software systems”. SDIR05, Principles and Practice of Software Development in Robotics, ICRA workshop, April 18, Barcelona, Spain, 2005.
[2] Robotics Engineering Task Force. https://retf.info/
[3] R. Lumia., J. Fiala and A. Wavering, “The NASREM robot control system and testbed”, IEEE Journal of Robotics and Automation, 5(1), pp. 20-26, 1990.
[4] R.A. Brooks, “A robust layered control system for a mobile robot”, IEEE Journal of Robotics and Automation, pp 14-23, March 1986.
[5] J. Rosenblatt, “DAMN: A distributed architecture for mobile navigation,” Journal of Experimental and Theoretical Artificial Intelligence, 9(2/3), pp. 339-360, 1997.
[6] S. Schneider, V. Chen, G. Pardo-Castellote and H. Wang, “ControlShell: A Software architecture for complex electro-mechanical systems”, International Journal of Robotics Research, Special issue on Integrated Architectures for Robot Control and Programming, 1998.
[7] P. Ridao, M. Carreras, J. Batlle, J. Ama, “OCA: A new hybrid control architecture for a low cost AUV”, Proceedings of the Control Application in Marine Systems, 2001.
[8] R.C. Arkin and T.R. Balch. “AURA: principles and practice in review”. Journal of Experimental and Theoretical Artificial Intelligence, 9(2/3), pp. 175-188, april 1997.
[9] R. Alami, R. Chatila, S. Fleury, M. Ghallab et F. Ingrand. “An architecture for autonomy”. Technical report LAAS n°97352, september 1997.
[10] I.A.D. Nesnas, A. Wright, M. Bajracharya, R. Simmons et T. Estlin. “CLARAty and challenges of developing interoperable robotic software”. Technical report, Jet Propulsion Laboratory (California Institute of Technology), NASA Ames Research Center and Carnegie Mellon University, march 2003.
[11] J.S. Albus et al. “4-D/RCS : a reference model architecture for unmanned vehicle systems”, version 2.0. Technical report NISTIR 6910, Intelligent Systems Division, National Institute of Standards and Technology, August 2002.
[12] JAUS Working Group. “Joint Architecture for Unmanned Systems", Reference Architecture Specification, vol. 2, 2004.
[13] O. Rogovchenko, J. Malenfant. “Resource-aware composition of component-based autonomous robot control architectures”. 5th National Conference on Control Architecture of Robots, May 18-19, Douai, France, 2010.
[14] EUROP. The European Robotics Platform. Strategic Research Agenda, May 2006.
[15] M. Barbier – JF. Gabard, D. Vizcaino, O. Bonnet-Torres. “ProCoSA: a software package for autonomous system supervision”. 1th National Conference on Control Architecture of Robots, April 6-7, Montpellier, France, 2006.
[16] I. Nesnas, R. Volpe, T. Estlin, H. Das, R. Petras, and D. Mutz. “Toward developping reusable software components for robotic applications". In International Conference on Intelligent Robots and Systems, October 2001.
[17] S. Fleury, M. Herrb, R. Chatila. “Genom : A tool for the specification and the implementation of operating modules in a distributed robot architecture". In International Conference on Intelligent Robots and Systems (IROS'97), vol.2, pp.842-848, September 1997.
[18] D. Simon, R. Pissard-Gibollet, S. Arias. “ORCCAD, a framework for safe robot control design and implementation”. 1th National Conference on Control Architecture of Robots, April 6-7, Montpellier, France, 2006.
[19] D. Dufourd and A. Dalgalarrondo. “Integrating human / robot interaction into robot control architectures for defense applications”. 1th National Conference on Control Architecture of Robots, April 6-7, Montpellier, France, 2006.
[20] C. Riquier, N. Ricard, C. Rousset. “DES (Data Exchange System), a publish/subscribe architecture for robotics”. 1th National Conference on Control Architecture of Robots, April 6-7, Montpellier, France, 2006.
[21] V. Ambriola, A. Cisternino, D. Colombo, G. Ennas. “Increasing decoupling in a framework for programming robots”. Principles and Practice of Software Development in Robotics, SDIR05, ICRA workshop, April 18, Barcelona, Spain, 2005.
[22] N. Muscettola, G.A. Dorais, C. Fry, R. Levinson, and C. Plaunt. “Idea: Planning at the core of autonomous reactive agents". In Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, 2002.
[23] A.C. Dominguez-Brito, D. Hernandez-Sosa, J. Isern-Gonzalez, J. Cabrera-Gamez. “Another component based programming framework for robotics”. Principles and Practice of Software Development in Robotics, SDIR05, ICRA workshop, April 18, Barcelona, Spain, 2005.
[24] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, A. Oreback. “Towards component-based robotics”. Principles and Practice of Software Development in Robotics, SDIR05, ICRA workshop, April 18, Barcelona, Spain, 2005.
[25] R. Passama, D. Andreu, C. Dony, T. Libourel. “Overview of a new Robot Controller Development Methodology”.1th National Conference on Control Architecture of Robots, April 6-7, Montpellier, France, 2006.
[26] A. El Jalaoui, D. Andreu, B. Jouvencel. “Contextual Management of Tasks and Instrumentation within an AUV control software architecture”. IEEE/RSJ IROS, Beijing, China, October 9-15, 2006.
[27] A. El Jalaoui, D. Andreu, B. Jouvencel. “AUV Control Architecture for Control Management of Embedded Instrumentation”, 4th IFAC Symposium on Mechatronic Systems (Mechatronics'06), Heidelberg, Germany, 12-14 September, 2006.