Execution Control in Autonomous Mobile Robots

Adelardo A.D. Medeiros* **Raja Chatila****

LAAS/CNRS
7, Ave. du Colonel Roche
31077 Toulouse Cedex
France

Abstract. The approach for the design of autonomous robot control architectures we follow aims to providing the robot with deliberation and reactivity capacities. We first recall the basics of this approach and the general architecture we have proposed, which is composed of a decisional level and a functional level. The latter is composed of a set of modules embeding robot functions and controlled in real time by an Executive, the structure and operation of which are presented in detail.

1 Introduction

Autonomous robots are composed of several subsystems that must be interconnected according to a software architecture to ensure the robot will correctly accomplish its tasks. Technical literature reports on a wide spectrum of approaches to the design of robot architectures (like [2] and [10]) and on their analyses and comparisons between them ([5] and [8], for example).

The LAAS architecture [4, 1], adopted in this work and represented in fig. 1, has been designed to endow a robot with an autonomous and intelligent behaviour in achieving its goals, combining deliberation and reactivity [3]. It advocates the system decomposition into two levels:

- The *decisional level*, which centralises the capacities of planning and decision-making. This level may include one or more layers, usually two, one for mission-level planning and supervision, and one for task-level planning and supervision. This last level is based on a procedural reasoning system that selects the more appropriate manner to achieve a task according to the environment context and robot status. The actions composing the tasks are transmitted for execution to the functional level.
- The *functional level*, which includes the operational functionalities of the robot, like processing functions and control loops. It is composed of an Executive controlling a set of modules that embed the functions for sensing and acting.

The general characteristics and properties of the LAAS architecture are well described in other papers [4, 1], and article [9] presents a recent implementation of the task layer of the decisional level using PRS, a well-suited tool for real-time systems based on procedural reasoning. This paper, in its turn, presents a general view of the functional level and a more detailed exposition of the Executive unit, mainly focused on the real-time execution control at the functional level.

2 The Functional Level

The functional level is composed of a set of elementary robot actions implementing task-oriented servo-loops as well as a set of robot primitive functions (motion planner, perception, etc.), grouped into *modules* and under the control of an *Executive*. All the control flow between the decisional and functional levels passes through the Executive. But, even if the Executive centralises some of its competences, the functional level is not completely centrally arranged: a module may read data exported by other modules and send some requests directly to another one, in both cases without passing through the Executive. It can also export its data, making them directly available to the other modules and even to the decisional level, if necessary.

* E-mail: `adelardo@leca.ufrn.br`. On leave from the *Departamento de Engenharia Elétrica – Universidade Federal do Rio Grande do Norte* – 59072-970, Natal, RN, Brazil. Supported by UFRN and the Brazilian agency CAPES.
** E-mail: `raja@laas.fr`.

2.1 The Modules

The modules (well described in [6]) should react to external requests from a client and, after processing, send back a report to this client, using a client/server communication protocol (with requests and replies). If necessary, while executing a function a module can dispatch requests to other ones. A shared memory, called *poster*, is used to export data by a module that may be read by others.

A module usually encapsulates functions that manipulate the same data or resources. In case of a new request incompatible with a previous one under execution, the module interrupts the previous request, starts the new one and reports the interruption in the final reply of the interrupted request.

This policy of always serving the last arrived request is a consequence of concentrating all the decision-making in the decisional level. As the functional level does not know the robot's global tasks, the conflicts are solved at a higher level which has a more global representation of the context. If the adopted priority policy decides not to start a function, the corresponding request is not sent to the functional level; but if, on the contrary, the new request is considered to have priority, the functional level will have to serve it. Hence the functional level is fully reactive.

The organisation of the modules is not fixed. Their interactions depend on the task being executed and the perceived environment. One important characteristic of this organisation is its flexibility. The robot behaviour is not fixed by any "association of behaviours": all the functions are available to the decisional level, that decides, by means of an event driven and goal driven procedure selection, which one to use according to the robot global task.

The functions The functions of the modules must implement the atomic actions[3] the robot can execute. The conflicts between functions are analysed and resolved when the request to activate a new function arrives: the Executive and the controller of the module know the resources the new function needs and verify if one of the active functions is using the same resource. As the allocation of the resources to a function is made *a priori* and stays valid during its whole execution[4], the functions must respect some rules to optimise the utilisation of the resources:

1. If a function uses a non-sharable resource, it must use the resource during all (or a large part) of its execution time. If this is not the case, the function must be subdivided into parts respecting this rule; otherwise, either the function will be unnecessarily interrupted or will interrupt other functions that could continue executing.
2. If a function sends requests to activate other functions that use non-sharable resources, the set of called functions must be known *a priori* and immutable, to allow the Executive and the controller of the module to take all the functions that will be activated into account when deciding about the conflicts the new function can create. The resources the called functions use are considered as if they were used by the calling function and therefore the first rule must be respected concerning these resources.

The function reports The functions emit logical *reports* about their execution: these reports are sent by the Executive to the higher levels. Each function must emit two kinds of report: the first one, the *execution status*, concerns the final state of the function execution; the second one is the *result report*, and presents in a abstract way the output of the function.

The execution status assumes only two values, `OK` or `ERROR`, according to the normal or abnormal execution of the function. We consider an abnormal execution the cases where a non-nominal behaviour of a robot subsystem prevents the conclusion of the function execution.

The data that are results of an execution are usually stored in posters, where they can be accessed and used by other functions. But the decisional level may need some information about these results in order to take its decisions. This information, in a abstract format, must be furnished in the result report of the specific functions. Algorithmic errors, that are not considered as execution errors if their occurrence is more or less foreseeable when executing this function, are also reported. Hence, the sense and the possible values of the result report depend on the concerned function.

[3] These actions are not necessarily simple or elementary: they are considered to be atomic because of their indivisibility

[4] The execution can nevertheless be interrupted to liberate a resource

2.2 The Executive

Some of the Executive's missions are:

- Managing of conflicts between modules.
- Monitoring the execution of the functions.
- Accomplishing some reflex actions.
- Maintaining the knowledge about the actual state of the modules.

The majority of the possible conflicts between functions involve functions of the same module: in this case, the module itself can manage the situation. In some cases, however, the incompatible functions reside in different modules, and the Executive itself must interrupt the offending one.

As explained in [6], the activation of a function involves the exchange of a request and some replies between the module and its client. But the decisional level knows neither the division of the functional level in modules nor the communication protocol, and the Executive will act as the client of all the modules in its place. From the point of view of the decisional level, all works as if a request was sent to the Executive and, some time later, the Executive answers with a result report.

Sometimes we expect the robot to react quickly when some events happen (for example, in emergency situations). As the response time of the decisional level is not bounded, we can profit from the temporal properties of the Executive (limited and known execution time) to implement these reactions at this level of the architecture. The actions accomplished this way, however, must be purely reactive and not deliberated, because the Executive does not know the global task of the robot.

Finally, the decisional level may need to know the current utilisation of the robot resources. The Executive, using the knowledge about the intrinsic behaviour of the modules, maintains a logical description of the operating state of each one of them.

3 Architecture and Operation of the Executive

The Executive, represented in fig. 2, is composed of five main elements:

Function description list: this static list contains a description about each one of the functions the decisional level can ask the Executive to start: the module it belongs to, input parameters, etc.

Function execution list: stores the current execution state of the active functions.

Request monitor: monitors the arrival of requests from the decisional level and sends the appropriate requests to the modules.

Reply monitor: monitors the arrival of replies from the modules and sends the appropriate result reports to the decisional level.

Automaton: represents the state of all the robot modules and decides about starting the functions.

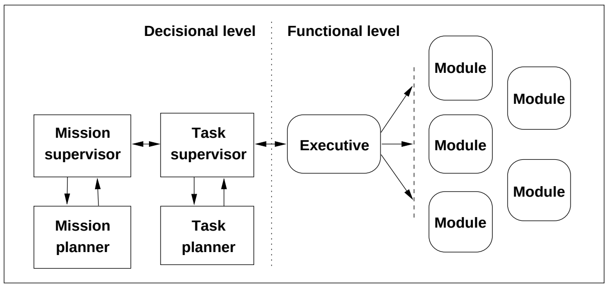

Fig. 1.: A Global LAAS architecture

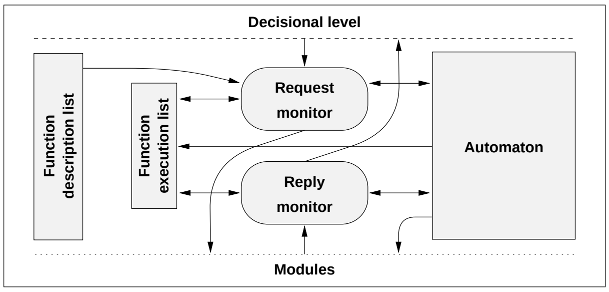

Fig. 2.: Architecture of the Executive

When a request arrives from the decisional level, the request monitor wakes up and the function description list is examined to determine if it corresponds to a recognisable one. If the function is not recognised, the Executive returns an execution status of `ERROR`. If it is recognised, the new function is included in the execution list, its execution state is marked `INIT` and the automaton is called before starting its execution.

If the automaton delivers the authorisation to start the function, the request monitor changes the state of the function in the execution list to `EXEC` and sends the appropriate request to the modules. This ends the execution cycle of the request monitor: it will be asynchronously reactivated if a new request arrives.

A function of a module can have more than one description in the function description list: it is the case, for example, when a function can be called with different input parameters or when the policy of resolving conflicts concerning this function is not fixed (see section 3.2).

The reply monitor executes a cycle when a reply arrives from a module. In the case of an intermediary reply, only the state of the corresponding function is actualised in the execution list. When the reply indicates the termination of an execution, and if the evolution of this function must be followed by the automaton, it is called to take the event into account. After the execution of the automaton, the reply monitor sends the execution status and the result report to the decisional level, eliminates the function from the execution list and puts itself in a waiting state, on the watch for new replies.

3.1 The automaton

The automaton describing the module states is implemented as a function with some input variables (corresponding to the signals able to change the automaton state) and output variables (their set defining the automaton state). Each execution cycle of this function represents the transition of an automaton state to another one.

The automaton has two explicit inputs:

- `function` – the function for which an event is being reported.
- `operation` – the kind of event: `START`, `END` or `ERROR`.

The automaton outputs are the state of the modules and a `status` variable. The module state mainly describes which function (if any) is using the non-sharable resource associated with this module. `status` indicates the reaction the automaton recommends to the event. As the previous state of an automaton is necessary to determine its next state, the states of the modules will be used as implicit inputs when executing the next cycle.

When the automaton receives a request to take account of a function start (`operation = START`), `status` can assume the following values:

- `EXEC` – starting the function is authorised.
- `IGNORE` – starting the function is authorised and the automaton does not need to be (and should not be) informed about its evolution; concerns the functions that do not use non-sharable resources.
- `REFUSE` – the function cannot be started.
- `WAIT` – starting the function is temporarily impossible and the request must be resubmitted.
- `ERROR` – unknown function.
- `OK` – the concerned function is internal to the Executive; all the necessary processing was already done during the execution cycle of the automaton (see the `restart` function later).

If the automaton is being informed of a normal or an abnormal (`operation = END` or `ERROR`) execution ending, the possible `status` values are:

- `OK` – the termination was taken into account.
- `ERROR` – according to the automaton, this function is not in execution.

Some particular rules were adopted to define the behaviour of the automaton when facing execution errors. All abnormal endings of execution place the corresponding module in a `ERROR` state. When a module is in a `ERROR` state, the automaton refuses all requests to activate any of its functions. A `restart` request restores the module to an `IDLE` state. This policy imposes on the decisional level to actively react to exceptional events.

3.2 Resolution of Conflicts

Except for the `ERROR` state, normally any incoming request is accepted by the automaton (because of the reactive paradigm of the functional level). But when conflicting functions exist, we can easily imagine situations where, according to the general context of the task being executed, the decisional level expects the functional level to exhibit a different behaviour. Let us exemplify with the example of a request to take a image with the laser telemeter that arrives when the robot is moving. Three possibilities exist:

1. The image must be accurate and is urgent: the robot will be immobilised.
2. The image must be accurate but is not urgent: we can wait until the end of a potential movement.
3. The image can be inaccurate: the telemeter can be used even if the robot is moving.

These behaviours can be implemented by three distinct entries in the function description list, that we will call `TAKEIMAGESTOP`, `TAKEIMAGEWAIT` and `TAKEIMAGE`. All three requests, when they arrive from the decisional level and if they are accepted, occasion the sending of a `TAKEIMAGE` request to the `LASER` module, but the decisional level sees them as three independent functions and can chose one of them according to the general context of the robot's global task. The example in section 5 shows the behaviour of the Executive when facing a `TAKEIMAGESTOP` request.

4 The KHEOPS System

Coding the transitions between states of the automaton requires representing a large amount of the system designer's knowledge. Then, a rule-based language seems appropriated to implement this subsystem. But the position occupied by the automaton in the control architecture imposes some constraints on its temporal characteristics, like limited execution time, that are not usually respected by rule-based developing systems. One solution to this problem is the utilisation of the Kheops system [7].

The Kheops system is a development environment for rule-based systems devoted to real-time applications. Its knowledge representation relies on multi-valued propositional logic and monotonic reasoning. Kheops tackles the usually poor temporal properties of rule-based declarative programming by pre-compiling the knowledge. It rewrites a set of declarative rules into a deterministic network, the parsing through of which gives an upper-bound on the response of the system.

A knowledge representation in Kheops relies on a finite set of propositional input and output variables (or *attributes*) which take their values over finite domains and on a knowledge base which defines a consistent mapping from the input space to the output space. The compiler produces a decision tree (with an equivalent C code source), the nodes of which are tests on the input attributes values, and the leaves of which characterise the resulting situation (completeness, incompleteness or inconsistency). As the compilation is done during the development phase, the knowledge base can be progressively refined until we are assured the output attributes are completely and unequivocally determined for each combination of values of the input attributes.

Figure 3 shows two examples of Kheops rules of the automaton: if a `TAKEIMAGESTOP` request arrives when the robot is executing a trajectory, the first one of them will probably be triggered during the transition of the automaton immediately after the arrival.

```
TakeimagestopWhenLocostateExecutingTraj:
  function == "TAKEIMAGESTOP",
  operation == "START",
  @laserstate == "IDLE",
  @locostate == "EXECUTINGTRAJ"
==>
  do sendinterrupt("EXECTRAJ"),
  restrict locostate == "EXECUTINGTRAJ",
  restrict laserstate == "IDLE",
  restrict status == "WAIT"
```

```
StopWhenLocostateExecutingTraj:
  function == "STOP",
  operation == "START",
  @locostate == "EXECUTINGTRAJ"
==>
  do changestate("EXECTRAJ","ZOMBIE"),
  restrict locostate == "STOPPING",
  restrict status == "EXEC"
```

Fig. 3.: Two rules of the automaton written with KHEOPS

5 An example of function execution

We present in fig. 4 and 5 the activities of the Executive and the evolution of the execution list when a request to execute a trajectory (`EXECTRAJ`) arrives and, while this function is executing, a `TAKEIMAGESTOP` request, incompatible with the previous one, arrives.

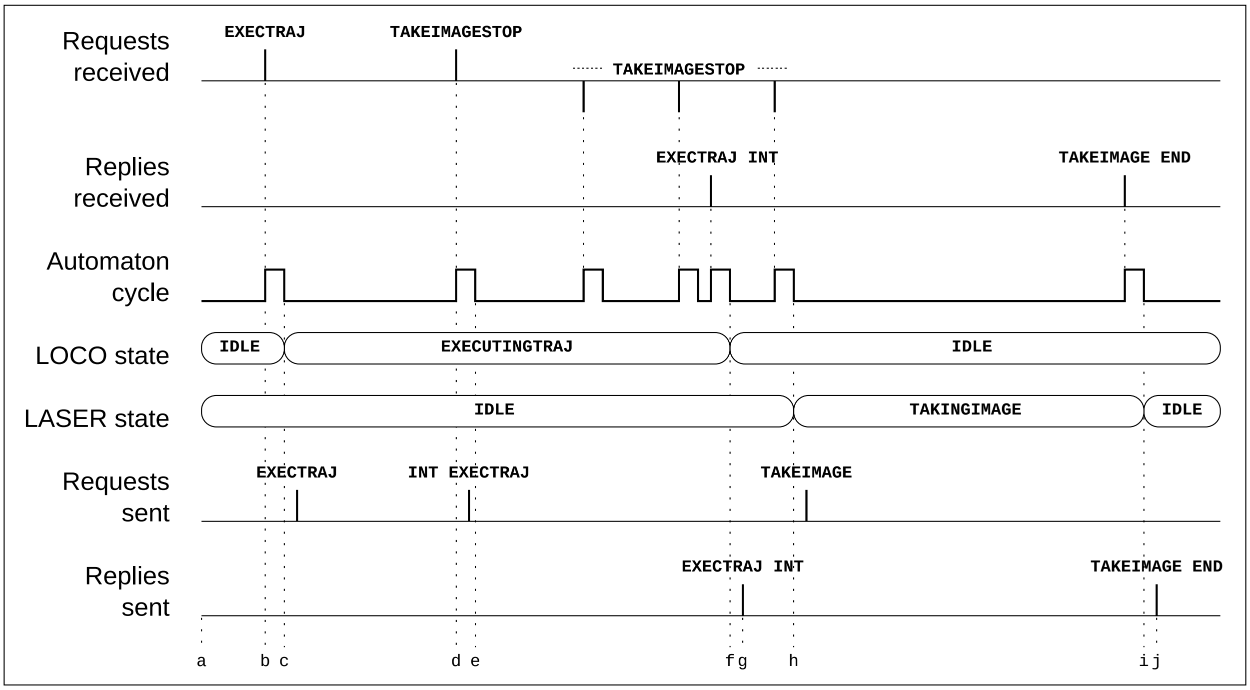

Fig. 4.: Temporal evolution of the system

The first line in fig. 4 represents the events corresponding to arrival of requests to the Executive: those above the line arriving from the decisional level and those below the line being internally generated by the Executive. The second line shows the arrival of replies from the modules, and the third indicates the instants of activity of the automaton. The fourth and fifth lines display the state of two of the robot modules, `LOCO` and `LASER` that control respectively robot motion and image acquisition by a laser range finder. The sixth and seventh lines show respectively the requests sent by the Executive to the modules and the replies sent by the Executive to the decisional level.

Function	A	B	C	D	E	F	G	H	I	J
`EXECTRAJ`		init	exec	exec	exec	end				
`TAKEIMAGESTOP`				init	wait	wait	wait	exec	end	

Fig. 5.: Evolution of the execution list

At instant A, when our example begins, the execution list is empty and the `LOCO` and `LASER` module are in an `IDLE` state. When the `EXECTRAJ` request arrives (B), the request monitor is activated: the function is included in the execution list with an `INIT` state and the automaton is called to decide about the starting of the function. The automaton, after executing a cycle (C), gives the authorisation to start the function (making its output variable `STATUS = EXEC`) and consequently changes the `LOCO` state to `EXECUTINGTRAJ`. The request monitor continues its cycle and, taking into account the response of the automaton, sends an `EXECTRAJ` request to the `LOCO` module and changes the state of the `EXECTRAJ` function in the execution list to `EXEC`[5].

While `EXECTRAJ` is executing, a `TAKEIMAGESTOP` request arrives (D) and is included in the execution list with a `INIT` state. During the execution of the resulting cycle of the automaton (D-E), the first rule in fig. 3 is triggered and a control request to interrupt the `EXECTRAJ` function is sent to the `LOCO` module. At the end of the processing (E), the automaton does not change either the `LOCO` state (because `EXECTRAJ` is still executing until its interruption) or the `LASER` state (because `TAKEIMAGESTOP` is not yet executing) and responds

[5] After the reception of an acknowledge intermediate reply from the module stating the request is taken into account, not represented in the schema

with `STATUS = WAIT`, indicating to the request monitor that the execution state of the concerned function (`TAKEIMAGESTOP`) must be marked `WAIT`.

When the final reply of the `EXECTRAJ` function arrives, giving notice of its interruption, the automaton is called to take into account this event and changes the `LOCO` state to `IDLE` (F). The reply monitor changes the execution state of `EXECTRAJ` to `END` and sends the appropriate reply to the decisional level. When this reply is received by the decisional level (G), the reply monitor excludes `EXECTRAJ` from the execution list.

The controller of the `LOCO` module needs some time to immobilise the robot after the request to interrupt `EXECTRAJ` arrives. During this time, `TAKEIMAGESTOP` remains in the execution list with a `WAIT` execution state and will be periodically resubmitted to the automaton. While the `LOCO` state is not `IDLE`, the automaton does not authorise the execution of the function[6]. The first time the request is resubmitted after the end of `STOP`, the automaton will decide to start the function, changing the `LASER` state to `TAKINGIMAGE` (H). The `LASER` state will return to `IDLE` after the end of its execution is processed by the automaton (I) and `TAKINGIMAGE` will be excluded from the execution list after the reply that informs this event is received by the decisional level (J).

6 Conclusion

The general concepts of the LAAS architecture have been validated by several experiments and projects: the *Hilare* robot, the EDEN experiment and the MARTHA (Multiple Autonomous Robots for Transport and Handling Applications) European ESPRIT project. In all these projects the idea of an Executive was present, but the implementation did not emphasise its separation from other units. The inclusion of an Executive developped according to the ideas presented in this article adds some important features to the architecture:

– A better distinction between the reactive and deliberative parts of the architecture: roughly, the first one manipulates data to accomplish atomic actions while the second one plans sequences of actions based on simpler abstract data. The temporal requirements concerning the two levels of the architecture are different. Having a well defined border between them permits the utilisation of the better tools to implement each one.
– The execution control can de bone with a guaranteed bounded execution time adapted to the dynamics of the environment. The decisional level - because it is compode by a planner-supervisor pair at each layer has also a bounded, but longer, reaction time (which is the reaction time of the supervisor).
– The functions implemented in the experimental robots are bound to evolve and to be modified. The Executive creates a relatively stable interface between the decisional and functional levels and allows, for example, that changes in the modules do not affect the way the task level sees these modules through the Executive.

The concepts presented here are being incorporated into the control system of the robot *Lama*, an outdoor robot with six conical wheels.

References

1. R. Alami, R. Chatila, and B. Espiau. Designing an intelligent control architecture for autonomous robots. In *ICAR - International Conference on Advanced Robotics*, pages 435–440, Tokyo, Japan, Nov. 1993.
2. R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, Mar. 1986.
3. R. Chatila. Deliberation and reactivity in autonomous mobile robots. *Robotics and Autonomous Systems*, 16:197–211, Dec. 1995.
4. R. Chatila, R. Alami, B. Degallaix, and H. Laruelle. Integrated planning and execution of autonomous robot actions. In *IEEE International Conference on Robotics and Automation*, volume 3, pages 2689–2695, Nice, France, May 1992.
5. R. E. Fayek, R. Liscano, and G. M. Karam. A system architecture for a mobile robot based on activities and a blackboard control unit. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 267–274, Atlanta, Georgia, USA, May 1993.

[6] Each function has an intrinsic associated timeout: if it is reached and the execution is not authorised, the function is considered as failed and a `ERROR` reply is sent to the decisional level.

6. S. Fleury, M. Herrb, and R. Chati la. Design of a modular architecture for autonomous robots. In *IEEE International Conference on Robotics and Automation*, volume 4, pages 3508–3513, San Diego, California, USA, May 1994.
7. M. Ghallab and H. Philippe. A compiler for real-time knowledge-based systems. In *IEEE International Workshop on Artificial Intelligence for Industrial Applications*, Hitachi City, Japan, May 1988.
8. J.-M. Hasemann. Robot control architectures – application requirements, approaches, and technologies. In *XIV Intelligent Robots and Computer Vision: Algorithms, Techniques, Active Vision, Materials Handling (part of the SPIE's Photonics East Symposium)*, Philadelphia, Pennsylvania, USA, Oct. 1995.
9. F. F. Ingrand, R. Chatila, R. Alami, and F. Robert. Embedded control of autonomous robots using procedural reasoning. In *ICAR - International Conference on Advanced Robotics*, volume 2, pages 855–861, Sant Feliu de Guixols, Catalonia, Spain, Sept. 1995.
10. R. G. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, Feb. 1994.

This article was processed using the TEX macro package with SIRS96 style