Specification and Validation of a Control Architecture for Autonomous Mobile Robots

Adelardo A.D. de Medeiros* Raja Chatila Sara Fleury
LAAS/CNRS – 7, Ave. du Colonel Roche – 31077 Toulouse Cedex France
adelardo@leca.ufrn.br raja@laas.fr sara@laas.fr

Abstract

We describe in this paper the specification of a software control architecture for autonomous mobile robots. The architecture, designed to provide the robot (in a task-dependent context) with the capacity to react to events but also to intelligently anticipate the future and plan its actions, is based on the decomposition of the robot system into a functional and a decisional level. The article is mainly focused on some aspects of the organisation and of the operation of the system such as execution control, inter-levels communication, reactivity. An important aspect that is developed is the possibility to prove some temporal and logical properties of parts of the system.

1 Introduction

The tasks a programmable and autonomous robot shall carry out are multiple and not completely known *a priori*. In this case, the design of its control architecture is an important aspect to ensure the robot will be able to fulfill its objectives. The architecture must include both some decision-making and real-time execution processes.

Technical literature reports on a wide spectrum of approaches to the design of robots architectures (like [3, 16, 2]) and on analysis and comparisons of them ([6, 10, 17] for example). We will not overview here all proposed architectures, but they generally emphasis either a reactive part or a highly centralised structure.

The LAAS architecture [1] has been designed to endow a robot with an autonomous and intelligent behaviour in achieving its goals, combining deliberation and reactivity [4]. In order to achieve both anticipation capacities and real time behaviour, the system is decomposed into two levels:

- a *functional level*, which includes processing functions and control loops; and
- a *decisional level*, possibly organised in layers, embedding the capacities of planning and decision-making. Each of the decisional layers is built with the supervisor-planner paradigm [1]. Since planning requires an amount of time usually longer than the dynamics imposed by the occurrence of an event, this paradigm consists in guaranteeing a bounded reaction time for a first response. The layers are composed of *planners*, which produces the sequence of actions necessary to achieve a given task or to reach a given objective, and *supervisors*, which actually interact with the inferior levels, control the execution of the plans and react to incoming events.

One of the most generic implementations of this principle organises the global system architecture into three levels, representing two decisional layers and a functional level (fig. 1). The decomposition of the decisional level into two layers is heuristic, in order to improve performance and take better account of the interactions with the environment and the limitations of the embarked computational material.

The mission level uses a temporal planner. It will not be treated in this paper, but in [5] is presented a mission level based on the temporal planning system IxTeT (Indexed Time Table), which complies with the general properties discussed previously.

The task level receives tasks from the mission level, transforms them into sequences of actions and supervises their execution. As it is based on a selection of predefined procedures for each task (no general planners are used), its response time can be bound.

The functional level includes the basic functionalities of the robot. It is composed of an executive and of a set of modules embedding the functions for sensing and acting. The basic reactivity is also implemented at this level.

This paper presents a general view of the last two levels of this architecture: the functional level is described in section 2 and some aspects of the task level in section 3. Validation and provability properties are discussed for both levels. For the functional level, they concern reaction time bounds as well as completeness and inconsistency detection. For the task level, they mainly concern the existence of deadlocks and loops, and reaction time bounds. The presented architectural concepts were implemented and intensively used in several projects and experiments at LAAS for multi-robot cooperation (MARTHA), indoor navigation (HILARE), or outdoor navigation (EDEN). For lack of space, we will not develop these projects here, but rather encourage reading published papers such as [12] and [14].

*Usually with the *Departamento de Engenharia Elétrica – Universidade Federal do Rio Grande do Norte (UFRN) – 59072-970, Natal, RN, Brazil.* Supported by UFRN and the Brazilian agency CAPES

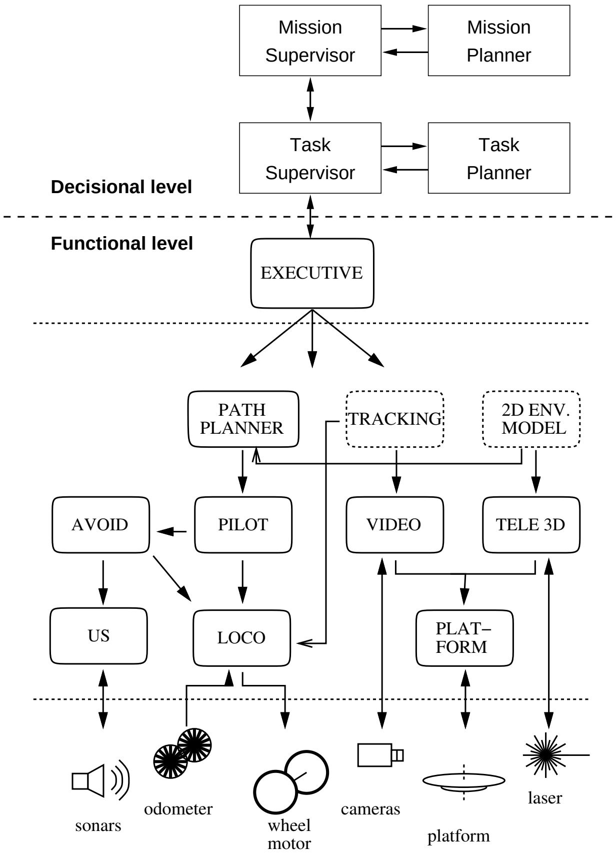

Figure 1: A Global LAAS architecture

2 The Functional Level

A functional level can be seen as a library of functions with real-time abilities. They permit the elementary robot actions including task-oriented servo-loops as well as a set of robot primitive functions (motion planner, perception, etc.), grouped in *modules* and under the control of an *executive*. The example of functional level given in fig. 1 represents the organisation of the experimental indoor mobile robot HILARE 2, equipped with driving and odometer wheels, a rotating platform supporting colour cameras and a 3D scanning laser range finder and ultrasound sensors.

Even if the executive centralises some of its competences (like the control exchanges with the decisional level), the functional level is **not** centrally arranged: a module may send requests directly to another one, only informing the executive, and read data exported by other modules.

2.1 The Modules

The modules (well described in [7]) 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 others. A shared memory, called *poster*, is used to export data by a module. It can be only written by the owner module and read by any other one.

A module usually encapsulates similar functions or functions that manipulate the same data or resources. All the functions of a given module are pre-defined at the system design stage. Some functions can have several instances running in parallel, but certain other ones can not be executed simultaneously, mainly because they use the same non-sharable resource. In case of a new request incompatible with a previous one, 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 must not be understood as a general priority policy of the architecture, but only as a consequence of concentrating all the decision-making in the decisional level. As the functional level does not know the robot global tasks, the conflicts should be solved at a higher level. 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 serve it.

The organisation of the modules is not fixed. Their interactions are dynamic and depend on the task being executed and the perceived environment. We will give some examples of the HILARE 2 system.

2.1.1 Example of Module Interactions

The `LOCO` module achieves the low-level feedback control of the wheel motors. It possesses a unique main function, which receives as input parameter a poster where to read the desired position reference. Once this function is started, it reads at a fixed frequency the reference, and ensures the servoing according to the position as computed from odometry. This function never ends by itself and, obviously, no more than one instance of this function can be active simultaneously.

The `PILOT` module generates some primitive trajectories (like straight lines, circles, clothoids, etc.) and writes the corresponding series of position references to a poster. For moving the robot according to one of these trajectories, the `PILOT` module offers a function that accomplish the following steps:

1. starts the appropriate `PILOT` function for generating the trajectory.
2. activates the `LOCO` tracking function, giving as input parameter the poster of the `PILOT` function.
3. at the end of the trajectory interrupts the `LOCO` function.

Some basic reactivity is also implemented by interconnection of modules. The `AVOID` module, for example, possesses a filter function that, at a given frequency, reads a position of a poster (for instance the `PILOT`'s) and compares with the information from the ultrasound sensors: if the desired position is not perceived as too near to an obstacle, it is simply copied to the AVOID poster; otherwise, it starts an avoidance process based on a potential field using the ultrasound data:

1. begins the processing of the PILOT poster, writing the filtered positions in the AVOID poster.
2. activates the LOCO tracking function, giving as input parameter the poster of the AVOID filter.
3. at the end of the trajectory interrupts the LOCO function.

One important feature of this organisation is its *flexibility*. The robot behaviour is not fixed by any "association of behaviours": all the functions are available and can be combined to achieve a given task or a desired activity.

2.1.2 The Function Reports

While the modules process numbers, images, matrices and other complex data, the decisional level needs a more abstract view of the world and of the robot state. For this, the functions emit two logical *reports* about their execution to the requesting module or the executive: the *execution status* concerns the final state of the function execution; and the *result report* represents the output of the function.

The execution status assumes an OK or ERROR value according to the normal or abnormal execution of the function. An abnormal execution is reported when a non-nominal behaviour of a subsystem prevents the conclusion of the function execution: failures signaled by the operating system (this type of error is useful mainly during development) or related to the function itself. An algorithmic error is not enough to characterise an abnormal execution if its occurrence is more or less foreseeable when executing this function. For example, we must report a normal execution even if a target is not recognised when processing an image.

The low-level results of executing a function are usually stored in posters, where they can be used by others functions. But the decisional level may need some information about these results in order to take its decisions. These informations, in a abstract format, must be furnished in the result report of the specific functions. Algorithmic errors not considered as execution errors are also reported. Hence, the semantics and the possible values of the result report depend on the concerned function.

2.2 The Executive

The executive acts like an interface between the decisional level and the modules. Its main role is:

- dispatching the requests, monitoring the execution of the functions and receiving the replies.
- Maintaining the knowledge about the actual state of the modules.

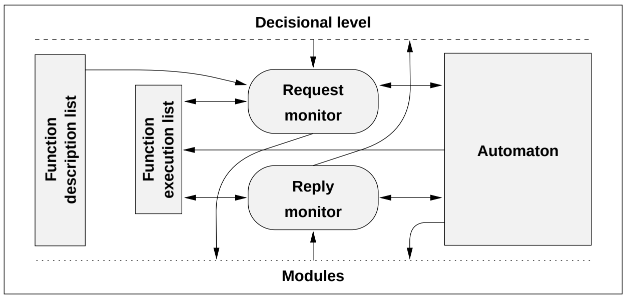

Figure 2: The architecture of the executive

- Managing conflicts between modules.

The majority of possible conflicts between functions involves functions of the same module: in this case, the module controller 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.

The activation of a function involves the exchange of a request and some replies between the module and its client [7]. The decisional level does not know neither the division of the functional level in modules nor the communication protocol: then, the executive will act as the client of 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.

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

2.2.1 Architecture and Operation

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

Function description list: this static list contains descriptions about the functions in the functional level: embedding module, 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 replies arriving 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.

When a request arrives from the decisional level, the request monitor executes a cycle. Firstly, the function description list is examined to determinate if the request corresponds to a recognisable one. A single function may have more than one description (that are perceived as different functions by the decisional level): it is the case, for example, when a function can be called with different input parameters, e.g., a tracking function taking as input the outputs of different perception functions.

If the function is not recognised, the executive returns an execution status of `ERROR` (this may happen essentially during development). If it is recognised, the new function must be included in the execution list, its execution state is marked `WAITING` and the automaton is called before starting its execution.

The automaton describing the module behaviour is implemented as a function where the output variables define the automaton state and the input variables represente the signals able to change this 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 concerned by the event,
- `operation` – the event: `START`, `END` or `ERROR`,

and one implicit input: its previous state.

The automaton outputs, updated after each execution cycle, 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 processing the automaton recommends to the function.

When the automaton receives a `START` operation, `status` can assume the following values:

- `EXEC` – starting the function is authorised and the end of its execution must be reported.
- `IGNORE` – starting the function is authorised but the automaton does not need to 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, but the request can be resubmitted latter.
- `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 gives the authorisation to start the function, the request monitor changes the state of the function in the execution list to `EXECUTING` and sends the appropriate request (defined in the function description list) to the module. This ends the execution cycle of the request monitor; it will be asynchronously reactivated when a new request arrives.

The reply monitor executes a cycle when a reply arrives from a module. If it is an intermediate reply, the state of the corresponding function is updated 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, the automaton is called to take into account the event.

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

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

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.

This description of the executive's behaviour covers mainly its nominal functioning. Some reactions to errors and exceptional situations are not presented here, because of concision reasons, but the general policy is to send an execution status of `ERROR` to the decisional level each time an abnormal behaviour arrives. In case of a severe failure, not attributable to an specific function, the executive places itself in a exception state, demanding a specific intervention to resume.

2.2.2 The Automaton

Some rules were adopted to define the behaviour of the automaton. An abnormal end of execution places the concerned module in a `ERROR` state. When a module is in a `ERROR` state, the automaton refuses all request to activate any of its functions: only a `restart` request restores it to an `IDLE` state. This policy imposes on the decisional level to actively react to these events.

Except for the `ERROR` state, normally all arriving request is accepted by the automaton (because of the reactive paradigm of the functional level). But, when conflicting functions exist, the decisional level can expect the functional level to exhibit different behaviour according to the general context of the task being executed. Let us examin the classical example of a request to take an image for localising the robot that arrives when the robot is moving. Three possibilities exist[1]:

1. The image must be accurate and is urgent: the motion functions shall be interrupted.
2. The image must be accurate but is not urgent: we can wait the end of the motion functions.
3. The image can be inaccurate: the camera can be used while the robot is moving.

These behaviours can be implemented by distinct requests in the function description list, that we will denote `takeimage1`, `takeimage2` and `takeimage3`:

1. `takeimage1` requests are always accepted, and the active motion functions are interrupted.

[1]When the conflicting functions use the same non-sharable resource, only the first two possibilities exist. An hypothetical fourth behaviour of refusing the request is unacceptable because it violates the reactive paradigm of the functional level.

2. If a `takeimage2` request arrives and no motion function is executing, it is accepted. On the contrary, the automaton emits `status = WAIT` and the request remains in the execution list in a `WAITING` state; periodically it will be resubmitted, until the conflicting function ends.

3. The `takeimage3` requests are accepted independently of any motion function in execution.

The decisional level sees these requests as independent functions and can chose one of them according to the general context of the global task to accomplish.

When a function conflicts with another one of the same module, the executive does not need to interrupt it: the module controller itself will interrupt the previous one when the new requests arrives. If the conflicting functions are in different modules, however, the executive must explicitly interrupt the offending one.

Coding the transitions of the automaton requires representing knowledge of the system designer, that might eveolve during development. Then, a rule-based language seems appropriated to implement this subsystem. But the control architecture imposes some constraints on the automaton's characteristics, such an limited execution time, that are not usually respected by rule-based developing systems. One possible solution is the utilisation of Kheops[9].

2.2.3 The KHEOPS system

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 usual 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 traversal of which gives an upper-bound on the time 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, the nodes of which are tests on the input attributes values, and the leaves of which characterise the resulting situation (complete determination of the output attributes, incompleteness or inconsistency).

Implementing the automaton with Kheops permits to ensure some important properties of this subsystem:

- The execution cycle of the automaton will have a finite (and *a priori* measurable) execution time, because of the characteristics of the decision tree.
- As the compilation is made during the developing phase, the knowledge base can be refined until we are assured the output attributes are completely and unequivocally determined to all possible values of the input attributes. This assures that the knowledge base provides one and only one evolution of the automaton state for each possible combination of the signals able to change this state.
- The compilation converts the knowledge base into an equivalent C code source that can be linked to the other subsystems. No trace of Kheops (like inference machines, usually necessary when executing rule-based programs) will be present in the executable version of the automaton.

```
Takeimage1WhenLocostateCircling:
   function == "TAKEIMAGE1",
   operation == "START",
   @videostate == "IDLE",
   @locostate == "CIRCLING"
==>
   do sendrequest("LOCO","STOP"),
   do setexecstate("LOCOCIRCLE","ZOMBI"),
   restrict locostate == "IDLE",
   restrict videostate == "IMAGETAKING",
   restrict status == "EXEC"
```

Figure 3: An automaton rule

Fig. 3 shows an example of a Kheops rule representing the transition of the automaton when a `takeimage1` request arrives and the HILARE's locomotion module is executing a circular trajectory.

3 The Task Level

The task level receives abstract tasks (like `GOTO goal`) from the mission level: the task planner transforms them into sequences of actions the functional level knows and the task supervisor watchs their execution. The planning is based on a selection of predefined procedures (no general planners are used): each procedure contains its applicability conditions and the planner selects the must suitable one for the actual conditions. The procedures include modalities that are chosen by the supervisor during execution time, according to the evolution signalled by the functional level.

Implementing the task planner implies that a certain knowledge must be represented, whereas the task supervisor needs essentially a good procedural representation. We could implement these two components independently, but the high level of interaction between them would require the development of a powerful communication protocol and perhaps introduce performance limitations. A better solution is to use PRS (Procedural Reasoning System) [8], which provides a suitable framework for goal-driven as well as situation-driven deliberation processes.

3.1 The PRS system

We use a C implementation of PRS under Unix, called C-PRS [11]. A C-PRS kernel consists of:

1. a library of plans (or procedures), called *knowledge areas* (KAs), which describe sequences of ac-

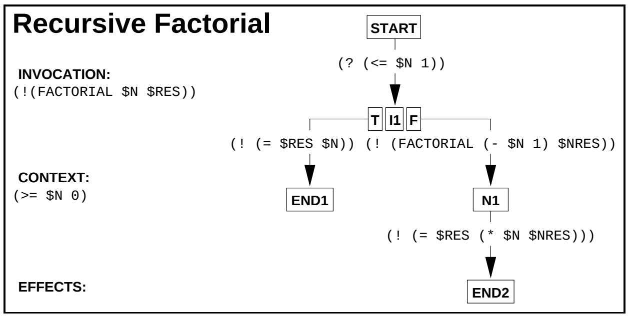

Figure 4: Factorial calculated by a KA

tions and tests that may be performed to achieve given goals or to react to certain situations;

2. a database containing the system current beliefs about the world; and
3. the intention graph, a partially ordered set of all plans chosen for execution at runtime.

An interpreter selects an appropriate KA based on system beliefs and goals, places the selected KAs on the intention structure and finally executes them.

To exemplify, fig. 4 presents a KA to recursively calculate factorials. This KA is considered to possible execution when a request to reach a `FACTORIAL` goal is posted (because of its `INVOCATION` field). It can be selected if the conditions in the `CONTEXT` field are satisfied. If the system fails to traverse an arc and there is no alternative way, the entire KA fails. But when a `END` node is reached, the goal is considered satisfied and facts listed in the `EFFECTS` field are concluded in the system database. A KA can post goals, that will be incorporated into the intention graph and must be satisfied before accomplishing the KA.

C-PRS offers other programming possibilities, like:

- KA invoked by events (and not by goals);
- a communication library to allow its integration with other systems;
- meta-level KAs, that are KAs able to manipulate applicable KAs, goals and intentions of PRS itself;
- incorporation of user-defined primitive actions;
- parallel execution of threads; and
- a comfortable developing environment,

that make possible the implementation of task levels able to manage complex robotic problems [12].

3.2 Implementation of a Task Level using C-PRS

The communication between the functional and the PRS-based task levels is done by specific KAs that use the functions of the modules. The low-level exchange of informations is made by two primitive functions, linked to PRS, that send requests to and receive replies

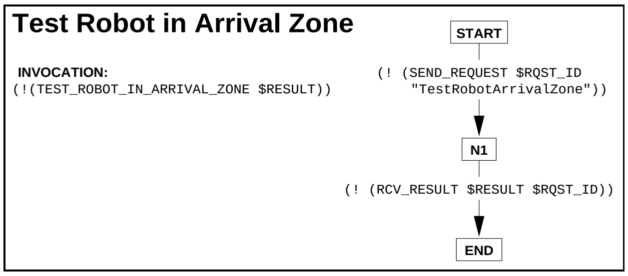

Figure 5: KA communicating with the functional level

from the executive. If one of these functions fails (if they receive an `ERROR` execution status from the executive), the entire KA fails and the goal is not satisfied. In fig. 5 we exemplify with a function to test if the robot arrived to its target zone: the return result, in this case, is a boolean containing the test result.

The goals satisfied by these specific KAs are included in more general KAs, like in fig. 6. This KA was actually used in the Eden experiment of navigation in outdoor environments with the robot Adam [14]. The canonical task to execute was `GO_TO TARGET`, the environment being initially totally unknown and the target corresponding to a landmark recognisable by image processing. The KA of fig. 5 which tests if the robot is near enough to its target intervenes to decides if it is necessary to execute a new cycle of perceiving and classifying the terrain, selecting a new sub-goal, planning a trajectory and moving the robot.

3.3 Validation of PRS Scripts

PRS is a suitable tool for representing complex systems like the decision-making of a robot, but does not offer a formal validation of the resulting implementation. Even if C-PRS can guarantee an upper bound on reaction time, we have neither formal proofs that the structural properties of the set of KAs is appropriated (existence of eternal loops, deadlocks, etc) nor a guaranteed bound on their execution time.

To advance in this direction, a promising possibility is to represent the set of KAs by an equivalent Petri net (PN). If such a representation is obtained, the existing methods to determinate structural and temporal properties of Petri nets can be used to validate the equivalent PRS-based system. The versatility of PRS almost surely prevents defining a general converting rule, but a restricted subset of the PRS syntax can be imposed to make possible this modelling.

The general idea is to represent the KAs by a coloured PN [13], where the PRS-nodes correspond to places and the PRS-arcs to transitions. The variables are propagated by coloured tokens, where the colour symbolises their values. To exemplify, in fig. 7 we show the PN equivalent to the KA in fig. 4.

Each goal in the KA set will cause the apparition of three places in the PN: one to symbolise the posting of the goal and the other two representing the success or not when accomplishing the goal. The PN equivalent

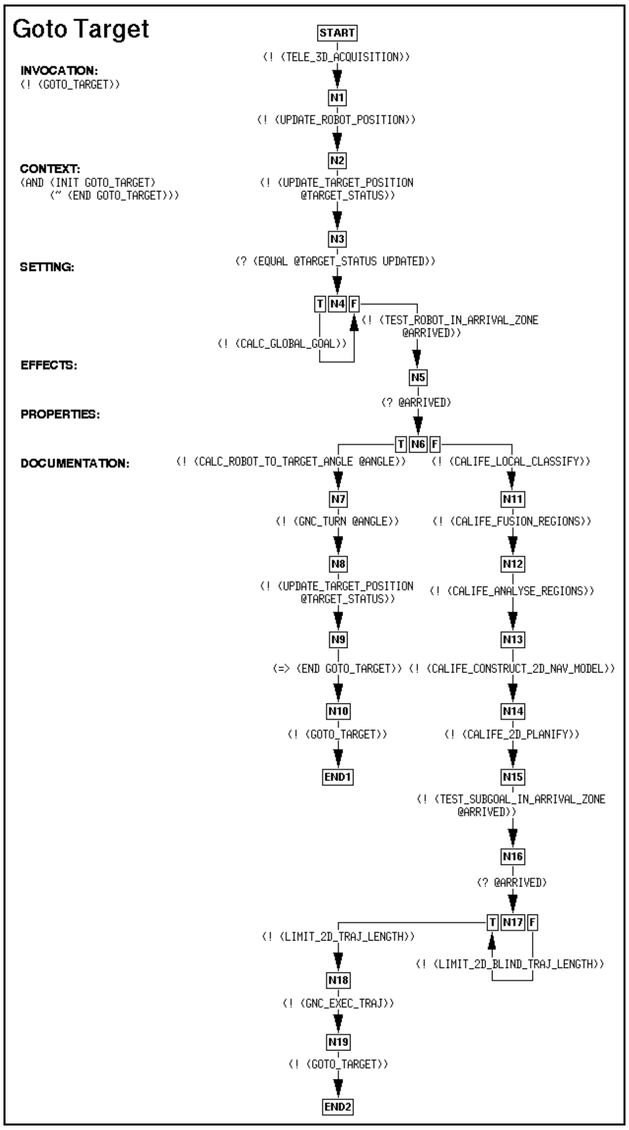

Figure 6: A KA used in the EDEN experiment

to a KA activated by this goal checks the existence of tokens into the `GOAL POSTED` place and puts a token into one of the `GOAL SUCCEEDED` or `GOAL FAILED` places, according to the result of its execution. When a KA posts a goal, the corresponding PN puts a token into the desired `GOAL POSTED` place and waits for the apparition of a token into one of the resulting places.

The database is modelled by a set of places. Each fact that can be asserted or tested corresponds to two complementary places: the existence of a token into a place (or into its complementary place) means that the corresponding fact was (or was not) added to the database. The token colour indicates the value for which the fact (a first-order predicate) was asserted: concluding the `(OPEN DOOR1)` fact is represented by putting a `DOOR1` token in the `OPEN` place.

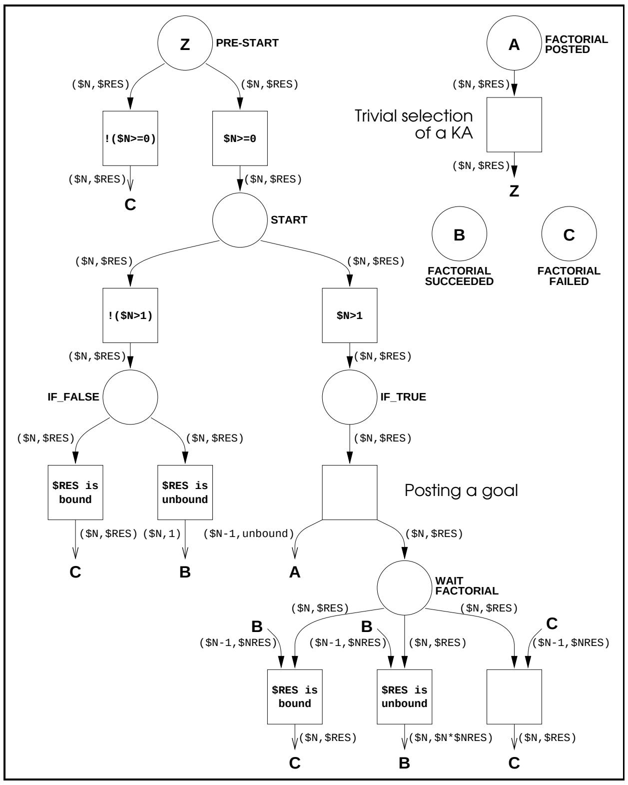

Figure 7: Petri Net representing the factorial KA

The tokens must take their values (or colours) over finite domains for not loosing the possibility of formal analysis. This imposes that KAs cannot use real numbers, for example, and the values of integer variables must belong to finite (and pre-defined) sets[2]. Other conversion rules, which we will not present here, are necessary to represent several aspects of the execution control of PRS, mainly:

- selection of a KA when several ones are applicable to accomplish a goal;
- non-determinism when several arcs starts from the same node; and
- backtracking when an arc fails and other possibilities exist.

If the set of KAs implementing the task level can be represented by an equivalent PN, the provable behavioural properties of the Petri nets, like reachability and liveness [15], can also be proved for the equivalent system. Their meaning, however, must be interpreted: for example, liveness, closely related to the complete absence of deadlocks, is an ideal property for many system, but can signify that the robot task level stays in eternal loop for a given goal. If we include temporal characteristics in the PN, with estimations of the execution times consumed by the functional level, the

[2]The PN in fig. 7, for example, is not analysable.

PN analysis can also supply some indications about the temporal behaviour of the robot task level.

4 Conclusion and Future Work

The general concepts of the LAAS architecture have been validated by some experimental projects: the HILARE robot , the EDEN experiment for planetary exploration and the MARTHA (Multiple Autonomous Robots for Transport and Handling Applications) European ESPRIT project. Many concepts of this article were already present at the design of these architectures (even if sometimes the implementation imposed simplifications) and they are currently being used to develop the control system of the outdoor robot LAMA.

There is a number of possibilities to improve the specification presented in this article, two of them we have begun to evaluate the viability. The first one consists of semi-automating the generation of the Kheops rules that describe the behaviour of the executive's automaton.

The functional modules are assembled using a module generator called G^{en}oM [7]. For this, a formal description of some of their characteristics must be furnished, which includes a listing of the resources required by each function. This knowledge can be used to generate Kheops rules describing the behaviour of the module in case of conflict. These rules, if complemented with specific rules to manage the intermodules conflicts, can constitute a good starting point when developing the executive's automaton.

The second possible improvement concerns the validation of PRS scripts. Some work is currently being done to formally define the subset of the PRS syntax that can be represented by a Petri net; after that, the development of an automatic generator of the equivalent Petri net is envisaged.

References

[1] R. Alami, R. Chatila, and B. Espiau. Designing an intelligent control architecture for autonomous robots. In *ICAR*, pp. 435–440, Tokyo, Japan, Nov. 1993.

[2] P. Baroni, G. Guida, S. Mussi, and A. Vetturi. A distributed architecture for control of autonomous mobile robots. In *ICAR*, vol. 2, pp. 869–877, S. F. de Guixols, Spain, Sept. 1995.

[3] 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.

[4] R. Chatila. Deliberation and reactivity in autonomous mobile robots. *Robotics and Autonomous Systems*, 16:197–211, Dec. 1995.

[5] R. Chatila, R. Alami, B. Degallaix, and H. Laruelle. Integrated planning and execution of autonomous robot actions. In *IEEE ICRA*, vol. 3, pp. 2689–2695, Nice, France, May 1992.

[6] 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 ICRA*, vol. 2, pp. 267–274, Atlanta, USA, May 1993.

[7] S. Fleury, M. Herrb, and R. Chatila. Design of a modular architecture for autonomous robots. In *IEEE ICRA*, vol. 4, pp. 3508–3513, San Diego, USA, May 1994.

[8] M.P. Georgeff and A.L. Lansky. Procedural knowledge. *Proceedings of the IEEE*, 74(10): 1383–1398, Oct. 1986.

[9] M. Ghallab and H. Philippe. A compiler for real-time knowledge-based systems. In *IEEE International Workshop on AI for Industrial Applications*, Hitachi City, Japan, May 1988.

[10] J.-M. Hasemann. Robot control architectures – application requirements, approaches, and technologies. In *XIV Intelligent Robots and Computer Vision (part of the SPIE's Photonics East Symposium)*, Philadelphia, USA, Oct. 1995.

[11] F.F. Ingrand. *C-PRS Development Environment Manual*. ACS Technologies, BP 556 – 31674 Labege – France, 1994.

[12] F.F. Ingrand, R. Chatila, R. Alami, and F. Robert. Embedded control of autonomous robots using procedural reasoning. In *ICAR*, vol. 2, pp. 855–861, S. F. de Guixols, Spain, Sept. 1995.

[13] K. Jensen. Coloured petri nets and the invariant method. *Theoretical Computer Science*, 14:317–336, 1981.

[14] S. Lacroix, R. Chatila, S. Fleury, M. Herrb, and T. Siméon. Autonomous navigation in outdoor environment: Adaptive approach and experiment. In *IEEE ICRA*, vol. 1, pp. 426–432, San Diego, USA, May 1994.

[15] T. Murata. Petri nets: Properties, analyses and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[16] R.G. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, Feb. 1994.

[17] C.E. Thorpe and M. Hebert. Mobile robotics: Perspectives and realities. In *ICAR*, vol. 1, pp. 497–506, S. F. de Guixols, Spain, Sept. 1995.