An Architecture for Autonomy *

R. Alami, R. Chatila, S. Fleury, M. Ghallab, F. Ingrand
LAAS-CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4
E-mail: {rachid,raja,sara,malik,felix}@laas.fr

Abstract

An autonomous robot offers a challenging and ideal field for the study of intelligent architectures. Autonomy within a rational behavior could be evaluated by the robot's effectiveness and robustness in carrying out tasks in different and ill-known environments. It raises major requirements on the control architecture. Furthermore, a robot as a programmable machine brings up other architectural needs, such as the ease and quality of its specification and programming.

This paper describes an integrated architecture allowing a mobile robot to plan its tasks, taking into account temporal and domain constraints, to perform corresponding actions and to control their execution in real-time, while being reactive to possible events. The general architecture is composed of three levels: a decision level, an execution level and a functional level. The later is composed of modules that embed the functions achieving sensor data processing and effector control. The decision level is goal and event driven, it may have several layers, according to the application; their basic structure is a planner/supervisor pair that enables to integrate deliberation and reaction.

The proposed architecture relies naturally on several representations, programming paradigms and processing approaches meeting the precise requirements specified for each level. We developed proper tools to meet these specifications and implement each level of the architecture: IxTeT a temporal planner, PRS a procedural system for task refinement and supervision, Kheops for the reactive control of the functional level, and GenoM for the specification and integration of modules at that level. Validation of temporal and logical properties of the reactive parts of the system, through these tools, are presented.

Instances of the proposed architecture have been already integrated into several indoor and outdoor robots. Examples from real world experimentations are provided and analyzed.

*This paper has been accepted for publication in the International Journal of Robotics Research (Special Issue on "Integrated Architectures for Robot Control and Programming"), 1998.

1 Introduction

The organization of a robotic system determines its capacities to achieve tasks and to react to events. The control structure of an autonomous robot must have both decision-making and reactive capabilities. Situations must be anticipated and the adequate actions decided by the robot accordingly. Tasks must be instantiated and refined at execution time according to the actual context. The robot must react in a timely fashion to events. To meet these requirements, a robot control structure should have the following properties:

Programmability: a useful robot cannot be designed for a single environment or task, programmed in detail. It should be able to achieve multiple tasks described at an abstract level. Its functions should be easily combined according to the task to be executed.

Autonomy and adaptability: the robot should be able to carry out its actions and to refine or modify the task and its own behavior according to the current goal and execution context as perceived.

Reactivity: the robot has to take into account events with time bounds compatible with the correct and efficient achievement of its goals (including its own safety).

Consistent behavior: the reactions of the robot to events must be guided by the objectives of is task.

Robustness: the control architecture should be able to exploit the redundancy of the processing functions. Robustness will require the control to be decentralized to some extent.

Extensibility: integration of new functions and definition of new tasks should be easy. Learning capabilities are important to consider here: the architecture should make learning possible.

Robot control architectures being at the core of the design of autonomous agents, many authors have addressed this issue. The approaches differ in several manners, sometimes by the philosophical standpoint itself: some projects aim at imitating living beings, while others are more AI oriented. The first trend of research sought inspiration from biology and ethology and has mainly yielded stimuli-response based systems. The second trend tried to use symbolic representations and some reasoning capacities. We will overview in section 7, some of these architectures and compare their approaches with our own work.

The architecture we present aims at meeting the above-mentioned requirements by endowing autonomous robots with deliberation and reactivity capabilities. We shall describe it in this paper as follows:

– In the next section we present the rationale of the system organization and overview its main features.

– We then introduce the representations and tools that we have developed or chosen for designing the components of the architecture (sections 3 to 5). These tools are critical for implementing the concepts in a feasible architecture and for endowing it with the required properties. This important aspect, akin to real-time software engineering, is one of the claimed contributions of this work. However the tools proposed here are not a unique characterization of our architecture: one certainly may rely on other representations and tools, with similar or improved properties. We will underline needed requirements and

point to shortcomings and restrictions in our tools that call for improvement. The reader who wants to have a complete overview of the architecture without getting into the details of the proposed tools may skip part of sections §3.4 (about GenoM), §4 (Kheops), §5.2 (PRS) and §5.3 (IxTeT).
– An instantiation of the architectural concepts using the specific tools integrated together is presented in section 6. This is just one of the few that we have implemented and experimented with our indoors (HILAREs) and outdoor (ADAM, EVE, LAMA) mobile robots, and this work is still on going. This fairly broad experimental testing, redesign, extension, and tuning has been essential in the development of the different concepts and tools. The relative ease with which we have been able to integrate research results and build up incrementally robot functions is one of the nice properties of this architecture.

2 Global View

This section outlines the general approach and presents the system architecture. Each component is detailed later.

Together with action and perception capacities, a robot architecture has to be a suitable framework for the interaction between deliberation and action. Deliberation here is both a goal-oriented process wherein the robot anticipates its actions and the evolution of the world, and also a time-bounded context-dependent decision making for a timely response to events.

The robot will face high emergency situations where a first and immediate reflex reaction should be performed. But such situations often require second step actions for correcting more globally the robot's behavior. This will need more information gathering or more search in some representation space of the environment and of the robot state.

Hence we can consider that the architecture should include several task-oriented and event-oriented closed-loops in order to achieve both anticipation capacities and real time behavior. We decompose the robot architecture into three levels (Figure 1), having different temporal constraints and manipulating different data representations. From bottom up, the levels are:

- **a functional level.** It includes all the basic built-in robot action and perception capacities. These processing functions and control loops (image processing, obstacle avoidance, motion control, etc.) are encapsulated into controllable communicating modules (section 3). In order to make this level as hardware independent as possible, and hence portable from a robot to another, its is interfaced with the sensors and effectors through a *logical robot level*.

  In order to accomplish a task, the modules are activated by the next level.

- **an execution control level,** or Executive. It controls and coordinates the execution of the functions distributed in the modules according to the task requirements (section 4).

- **a decision level.** This level includes the capacities of producing the task plan and supervising its execution, while being at the same time reactive to events from the previous level. This level may be decomposed into two or more layers, based on the same conceptual design, but using different representation abstractions or different algorithmic tools, and having different temporal properties. This choice is mainly application dependent. Typically for the architecture of an autonomous agent, there will be two such layers as we shall see (section 5).

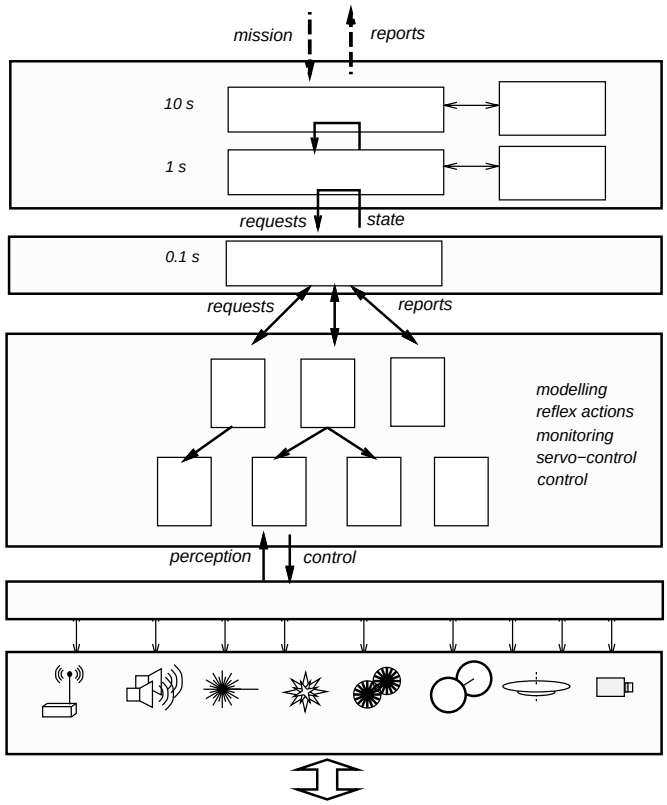

Figure 1: Generic Architecture.

To realize each level in this architecture, we have developed adequate tools. We describe and discuss them in the corresponding sections.

The architecture has been designed, implemented and experimented within several mobile robots [2, 16]. The implementations, presented in here, may differ in the number of layers of the decision level, or in the tools used to realize each of the levels.

3 The Functional Level

The Functional Level embeds a set of elementary robot actions implementing processing functions and task-oriented servo-loops (motion planning, vision, localization, tracking, motion control, etc.), and programmable monitoring functions (condition-reaction pairs) used to trigger reflexes. The functions are embedded into *modules* ([22]). A module is a software entity capable of performing a number of specific functions by processing inputs from, or acting on, physical robot devices and/or other modules. A module may read data exported by other modules, and output its own processing results in exported data structures. The functional level can be seen as a library of functions with real-time abilities. The organization of the modules is *not fixed*. Their interactions depend on the task being executed and on the environment state. This is an important property that enables to achieve a flexible and not a systematic robot behavior.

This section presents the functional level requirements (§3.1) and organization (§3.2), as well as the standardized module structure (§3.3) and a development environment, GenoM, for integrating the functions into modules (§3.4).

3.1 Requirements

The operational functions of the functional level allow the robot to move, perceive, model, compute trajectory, estimate its position and so on. All these processes manipulate numerical data and perform algorithmic computations. For example position estimation here is based on odometry and external landmarks visually recognized [9].

It is seen from the decision level as a library of "intelligent" services to which it interfaces through the execution control level.

The functional level interacts continuously with the environment while making servo actions and reflex actions. It meets the requirements of:

- **A real-time distributed system:** the functional level embeds all the processes, implemented on several processors, that control the interactions between the robot and the environment through sensors and effectors. Some of them have strong real-time constraints such as servo-controls, event monitoring and reflex actions. This level offers basic real-time procedures (parallelization, synchronization, inter-task communication) and permits time-bounded execution and reactivity to asynchronous events.
- **A controlled system:** the functional level does not include decision capacities nor predefined behaviors (i.e., pre-wired and fixed interactions between its functions). It can be considered as a library of reactive services dynamically parameterized and selected by an upper controller. This selection depends on the task but also on the current execution context inferred from the informations returned by the services. In other words, the services must be observable and programmable in a coherent and predictable way.
- **An open system:** the robot, as a complex and experimental platform should be extensible. Services should be easily integrated, modified or suppressed according to the considered application and to recent developments. This requires an incrementally designed

system with common integration methodology and validation procedures for the involved developers.

These issues call for a modular architecture, for a standardized structure and interfaces of the modules and for a well defined development methodology.

3.2 Organization: A Network of Modules

The Functional Level is made of a network of modules. A module is a software entity that embeds services, i.e., processings, related to given resources. A resource may be physical or logical: sensors, effectors or data (position, map, image, trajectory, etc.). Typically, a functional level offers at least modules to estimate and to control the position of the robot, and, depending on the application, modules to control proximetric sensors or cameras, to model obstacles, to compute trajectories, and so on.

A module has the responsibility of the resources it manages and acts upon requests by means of a set of specific processings (algorithms). This decentralized "responsibility" has two main objectives: (i) to simplify and to relieve the central control: the notion of service hides the complexity and the dynamics of the low levels processings, (ii) to increase the robustness of the whole system by introducing control and recovery procedures at the lower level.

The services are parameterized and activated asynchronously through a non-blocking client/server protocol: a relevant request, that may include input parameters, applies to every service of each module. Thus requests start processings. An ongoing processing is called an *activity*. The end of the service is marked by a *reply* returned to the client that includes an *execution report* and possibly data results. For instance, the execution of a trajectory is an activity that runs until the trajectory is over, or until an error occurs. In this last situation, the execution report characterized the failure (important drift, obstacle on the way, ...) that can be recovered by the decisional level.

A module does not know a priori its clients: the client/server relationship is established dynamically. This offers a very flexible architecture: depending on application needs, modules may be included or removed from the system, and, the services are selected or not according to the context.

As an other consequence, the services of a module can be used by another module. This allows to design complex services combining primitive - but general and thus reusable - services. For instance, Figure 2 shows a tree of seven active modules accomplishing the complex service "follow a visual target while avoiding unexpected obstacles". The arcs of the tree (black arrows) represent the client/server relationships: the top-most activity "`tracking + avoidance`" uses two services: the "`visual tracking`" activity tracks the target and produces periodically an updated reference position for the robot which is filtered by the "`local avoidance`" activity to drive round local obstacles. Each of these two services uses itself two basic services: "`platform control`" (control of the orientation of the on-board camera) and "`video acquisition`" for the first one; "`position control`" (feedback motion control of the robot) and "`sonar control`" (proximetric data to detect obstacles) for the second one. The data flow is only partially represented in this figure (the

three gray arrows).

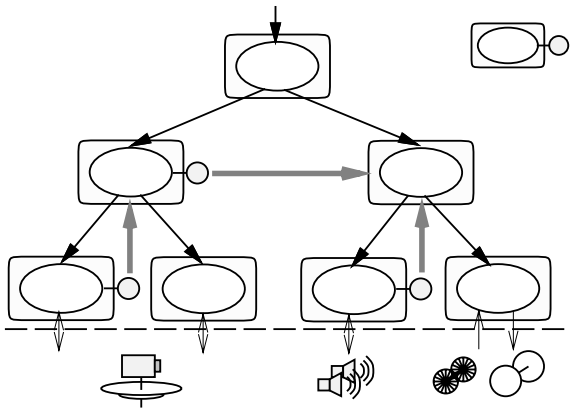

Figure 2: **Activity tree example:** local obstacles avoidance during visual tracking. The black arrows show control flow and the grey ones the main data flow.

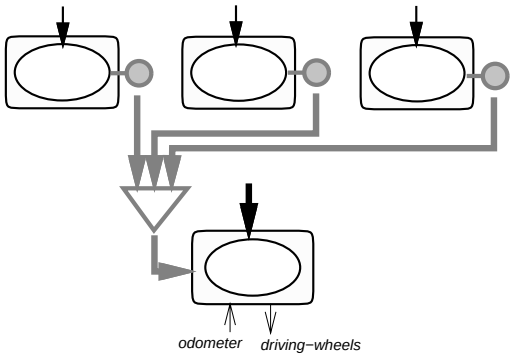

Figure 3: **Data flow redirection example:** different activities can produce a reference position in a poster. The parameter P_n of the **tracking_request** indicates the selected poster (P_n = P_1 or P_2 or P_3 or ...).

Thus module interactions are task dependent. In the above example, avoidance and tracking movements are combined by the "`tracking + avoidance`" activity. If, in another situation, the robot moves in an obstacle-free space, then the "`visual tracking`" activity will directly transmit the reference position to the "`position control`" activity (see also next example): the services "`local avoidance`" and "`sonar control`" are no more activated.

During their execution, the activities may have to read and/or produce data (for instance the reference position in the previous example). These exchanges are done via *posters* (the circles in Figures 2 and 3). A poster is a structured shared memory only writable by its owner, the module in charge of the exported data, but readable by any element of the architecture. The posters are completely independent form each others. They are not regions of a common blackboard. The content of a poster is data produced by the module (a robot position (x, y, θ), an image, a parameterized trajectory, etc.). There are no consistency problems between posters, i.e., a given piece of data is not provided by more than one module (and into not more than one poster) in the system, and it is time stamped. The data semantics are clearly defined. For example, odometric position is not in the same poster provided, say, by a module using GPS (if the robot has such a device). A fusion of these two informations is achieved by a specific module (or function in the same module) which will consider the time consistency of its inputs, and its output poster has a different address and semantics.

The posters permit to easily *redirect the data flow*. The poster to be read by an activity is simply specified through its execution request parameter. For instance, if we consider again the activity "`position control`" (at the bottom of the Figure 3) that servo-control the robot position on a given reference, this reference could be computed and

exported by a "`visual tracking`" activity, *or* a "`path tracking`" activity *or* as well a "`wall following`" activity.

3.3 Generic Module Anatomy

As a result of the distributed organization of the functional level, a module must be reliable: a malfunction may have catastrophic consequences on the whole system (failure of the application or even robot damage). The non-nominal situations must be anticipated to be managed immediately by the module; and, above all, clear information must be returned to the client to integrate the problem (target not found, path blocked, etc.).

First of all, a module must check the relevance of its inputs: validity of the request parameters and applicability of the required actions. Indeed, due to resource sharing, the services offered by a module are generally, but not systematically, conflicting. For instance, only one motion activity is possible at a given instant because the motors are not a sharable resource (this is true for any robot). Hence the module managing this resource can be executing one and only one motion request at a time.

To satisfy the *reactivity* requirement, conflicts are resolved according to the following rule: *the latest request is always preemptive* on current conflicting activities. Therefore, all activities must be interruptible with a delay compatible with the dynamics of the "subsystem" controlled by the module. Hence, despite the large diversity of the processings (periodic/aperiodic, synchronous/asynchronous, self/controlled termination), they must be integrated in such a way as: to be interruptible; to terminate properly to ensure the stabilization of the controlled process or the coherence of the produced data, but also to get back into conditions for a new start; to detect failures (algorithm limitations, unavailability of resource, memory limitation, inconsistent results, etc.) and inform about them.

These constraints, unlike the previous control aspects, are closely linked to processing algorithms. The execution context should offer a framework to structure the programs and mechanisms to manage non-nominal execution modes (interruptions and failures).

Structure of the Generic Module Every module is coded along the template of a "generic" module [15, 22] which clearly distinguishes control and execution aspects. The structure of the generic module has two parts (Figure 4):

- **The controller** manages the module according to the clients' requests and the current state of the module. It is implemented as an asynchronous process which wakes up on requests or internal events.
- **The execution engines** carry out the activities required by the controller. Each engine is one periodic or aperiodic process that is the execution context of operational functions.

These two parts interact by means of typed asynchronous events and two databases (Figure 4) : (i) *the functional database* includes all the data relative to the controlled system: default settings, parameters updated through requests, and processing results

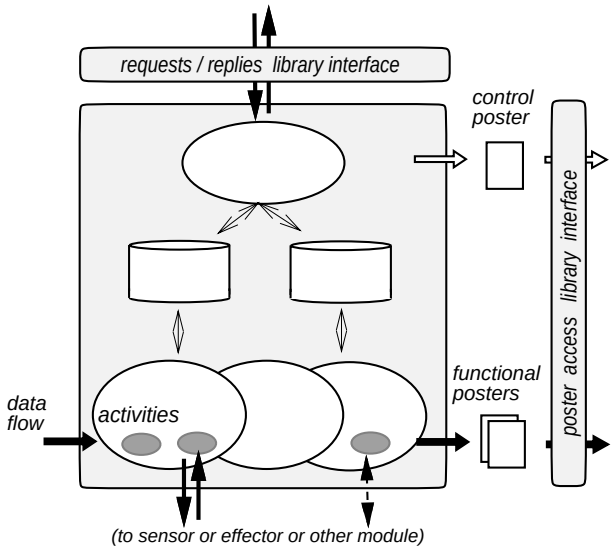

Figure 4: Structure of a module

(exported within the replies or the functional posters). (ii) *the control database*, which has the same structure for all modules, contains the states of the activities, the commands from the controller and the reports issued by the execution engines.

Moreover, the specification of the generic module includes two standard interface libraries to enable access to the services (emission of requests and reception of replies) and to the content of the posters.

Module Management At the reception of a request, the controller creates an activity which is managed according to a logic represented by a *control graph* composed of 5 states and 9 transitions (Figure 5). The transition events are imposed by the controller (noted `event/-` on the figure) or signaled by the execution engine (noted `-/event`).

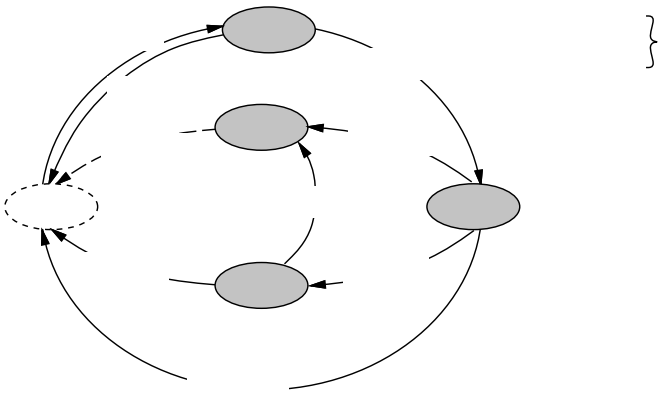

Figure 5: Control graph of an activity

The activity is initially brought from the fictitious state `IDLE` to `INIT` where the controller checks the request parameters and verifies if the new activity does not conflict with activities in progress. In such a case, incompatible activities are interrupted (event `abort/-`): once in `INTER`, the execution engine has to concluded their processings. The controller then orders an execution engine (events `exec/-`) to process the new activity (state `EXEC` described below). Finally, the execution engine signals the end of an activity sending one event `-/ended`. On the reception of this event, the controller sends back the final reply to the client with the specific execution report (`OK, failed, interrupted,` or other). The particular state `FAILED` is used to freeze temporarily the module in case of an activity error that requires a specific intervention or a re-synchronization.

Execution of the Activities The processing of an activity (state `EXEC`) is supported by an execution engine. It consists in the evaluation of *non interruptible* code parts that correspond to different steps of the algorithm (initialization, main nominal loop, normal or urgency termination, ...). These code parts, called *codel* (code element), are the smallest processing units known by a module.

Thus, an activity is the execution of a sequence of codels *selected dynamically* according to the previous codel evaluation and to the controller orders: each codel indicates which codel is to be executed next (eventually itself). However, an interruption (event `abort/-`) from the controller forces the selection of a termination codel (state `INTER`).

Typically, for a periodic activity (monitoring, servoing), once the initializing codel has ended normally, a same main codel is executed periodically until termination conditions are satisfied, or until a problem occurs. The adequate termination codel is then selected. This decomposition of an operational function into codels allows to structure and monitor the program execution, and, above all, to express explicitly the possible interruption points (i.e., the transitions between codels) and to install adequate termination/recovery procedures.

3.4 The Generator of Modules GenoM

Every module of the functional level is an instance of the generic module. However, not to mention the tedious sides of such a systematic and complex construction, it is well known that a precise definition of a system is not the guarantee of a correct implementation, all the more so when many people are involved. With GenoM this instantiation is formalized: only the specific parts of one module have to be expressed, and the new module is automatically generated.

A description language allows to define the module: all the services it managed by a module, their input and output parameters, the associated codels and their real-time characteristics (period, delay), the exhaustive list of the possible execution reports.

From this description, the module is generated: as illustrated by figure 6, the description file is the input of the Generator of Modules GenoM (1). This file is parsed and code files are produced by instantiating the *generic module* skeleton (2). The compilation of these intermediate files produces the three output of this generation (4):

- a module that can run on a Unix workstation (for emulation purposes) or on a VxWorks real time system,
- the interface libraries, used by the clients, to invoke the services and to access to the data exported by the module,
- an interactive program to run preliminary tests.

Thus, using the formal description, a module, with empty codels, can be already generated and *integrated* to the functional level. The codels will then be incrementally developed and linked to the module (3).

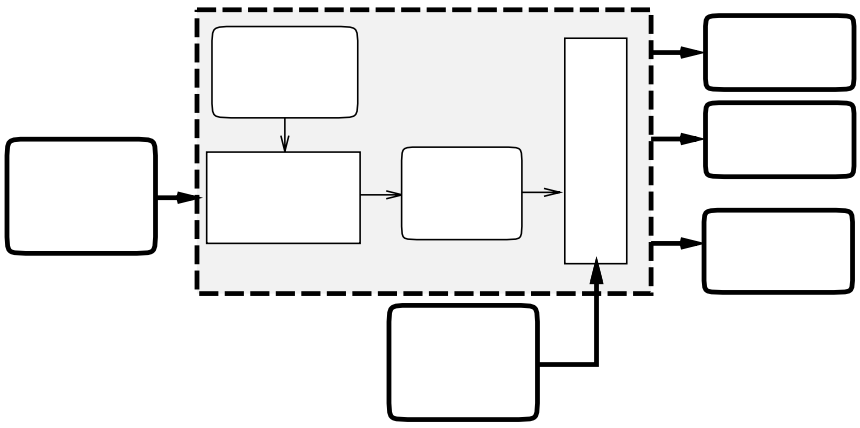

Figure 6: The generator of module

To sum up, a typical development cycle includes the four following steps (Figure 6):

(1) Formal description producing the input file for GenoM.

(2) Building of the module: generation of the sources and compilation.

(3) Coding the algorithms inside the codels and link editing with the module. This is guided by codel shells produced at step 2.

(4) Execution of the module on UNIX or VxWorks.

The algorithms of the codels are incrementally included and finalized by iterating on steps (3) and (4). Any service modification or addition is simply achieved by editing the formal description file (1).

In the frame of an integration with third-party modules, the formal description of the modules handled by GenoM provides a complete data sheet to summarize the awaited services, the behavior and the data structures of the modules.

The GenoM environment does not validate a priori the temporal constraints of a module – which are dependent of the algorithms and the host CPU capacities – but it offers a systematic methodology and a set of tools to verify the on-board system (using chronograms).

Finally, GenoM offers a structured development context that allows the programmer to focus on its algorithms, without caring about operating system, communication protocols

or architecture design. The automatic production of the modules guarantees a standard behavior and avoids tedious logical tests. This standardization associated to the formal description allows to easily develop a complex functional level.

This system is now systematically used in all our robotics experiment (indoor and outdoor mobile robots, manipulators), and has been validated in large scale projects (§ 6).

Beyond these software engineering aspects, the common template offers many perspectives. Firstly, general tools or methods based on this common template can be easily extended to all the modules. In this way, we are working on the demonstration of logical and temporal properties of the modules. Secondly, any evolution of this common template affects all the modules by simple regeneration. One important extension is the automatic generation of the executive rules (see §4) of every module using its formal description.

4 The Execution Control level

This level, composed of a single system, the Executive, is a pivot interface between the decision and functional levels. It fills the gap between the slow symbolic processing (0.1 to few seconds), and the higher bandwidth computation on numerical data (10 to 100 Hz).

The Executive is a purely reactive system, with no planning capability. It receives from the decision level the sequences of actions to be executed. It selects, parameterizes and synchronizes dynamically the adequate functions of the functional level. It is at the top of the activity tree (fig. 2).

The selection and instantiation of the request parameters depend on the task and on the current state of the system. This state is maintained by the Executive, according to the ongoing activities and to the output of previous processing, *i.e.* according to requests sent and to replies returned by modules once an activity is over. Replies may trigger requests delayed by the Executive. A report must be returned systematically to the decision level (*e.g.* robot-localized, target-not-found, trajectory-computed), to enable plan supervision and choice of next actions. The instantiation of the request parameters consists generally in redirecting data previously produced by others activities (*e.g.* a trajectory computed by a trajectory planner which is now to be executed). The Executive manages both the control flow and the data flow of the functional level. It is organized as follows (fig. 7):
– the *request monitor* reacts to decisions from the higher level and selects the adequate requests using the *request description database* that describes the mapping of requests into modules, their input parameters, and the data distribution (posters).
– the *replies monitor* surveys the execution of functions, it takes delivery of the replies from the functional level and returns reports to the decision level.
– both monitors maintain the *execution state database* and use a set of rules to manage possible conflicts.

Most conflicts involves functions within the same module, they are managed locally. The executive handles conflicts between different modules. It knows about priorities, it may interrupt a module, or leave a request pending. Using its knowledge about the behavior of the functional modules, the Executive maintains a logical description of the operating

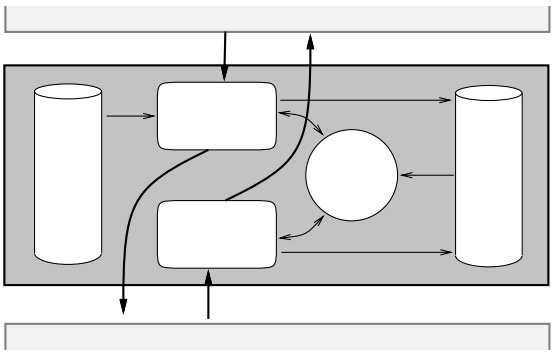

Figure 7: Executive organization

state of each module. This permits to process queries from the decision level about the current use of robot resources.

The Executive is a simple but critical component of the architecture. The verification requirements of the code implementing it are hence very important. One needs to formally prove its properties: consistency, completeness and bounded response time. This code must integrate all the logical constraints of a very large number of possible requests.

In previous instances of the architecture, the Executive rules were hand written in PRS (see §5.2). However, because of the combinatorial explosion, a manual synthesis of an automaton – for analysis and proof purposes – is not feasible. A more satisfactory approach is to automatically synthesize the Executive code from the set of rules. Moreover, these rules could be, at least partially, automatically generated by GenoM from module specifications. For this purpose, we have developed a formal language and the corresponding programming environment, called Kheops, which produces automatically the automata from a set of logical rules, and which permits the verification of the logical and temporal properties.

The modules at the functional level can be either asynchronous or synchronous. The temporal properties are more easily proved if the Executive controlling these modules is itself a synchronous system (e.g. as [7, 8]). The logical properties on the other hand are more easily checked if one relies on a formal representation, such as logical clauses or rules. To obtain both, temporal and logical properties, compiling techniques are needed at the control level; they require a restricted representation, but permit a drastic reduction in complexity.

The knowledge representation that meets the compiling requirement of a finite number of possible rule chainings involves basically a monotonic deduction on a propositional logic. Kheops represents the state of the functional level through a finite set of multi-valued attributes. Time is seen as a discrete sequence of periods. An attribute ranges over a finite domain of values or intervals. Attributes are partitioned into input and output attributes. At each period, Kheops reads the values of input attributes, it deduces from them the values of output attributes. The knowledge base is logically consistent if it defines a (partial) mapping from input to output attributes; it is complete if this mapping is complete.

The process of reading input and mapping it to output is repeated periodically. If

required by the knowledge base, Kheops may maintain past values of some input and/or output attributes. Attributes whose past values are needed in the knowledge base are buffered into cyclic buffers, which are automatically dimensioned at compile time. This does not change in principle the restricted assumption of reasoning on static states (*i.e.*, a Markovien system). It is just a convenient extension of the representation space.

Kheops compiler transforms a set of propositional rules into an optimized decision network, written as a C-runtime code. Nodes in this net are input attributes. Branches issued from a node, labeled by values or intervals, define a partition of the range of this attribute. Leaves and subtrees are labeled by values of output attributes. Compiling succeeds if and only if the knowledge base is consistent (otherwise an inconsistency is found along some path in the network). In that case, for a given input state there is just one valid path in the net from the root node to a leaf. Each such path corresponds to a logical model or a subset of models of the knowledge base. In that sense the compiler follows a sound inference algorithm.

Running the runtime code corresponds to a deterministic traversal of the net, in $O(m)$, m being the number of input attributes. The optimization tries to achieve well balanced nets, with logarithmic complexity. After compiling, an upper bound on the response time is computed from the longest path in the net (to some constant factor).

Illustration In a demonstration task our robot had to localize some objects in an initially unknown environment. This required 21 requests, dispatched into 8 modules (servo-control, local avoidance, trajectory planner, localization, *etc.*). The Kheops knowledge base consisted of 130 rules : 55 rules are related to communication protocols; they are standard and fit all experiments; 54 rules correspond directly to the formal description of the modules and could be automatically generated; only the 21 remaining rules express inter-modules constraints that are application dependent. Kheops compiler has synthesized a tree of more than 6000 branches; it factorized it into a net of 213 nodes and leaves, whose maximal depth is 8 steps. This provides the maximal response time of the system.

Our next step is to synthesize, at least partially, the knowledge base using GenoM, and to extend this system to all our experiments.

5 Decision Level

The Decision level is in charge of all the processes that require anticipation and some global knowledge of the task and the execution context. It embeds the deliberative capacities of planning and decision-making. This level should remain reactive to incoming events. Planning requires an amount of time usually longer than the dynamics imposed by the environment, in which the robot is permanently acting. To enable the integration of deliberation with reactivity, this level comprises two entities: a planner and a supervisor (Figure 8). The *planner* produces the sequence of actions necessary to achieve a given task or to reach a given goal. It is used as a resource by the *supervisor* which actually interacts with the next level, controls the execution of the plan and reacts to incoming

events. This paradigm consists in guaranteeing a bounded reaction time for a response to an event (possibly after the reflex reaction already taken by the functional and execution control levels).

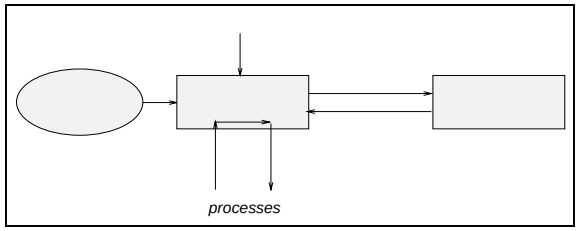

Figure 8: Supervisor - Planner paradigm

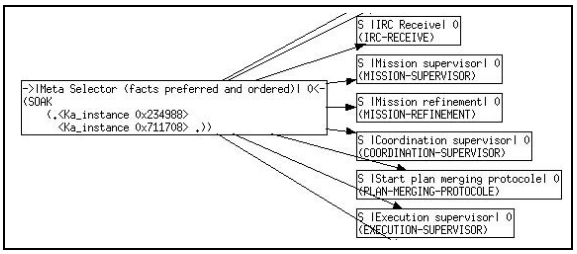

Figure 9: Partial Task Graph Snapshot

The planner is given a description of the state of the world and a goal; it produces a plan. The "quality" of the produced plan is related to the cost of achievement of a given task or objective (time, energy, ...), and to its robustness, i.e., its ability to cope with non nominal situations. This last aspect is one of the motivations of our approach: besides a plan, the supervisor will also rely on a set of execution "modalities" expressed in terms of:

- constraints or directions to be used by a lower planning level if any;
- description of situations to monitor and the appropriate reactions to their occurrence;

such reactions are immediate reflexes, "local" correcting actions (without questioning the plan), or requests for re-planning.

These "modalities" provide a convenient (and compact) representation for a class of conditional plans. The automatic synthesis of these modalities still remains to be investigated. It is possible to produce useful modalities in a second step by performing an analysis of the plan as produced by a first "classical" planning step. Such analysis can be based on a knowledge of the limitations of the planner itself and of the world description it uses, as well as on domain or application specific knowledge.

The supervisor interacts with the planner and with the other levels. These are viewed as a set of processes which exchange signals with the supervisor and which correspond to actions of the robot as well as to events associated with environment changes independent from robot actions. The processes are under the control of the supervisor which has to comply with their specific features. For example, a process representing a robot motion cannot be cancelled instantaneously by the supervisor: indeed, such a process has an "inertia". The supervisor may request a stop at any moment during execution; however, the process will go through a series of steps before actually finishing. The simplest way to represent such processes is through finite state automata.

The activity of the supervisor consists in monitoring the execution of the plan by performing situation detection and assessment and by taking appropriate decisions in real time, i.e., within time bounds compatible with the rate of the signals produced by the processes and their possible consequences. The responsibility of "closing the loop" at the level of plan execution is entirely devoted to the supervisor. In order to achieve it, the

supervisor makes use only of deliberation algorithms which are guaranteed to be time-bounded and compatible with the dynamics of the controlled system. Indeed, deliberation algorithms which do not verify this property are actually performed by the planner, upon request of the supervisor. Besides the plan and its execution modalities, the supervisor makes use of a set of *situation-driven procedures* embedded in a database and independent of the plan. These procedures are predefined in a flexible way: they take into account at execution time the current goal and plan by recognizing specific goal or plan patterns.

Hence, the planner can be viewed as a resource used by the supervisor in order to enhance its ability to deal with a situation, or to provide a plan to reach a goal state.

The decision level may be split into more than one layer, each based on the same concept of planner/supervisor pairs. Decomposition into layers relies in general on different representation abstractions. It may be heuristic in order to improve performance, and better take into account the interactions with the environment. Hence, there may be several decision layers, the lower ones manipulating representations of the environment and actions which are more procedural and closer to the execution conditions. For example, the same "`Go-to [place]`" action included in a plan can sometimes (or at some stage) be executed by planning a geometrical trajectory, by visually tracking a feature, or following a wall using ultrasonic sensors, etc. depending on the actual execution situation. Hence in the instantiation suggested by Figure 1, the decision level includes two "supervision-planning" layers. The higher **mission layer** achieves general task planning using a temporal planning system which produces a plan as a set of partially ordered tasks with temporal constraints. The **task refinement layer** is a contextual planner that receives tasks from the higher planning layer, transforms them into sequences of actions and supervises their execution. This decomposition of the decision level into two layers is probably typical for autonomous mobile robots.

Finally, the supervisor is in charge of receiving the task or the goal from a user interface. They are either sent to the planner for producing the sequence of actions achieving them, or directly executed if they correspond to procedures that are already present in the system. For instance, a basic user command "`move(d)`" which semantics are to move the robot `d` meters ahead, under the responsibility of the user, is directly passed to the functional level for execution. However, modalities could be attached to monitor possible collisions. A "`go-to(x,y)`" user command which semantics is a motion to a given point corresponds to a procedure using a motion planner and execution processes (obstacle avoidance, etc.) all at the functional level, and does not need further task-level planning. On the other hand, a "`go-to(place)`" which semantics are to move, say, inside a building with several rooms will require planning at the task level.

In the next two sections, we discuss the role and requirements of the supervisor (§5.1) and present PRS, the system we use in our architecture for the supervision function (§5.2). Similarly, the planner requirements are considered in section 5.3, and a planner, IxTeT, is described in section 5.3.

5.1 Supervision

The role of the supervision system is to follow the proper execution of the plans and tasks it has launched, while monitoring a number of particular conditions which may lead to specific reactions. Present on the two layers of the decision level, it interacts with the user, the missions planner, the task refinement module and the executive. For reactivity reasons, the reasoning carried on by the supervision remains short, time predictable and interruptible.

The main functions the supervision provides are:

- in the "plan supervision" layer:
  – interpretation of the mission given by the user,
  – call to mission planner,
  – transmission of the produced plans to the level below,
  – supervision and control of the execution of these plans.

- in the "task supervision" layer one can find similar type of activities geared toward task refinement:
  – interpretation of the higher level plans,
  – call to the task refinement module,
  – sending request to the functional level through the executive,
  – analysis of the requests results and asynchronous events coming from the executive.

All these functions require the following characteristics:
– a high level language allowing for the representation of goals and plans, and for the expression of asynchronous events and reactive actions
– the ability to deal with different tasks/activities in parallel
– temporal properties (guaranteed bound on reaction time, i.e. the time it takes for the system to react,no necessarily respond, to new events)
– control over the various possibles high level strategies available to the system,
– easy integration with the other components and modules of the architecture.

One of the tools we use in the decisional level is the Procedural Reasoning System PRS presented below. It is intended to be used as the Supervision system, however, in particular instances of our architecture, its role was extended to encompass the executive level and also some planning activities.

5.2 Supervisor using PRS

PRS [30, 28] is composed of a set of tools and methods to represent and execute plans and procedures. Procedural reasoning differs from other commonly used knowledge representations (rules, frames, ...) as it preserves the control information (i.e. the sequence of actions and tests) embedded in procedures or plans, while keeping some declarative aspects. PRS is composed of:

- **a database** which contains facts representing the system view of the world and which is constantly and automatically updated as new events appear. In our robot architecture, the database contains symbolic but also numerical information such as the position of the robot, the status of its arm, the payload in its gripper, pointers to trajectories produced by a motion planner, the currently used resources, etc.
- **a library of procedures/plans,** each describing a particular sequence of actions and tests that may be performed to achieve given goals or to react to certain situations. The content of this procedure library is application dependent. PRS does not synthesize plans by combining elementary actions, but by choosing among alternative procedures/plans or execution paths in an executing procedure. Therefore this library contains the usual plans needed to perform the tasks for which the robot is intended as well as plans for which there is no need or no time to call the planner.
- **a task graph** which is a dynamic set of tasks currently executing. Tasks are dynamic structures which execute the "intended plans", they keep track of the state of execution of the intended procedure, and of the state of their posted subgoals. This task graph can be thought of as the set of processes of an operating system. Figure 9 presents an example of the task graph, in a mobile robot application, which contains the tasks corresponding to various activities the robot is performing at this moment.

An interpreter runs these components. It receives new events (both from outside and from asserted facts) and internal goals (1), it checks sleeping and maintained conditions, selects appropriate plans (procedures) based on these new events, goals, and system beliefs (2), it places the selected procedures on the task graph (3), it chooses a current task among the roots of the graph (4) and finally executes *one step* of the active procedure in the selected task (5). This can result in a primitive action (6), or the establishment of a new goal (7).

Information about how to accomplish given goals or to react to certain situations is represented in PRS by declarative plan/procedures.

Each procedure consists of a *body* – presented under a textual (Figure 10) or graphical (Figure 11) format – which describes the steps of the procedure/plan[1], an *invocation* condition, which specifies the goal the procedure may fulfill (in our example the goal to get the robot to a particular position) or the events to which it reacts, and a *context* describing under which situations the procedure is applicable.

In PRS, *goals* are descriptions of a desired state associated to a behavior to reach/test this state. The possible goals are the goals to *achieve* a condition, to *test* a condition, to *wait* for a condition to become true, to passively *preserve* and to actively *maintain* a condition while doing something else. For example, the procedure on Figure 10 first *achieves* a notification of all the robot subsystems, then *waits* at most 60 seconds a particular robot status, then tests this status. If the test succeeds, it will analyze the terrain, etc.

The *wait*, *preserve* and *maintain* operators can usually be used to implement sophisticated supervision and control operations. There are also two operators which can be used to explicitly assert or retract information from the database. For a statement `c` they are

[1]Some procedures, called action procedures, just have an external function call as body.

```
(defka |Long Range Displacement|
:invocation (achieve (position-robot $x $y $theta))
:context (and (test (position-robot
            @current-x @current-y @current-theta))
     (test (long-range-displacement
            $x $y $theta @current-x @current-y
            @current-theta)))
:body ((achieve (notify all-subsystems displacement))
 (wait (V (robot-status ready-for-displacement)
          (elapsed-time (time) 60)))
 (if (test (robot-status ready-for-displacement))
     (while (test (long-range-displacement
                   $x $y $theta @current-x
                   @current-y @current-theta))
       (achieve (analyze-terrain))
       (achieve (find-subgoal $x $y $theta
                              @sub-x @sub-y @sub-theta))
       (achieve (find-trajectory $x $y $theta @sub-x
                                 @sub-y @sub-theta @traj))
       (& (achieve (execute-trajectory @traj))
          (maintain (battery-level 0.200000)))
       (test (position-robot @current-x @current-y
                             @current-theta)))
   (achieve (position-robot $x $y $theta))
   else
   (achieve (failed)))))
```

Figure 10: A Plan for Long Range Displacement

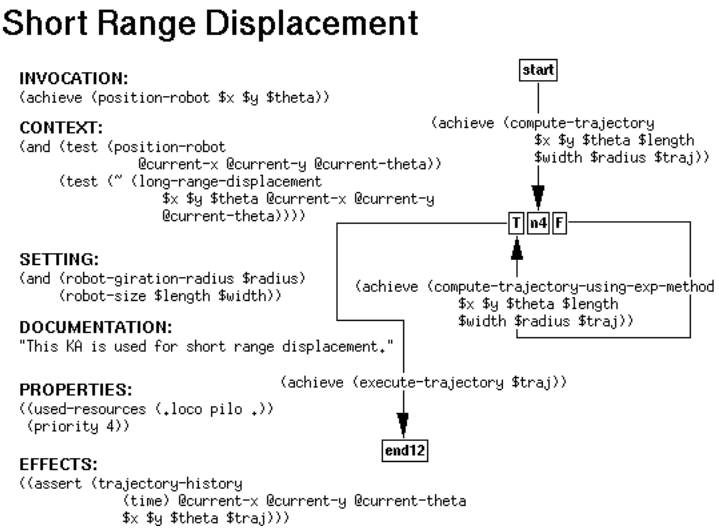

Figure 11: A Plan for Short Range Displacement

respectively written `(assert c)` and `(retract c)`.

The body part of the procedure is built using the different types of goal presented above, which are the "basic" instructions. As illustrated with our two examples, one can build "text" or graphic procedures. In text procedures, the execution starts from the first instruction and proceeds from then, following the standard programming structures such as if-then-else, while, do-while, parallel (//), goto, etc. In graphic procedures, the execution starts from the `start` node and proceeds to the next nodes achieving the goal labeling the edge. When a goal appears as the condition of a conditional instruction (such as an if-then-else, a while or before an "if-then-else" node (see node `n4` on Figure 11)) the condition is satisfied if the goal can be achieved, it fails otherwise. On the other hand, when a goal is not in a conditional position, its failure leads to the complete procedure failure.

The set of PRS procedures in the Supervision system not only consists of procedural knowledge about the robot (Figures 10 and 11), but also includes *meta-level* procedures – that is, procedures able to manipulate applicable procedures, goals, and tasks of PRS itself. Meta-level procedures enables methods for choosing among multiple applicable procedures, or for managing mutual exclusion on critical resources. To achieve such objectives, these meta-level procedures make use of information about plans, goals, facts that is contained in the database or in the *properties* slot of the procedure. For example, a meta procedure may ensure that any event (i.e. not goal) invoked procedure, is intended and may guarantee some kind of priority mechanism if particular properties hold for the procedure to intend.

The PRS language provides a general frame for acting on and maintaining a declarative representation of the environment. It is well suited for implementing components of the decisional level of our architecture, in particular the supervision components.

Partial Plan Representation In PRS, each plan/procedure is self-contained: it describes in which condition it is applicable and the goals it achieves. It usually contains in its "body" tests which condition the proper posting of its subgoals while leaving to the interpreter (and the meta-level procedures) the choice of the adequate plan to try to satisfy each posted subgoal. This is particularly well adapted to context based task refinement and to a large class of robot tasks which can be viewed as incremental. A typical task of this type is navigation in a partially known and/or dynamic environment. For example, in outdoor navigation experiments, a number of tests and actions must be performed before the robot begins to plan its motion. Nevertheless, the choice of the motion planner used (2d or 3d) is left to the interpreter and possibly to meta-level procedures which decides which method is the best in the current situation (e.g., flat or rough terrain).

Event and Goal Driven Behavior Procedures can be triggered upon occurrence of events or posting of goals. This is a key feature for implementing a periodic monitoring through a set of situation driven procedures while refining and executing a plan as provided by the planner. A convenient way to interface the supervisor and the planner is the PRS database. This allows for example to express "execution modalities" as facts which will modify the execution of plans, inhibit or awaken others.

The notion of goal in PRS is rather strong as it represents an objective the interpreter tries to satisfy by any means, i.e. by trying one after another the procedures which unify with it (the applicability of the procedure is reevaluated after each unsuccessful attempt). As a consequence, a goal is considered as failed after *all* procedures for each valid unification have been tried, and have failed.

An event may make some procedures applicable, but these procedures do not pursue an explicit goal. They merely reply to events and produce subgoals to be achieved without any explicit top level objective. Moreover, events represent new pieces of information which are usually stored in the database for further reference and for updating the state of the world as seen by the system.

Advanced Reasoning The meta level reasoning available under PRS provides a powerful mechanism to control the PRS main loop. Currently, meta level reasoning is mainly used in the procedure selection part of the PRS main loop[2].

Real-time Aspects The use of Kheops as part of the executive, alleviates the burden of PRS and exempts it of providing guarantee on response time. Nevertheless, it is important that the PRS main loop, at the decision level, can guarantee an upper bound on reaction time, i.e. the time it takes to perform one loop, which is therefore the time an event will wait before being taken care of.

It turns out that if T_{Pars} is the time to parse a procedure invocation, T_{Int} is the time to intend a procedure and T_{Choose} is the time to choose among applicable procedures, the

[2]However, it can easily be extended to other parts of the PRS interpreter (for example to react to task graph changes or to goal failure).

cycle time CT_i depends on the previous cycle time CT_{i-1}, the frequency of event arrival μ_{i-1} during the cycle C_{i-1}, the time $T_{Exec}(i)$ taken by the action (if any) executed during cycle C_i and a constant value $(T_{Int} + T_{Choose})$ (in the sense that one can find an upper bound constant value).

One can show that the cycle time of PRS is: $CT_i = (CT_{i-1} \times \mu_{i-1} \times T_{Pars}) + T_{Int} + T_{Choose} + T_{Exec}(i)$
An upper bound for all i is: $CT_i \leq \mu_{Max} \times T_{Pars} \times CT_{i-1} + T_{Exec}^{Max} + T_{Int} + T_{Choose}$
The first observation we can make is that to have a constant bound on all the CT_i we need: $\mu_{Max} \times T_{Pars} < 1$.
Interpreting this constraint means that if this value $\mu_{Max} \times T_{Pars} \geq 1$ then the value of CT_i diverges and it becomes impossible to guarantee an upper bound on the reaction time. However, if $\mu_{Max} \times T_{Pars} < 1$, then we can show the existence of a constant upper bound on CT_i which is: $\frac{T_{Exec}^{Max}+T_{Int}+T_{Choose}}{1-\mu_{Max} \times T_{Pars}}$. See [29] for a more detailed analysis.

From this bound on reaction time, the user can derive or implement other complex and advanced temporal properties, such as priority mechanisms, deadlines, and so on. For example, using meta level procedures, it is easy to implement a mechanism which can guarantee that a procedure with a particular property is intended as root of the task graph (therefore executed before any other procedure).

User Interface and Miscellaneous Features PRS interface provides mechanisms to the user to add new facts or goals, to examine the various tasks intended, to check/change the content of the database, to follow the execution of particular procedure, and to load or delete procedures on the fly.

PRS provides a number of mechanisms to allow interaction with the real world: a communication library over sockets, but also more sophisticated means to provide high level interprocess communications and shared memory services between PRS and low level modules [22].

Depending on the application, the user can thus give new missions or new orders to a robot by adding new facts, new goals or by sending messages to the PRS kernel from an external user interface.

The PRS code executes on various workstations but also on-board our mobile robots on 680X0/PPC boards running under VxWorks.

5.3 Task and Mission Planning

Several planners are needed in the robot architecture we advocate for here. Some of these, such as the path and trajectory planners [11], a manipulation planner [4], the viewpoint and sensor modality planners [34] are modules of the functional level, already existing or to be added into it. The task and mission planner belongs to the decision level. It is a system queried by the supervision. It has to deal with time explicitly, not only in task duration, but also in parallel activities within compound tasks, and in various temporal constraints between conditions (before or while a task proceeds) and effects of a task. It should deal,

at planning time, with the predictable part of a dynamic environment, e.g., contingent change not under the robot control such as day/night cycles, resource availability profiles, or expected events. It has to manage the resources of the robot, preferably while planning, not as a subsequent resource allocation and scheduling step, after the tasks have been chosen and constrained. It should offer to the designer a powerful representation for specifying and programming models of the robot tasks at the abstraction level required for planning.

There are very few planners meeting these requirements, among which in particular O-Plan [45], and the one we have developed, IxTeT [33]. O-Plan relies on the hierarchical task network representation (HTN), whereas IxTeT favors the partial-plan space approach. This means that the former, which proceeds by refining given skeleton of plans (tasks at various levels of abstraction), can be more efficient, but it requires significantly more programming than IxTeT does. This can be a further benefit for IxTeT in the perspective of enhancing the models of the planner through automated learning.

Representation Properties of the world are described by a set of **multi-valued attributes** and a set of **resource attributes**. Each attribute is a k-ary mapping from some finite domain into a finite range. A resource can be a single item, *unsharable*, or an aggregate resource that can be shared simultaneously between different actions seeing that its maximal capacity is not exceeded.

IxTeT is based on a reified logic formalism. Attributes are temporally qualified by the predicate *hold*, which asserts the persistence of an attribute value over an interval, and the predicate *event*, which states an instantaneous change of values. Resources are expressed by the predicates *use*, which represents a borrowing of a quantity of a resource over an interval, *consume* and *produce* which state the consumption or production of a quantity of a resource (see [33] for details).

IxTeT *time-map manager* [25] relies on time-points as the elementary primitives. Time-points are seen as symbolic variables on which temporal constraints can be posted. A *time-map manager* propagates *numeric* and *symbolic constraints* (precedence, simultaneity) to ensure the global consistency of the network. It answers queries about the relative position of time-points. Management of atemporal variables is achieved through a *variable constraint manager*. We consider variables ranging over finite sets and propagate *domain restriction*, *equality* and *inequality constraints*. Constraint propagation on atemporal variables is achieved through classical CSP techniques.

A deterministic planning operator, called a **task**, is a temporal structure composed of: a set of sub-tasks; a set of events describing the changes of the world induced by the task; a set of *hold* assertions on attributes to express required conditions or the protection of some fact between two events; a set of resource usages; and a set of temporal and binding constraints on the different time-points and variables of the task.

The initial plan is a particular task that describes a problem scenario, that is: (i) the initial values for the set of instantiated attributes (as a set of explained events); (ii) the expected changes on some contingent attributes that are not controlled by the planner (as

a set of explained events); (iii) the expected availability profile of the resources ; (iv) the goals that must be achieved (usually, as a set of assertions); and (v) a set of temporal constraints between these elements.

The Planner Algorithm IxTeT explores a search tree of partial plans. The root is the initial plan; branches represent some tasks or constraints inserted into the current plan in order to solve one of its **flaws**. Three kinds of flaws are distinguished:
– *pending subgoals* are events or assertions that have not yet been established; resolvers for a pending subgoal consist in inserting an event and an assertion that protects the attribute value (causal-link); such an establisher can already be in the plan or it may need the insertion of a new task;
– *threats* are possibly inconsistent events or assertions; when a threat is detected, it can be solved by adding temporal constraints (promotion, demotion) or variable constraints (separation, fusion of two assertions);
– *resource conflicts* are detected as **minimal critical sets** thanks to an efficient algorithm; resolvers include precedence constraints, inequality constraints between resource allocations, or insertion of resource production tasks.

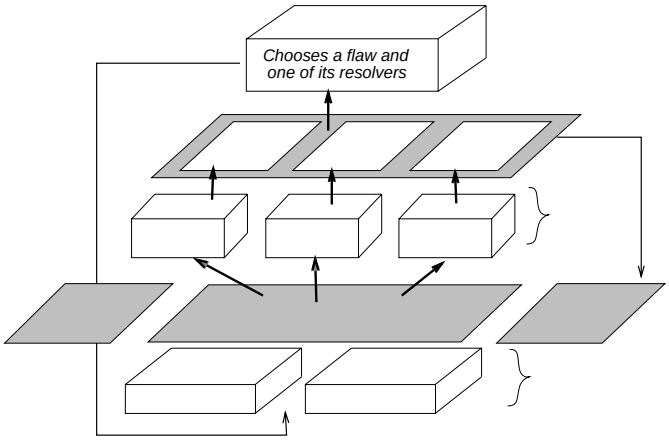

Figure 12: *the IxTeT Planner*

Three analysis modules in IxTeT are dedicated to the detection of the different flaws and the computing of their resolvers for a given node of the search tree (Figure 12). The control algorithm detects all flaws in a partial plan; if there is no flaw a solution is found; otherwise a flaw is selected, possible resolvers for this flaw are listed, one is non-deterministically chosen (with a backtrack to a previous choice if there is no resolver); it is inserted into the partial plan on which the algorithm proceeds recursively. IxTeT is complete in the sense that if there exists a solution, it will find it [24]. This is the case since a complete set of resolvers is computed for every flaw and the choice of a resolver in the search tree is a backtrack point.

For a given flaw, the choice of its resolver is done according to a **least-commitment** strategy: the best resolvers are those that constrain the less the current plan. The **com-**

mitment of posting a resolver on the current partial plan, is a local estimate of the number of solution plans that are eliminated by this resolver [33].

In a partial plan, the number of flaws to analyze may be very large. At a given level of the planning process, some flaws may be much more relevant than other ones. IxTeT uses a hierarchy on the different attribute names (and, as a consequence, on the different flaws) to structure the search space. This hierarchy verifies the **ordered monotonicity property** [32]: *at a given abstraction level, the resolution of a flaw creates only new flaws belonging to the current or less abstract levels.* Our hierarchy presents two interesting features [23]. It can be automatically generated from the description of the planning operators by analyzing the conditions and the main effects of the tasks. It is a hierarchy dynamically defined from the least-constrained partial-ordered of attributes meeting the ordered monotonicity property. The control tackles the flaws opportunistically according to this order.

We have also developed a set of operations on plans generated by IxTeT which enable to merge two or several plans. A *union operation* merges, whenever possible, two plans into a single consistent one. The two set of tasks are re-shuffled and further constrained to take into account their shared resources and other common constraints. An *insertion operation* adds to a given plan a set of goals (eventually those of a plan which could not be directly merged with the first one) and re-plans starting from this partial node. These operations have been motivated mainly for performing distributed planning over several robots. But they have been found to enhance significantly the performance of IxTeT through goal decomposition and incremental planning.

6 Examples and Analysis

Several instantiations of the architectural concepts presented above have been implemented with a progressive refinement of the architecture itself as well as an improvement of the different software tools involved.

The robot described in [17] demonstrated the use of a decision level including IxTeT planner on top of a task refinement layer. The use of IxTeT was dictated by the need to (re)plan the tasks and their partial order depending on the availability of resources in an in-door environment (composed of rooms connected by doors) which was only known to the robot at a topological level.

Another illustrative and demanding implementation involved a rough terrain robot performing navigation tasks in an an unknown environment [16]. This implementation involved essentially a single type of tasks; a high level planner was not necessary. However, it was particularly demanding in terms of task refinement depending on a context which was incrementally discovered by the robot itself. At each step, the robot had to choose between several possible environment models to be built, several motion planners to invoke as well as several navigation modes and localization schemes.

One recent instantiation concerns the development of an autonomous robot endowed

with cooperative abilities for load transfer in a structured environment [3][3]. The reason why we choose to present it here instead of the other instantiations mentioned above comes from the fact that it represented a very important amount of work involving several persons and resulting in a coherent and robust system. What is of interest here is to show how the generic architecture described above has served as a conceptual and practical tool for implementing such a system, and how the specific features of an application have been translated in terms of architectural choices.

6.1 An instance of the architecture

The global system is composed of a Central Station and a fleet of autonomous mobile robots (Figure 13). Each robot has an a priori knowledge of the environment. High level missions are produced by the Central Station and sent to robots. It is then up to the robots to refine their missions, to plan their actions and trajectories and to coordinate these actions and motions with the other robots. In particular, these coordinations occur in crossings, in lanes when unexpected obstacles require the robot to move in the opposite lane, and in open areas where robots need to synchronize their trajectories.

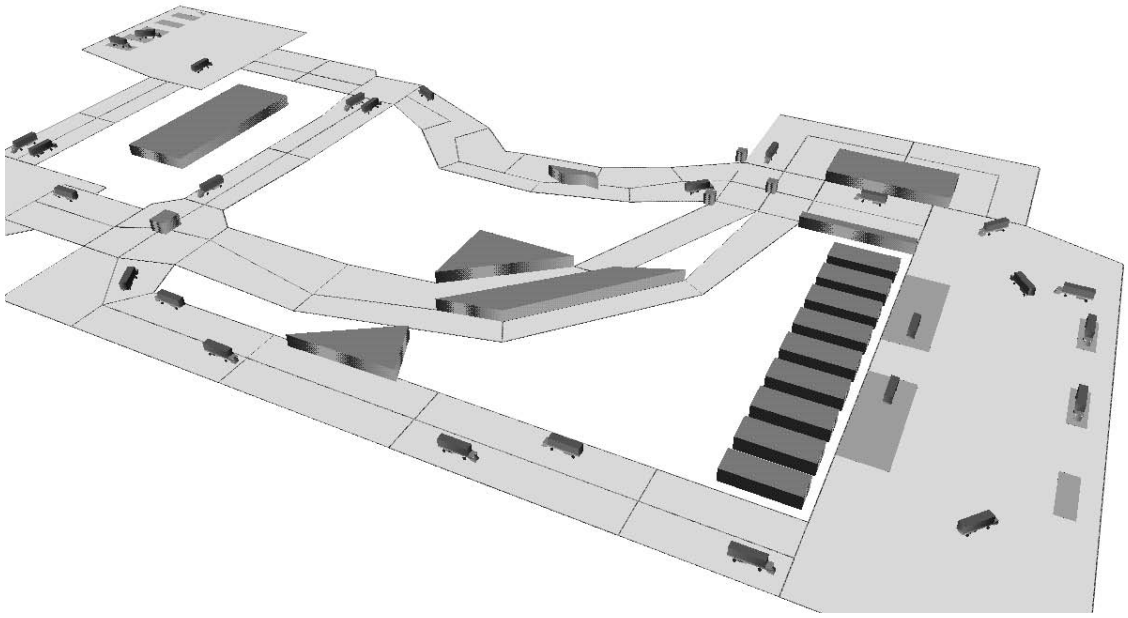

Figure 13: Simulation of a fleet of 30 mobile robots for container transport

The implemented architecture of such mobile robots is directly derived from the generic control architecture presented above.

[3]An application of this work was the MARTHA European ESPRIT III Project No 6668. "Multiple Autonomous Robots for Transport and Handling Applications".

6.1.1 The Decision Level

As discussed in §5, the decision level may be split into several layers corresponding to different representation abstractions. Since here we have a multi-robot coordination requirement, we have added a third layer, called "Coordination Layer" which transforms the task plans as produced by the Task Layer into coordination plans valid in a multi-robot context (Figure 14). This third layer implements a cooperative scheme - called "Plan-Merging" - where each robot autonomously and incrementally builds and execute its own plans taking into account the multi-robot context [1]. It is built on the same concept of planner/supervisor pair.

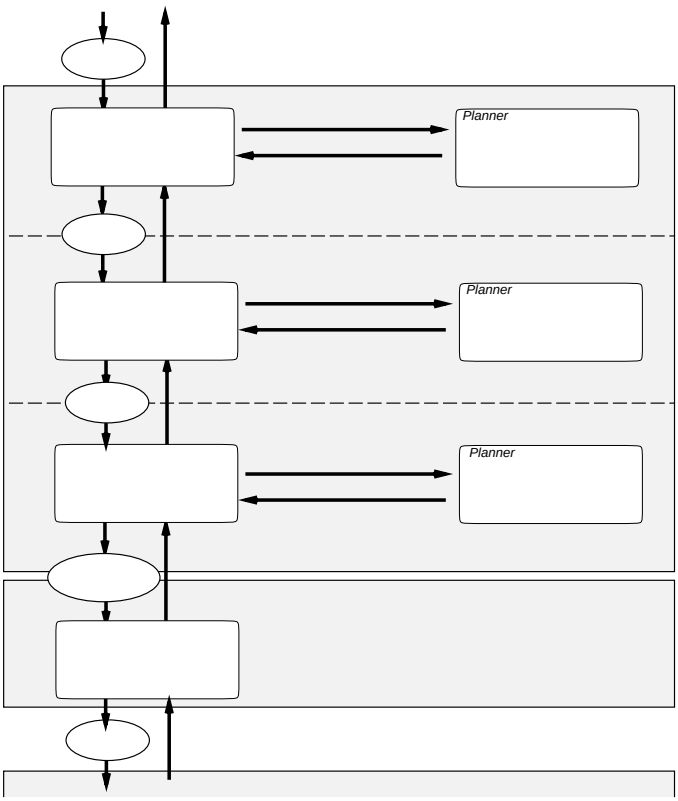

Figure 14: The robot decision level

a. The Mission Layer: From time to time, the Central Station sends a new mission to the robot: (transfer-load (station 1) (station 3)) or (park (station 5)). Mission planning, for this application, is mainly (and simply) a search in the topological graph representing the route network. There was no need to use a sophisticated planner like IxTeT[4]. In fact, we have chosen to implement temporal reasoning at an even higher level: at the Central Station ([47]) which deals with mission allocation to robots.

[4]Note that, other applications calling for explicit reasoning on temporal constraints have included an instance of our planner IxTeT on-board of the robot ([17]).

b. The Task Layer: The task layer receives sequences of tasks from the Mission Layer (Figure 15 illustrates a typical task plan). Task refinement is performed through context dependent instantiation of "plans skeletons". A task is first refined as if the robot was alone. The resulting plan is a sequence of actions (including planned trajectories) annotated with cell entry and exit monitoring operations which will be used to maintain the robot execution state and to synchronize its actions with other robots. The Task Supervisor is in charge of controlling the execution of such a plan. If a plan fails to achieve a particular goal, alternative plans are refined and attempted.

```
(task-plan (
   (action 1 (goto (station 1)) (using (lane 10)))
   (action 2 (dock))
   (action 3 (put-down))
   (action 4 (un-dock))
   (action 5 (goto (station 3)) (using (lane 12) (lane 8)))
   (action 6 (dock))
   (action 7 (pick-up (container 5)))
   (action 8 (un-dock))
   (action 5 (goto (end-lane 0)) (using (lane 9) (lane 0)))))
```

Figure 15: A task sequence example

```
(coordination-plan (
    (exec-p 1 (report (begin-action 1)))
    (exec-p 2 (wait-exec-event r3 9))
    (exec-p 4 (monitor (entry (cell 4))))
    (exec-p 8 (exec-traj 0))
    (exec-p 3 (wait-exec-event r7 48))
    (exec-p 5 (monitor (entry (cell 5))))
    (exec-p 6 (monitor (exit (cell 14))
              (signal-exec-event r4 17)))
    (exec-p 7 (monitor (exit (cell 4))))
    (exec-p 9 (exec-traj 1))))
```

Figure 16: A coordination plan

c. The Coordination Layer: produces and controls "coordination plans". It performs Plan-Merging operations and manages the interactions with the other robots (exchanging coordination plans and events). Indeed, the plans produced by the Task Layer are incrementally validated in a multi-robot context by the Coordination Layer through the use of a Plan-Merging protocol. The result is a "coordination plan" which specifies all trajectories and actions to be executed, together with all events to be monitored and sent to another robot or to be awaited from another robot. For example, Figure 16 represents a coordination plan obtained after a refinement and a coordination of the action 1 of the task plan presented in Figure 15. Note that an execution failure reported by a robot from which an event is awaited will cause a new Plan-Merging operation to be performed.

The Execution Control is in charge of the interpretation and the execution of coordination plan. As a result, it is responsible of most interactions with the functional level. Besides all actions and monitors included in the plan, it also monitors and reacts to a number of critical events, such as unexpected obstacles in its path, or its own status (battery or fuel level), failure reports from the different modules, as well as messages sent by the other robots (information about synchronization events or plan failures).

The decision level has been implemented in PRS. The seven components run as tasks inside PRS (§5.2) and communicate through its database. The data structures used for representing coordination plans as well as the algorithms for their manipulation have been borrowed from IxTeT. The communication with the Functional Level is performed through a set of C functions linked to PRS; they provide a mean for sending requests to modules and for transforming data produced by the various modules (replies to requests or posters contents) into facts in PRS database.

6.1.2 The Functional Level

The functional level implements all robot basic capabilities in sensing, acting, communication and computing. Figure 17 shows the functional level, for the presented application, including 8 modules, their client/server relations and 3 exported data (posters). Each module is briefly described below.

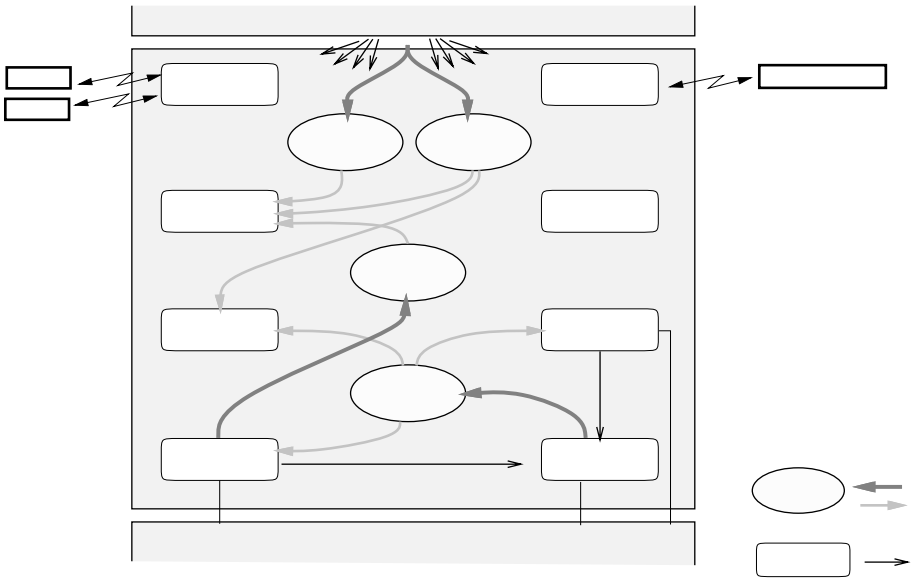

Figure 17: Architecture and interactions of the functional level

The Motion Planner Module It computes not only feasible trajectories but also synchronization events between different robot trajectories. It is composed of a Topological Planner, a Geometrical Planner and Multi-Robot Scheduler.

– **The Topological Planner** performs a search in the graph of cells in order to determine the set of cells to be used for a given motion task. The selection of cells may be done in two modes: a *Local Obstacle Avoidance* mode that selects only the cells which correspond to the nominal way of traversing lanes or crossings, and an *Extended Obstacle Avoidance* mode which is invoked, for example, when a major obstacle forces a robot to leave its current lane and to use cells belonging to a "parallel" lane.

– **The Geometrical Planner** computes non-holonomic paths using techniques similar to those described in [35]. When used in a "multi-robot" mode, it produces a path which avoids the last positions of the other robots.

– **The Multi-Robot Scheduler** determines, along each trajectory, the positions where the robot should signal a *trajectory synchronization event* to another robot, and the positions where it should stop and wait for synchronization. Figure 18 illustrates trajectory synchronizations: W_j^i stands for a position where robot R_i should stop and wait that robot R_j has passed position S_i^j.

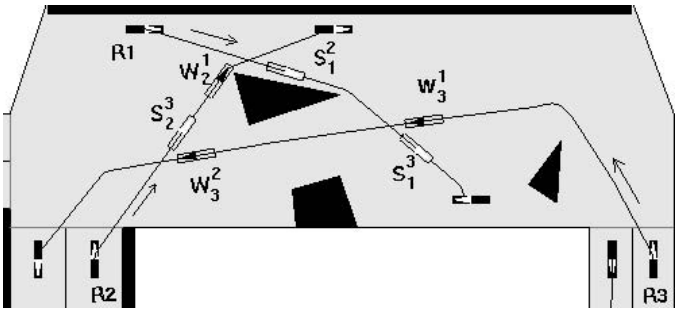

Figure 18: The result of a Plan-Merging Operation at trajectory level

The Motion Execution Module. This module has a permanent activity which consists in position computation from proprioceptive sensors (odometers, gyroscope) and in the feedback control on the robot position whose current value is exported in the `POSITION` poster. Besides, it executes trajectories composed of a sequence of segments and arcs of circle, and a polygonal bounding area in which the robot should remain (cell, lane). The trajectories are smoothed with clothoids. It is also able, for multi-robot coordination purposes, to monitor the curvilinear abscissa while executing a motion (see §6.1.2).

The Local Obstacle Avoidance Module. It monitors obstacles with range sensors (ultra-sonic sonars or laser range finder) and filters the trajectories before transmitting them to the Motion Execution module. In order to avoid unknown obstacles, the robot can be stopped or the trajectory can be slightly modified. However, the trajectory should remain in a bounding area specified by the Robot Supervisor. If the robot is blocked, the motion requests are ended. The Robot Supervisor will then produce a new planned trajectory taking into account the new obstacles (see the example in §6.2).

The External Perception Module. The role of this module is twofold [39]: (1) updating the robot position using exteroceptive data (range sensors) and performing landmarks based re-localization; and (2) building and maintaining a local map which may be provided, upon request, to the Motion Planner, through the `PERCEIVED_OBSTACLES` poster.

The Position Monitoring Module. This module allows to maintain the set of resources (cells, lanes, areas) occupied by the robot and to monitor the entry and the exit of these resources. This information is necessary for the Plan-Merging activity performed by the Robot Supervisor,

The External Communication Modules. The communications with the Central Station and with the others robots are achieved by two distinct modules called Inter-Robot Communication (IRC) and Central Station Communication (CSC). A message between robots can be dedicated to one specific robot or broadcasted to all robots in its vicinity (the IRC is assumed to have a limited range).

6.2 Illustrative Runs

This architecture has been used on 3 Hilare mobile robots (Figure 19) to demonstrate advanced autonomous features including non-holonomic motion planning, environment modeling, sensor-based obstacle avoidance, and decentralized cooperation schemes at mission and trajectory levels. The complete architecture has been taken on board each robot. It includes the decision level, the 8 presented modules and about 12 more basic modules distributed on five 680x0 cpu-board running VxWorks.

It has also been used to demonstrate the same capabilities in simulation (Figure 13) involving up to 30 robots (each robot emulated by a Unix workstation).

Figure 19: The three Hilare robots in action

We give here a short description of motion planning and execution control as it performed by the Hilare robots. This illustrates how execution modalities allow a flexible and efficient plan adaptation to run-time contingencies.

The environment model is known to the robots and is used for mission planning and coordination. Each robot makes use of this data to plan and execute its motions taking into account an estimation of the maximum value of its position uncertainty (which may grow between two re-localizations) and its local environment as perceived by its sensors. Coordination planning is performed incrementally by each robot in anticipation for a short period of time ahead (several seconds). However, to permit a proper avoidance of unknown obstacles, the planner produces together with planned trajectories a set of of polygonal regions which embed the trajectories and which are to be allocated as resources by the robots. The frontiers of such regions are considered as virtual obstacles which can be crossed only if re-negotiated with the other robots.

This defines several levels of reaction to unknown obstacles:

• level 1 (Figures 20 and 21): sensor-based obstacle avoidance within virtual frontiers;

• level 2 (Figure 22): level 1 reaction has failed; re-planning is performed without crossing virtual frontiers and hence without re-negotiating plans;

•level 3 (Figure 23): level 2 reaction has failed; re-planning is performed with new virtual frontiers leading to re-negotiate the robot plan with other robots.

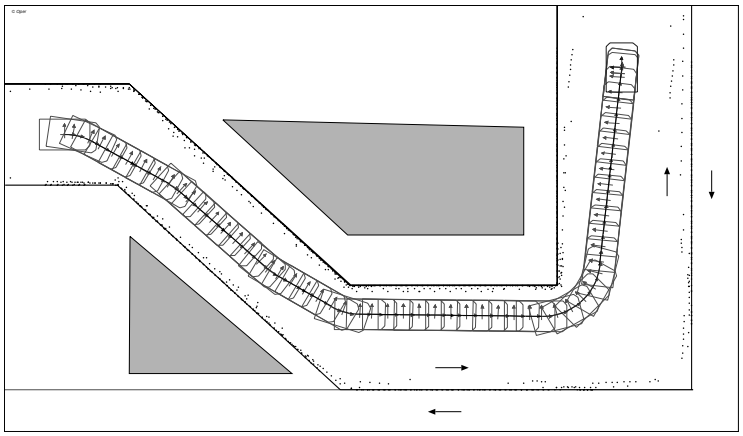

Figure 20: **First situation:** The robot has allocated a "corridor" which frontiers are *virtual* obstacles materialized through virtual sonars. During the execution it monitors the possible obstacles with sonars (the dots). The robot can execute its planned trajectory without problem.

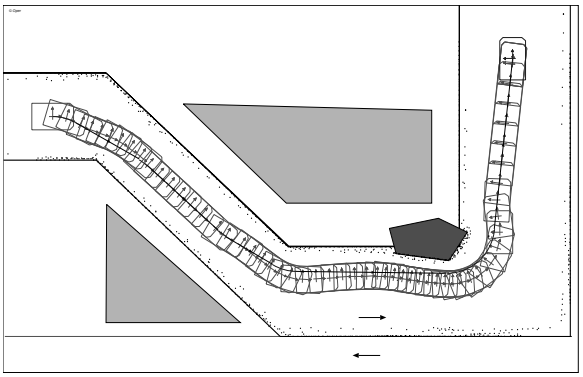

Figure 21: **Second situation:** A local obstacle has been detected and the planned trajectory is *dynamically* modified (with a potential-field algorithm) while remaining inside the boundaries thanks to the virtual obstacles.

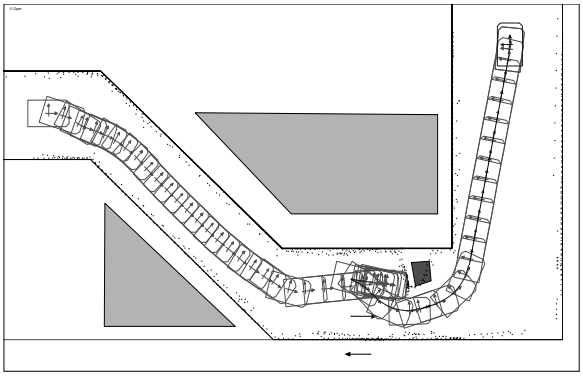

Figure 22: **Third situation:** The obstacle is more bulky: the robot was not able to dynamically avoid it. However, the information acquired during the execution allows to add the obstacle in the model and to plan a new path around it.

7 Related Work

The technical literature reports on a wide spectrum of approaches for the design of robot architectures and on analyzes of and comparisons between them (*e.g.,* [6, 13, 20, 26, 46]). We will not overview here all proposed architectures, but rather try to indicate the main trends.

The idea of a modular and programmable functional level is present in the Logical Sensor Specification suggested by Henderson [27] as a model of a physical sensor and its

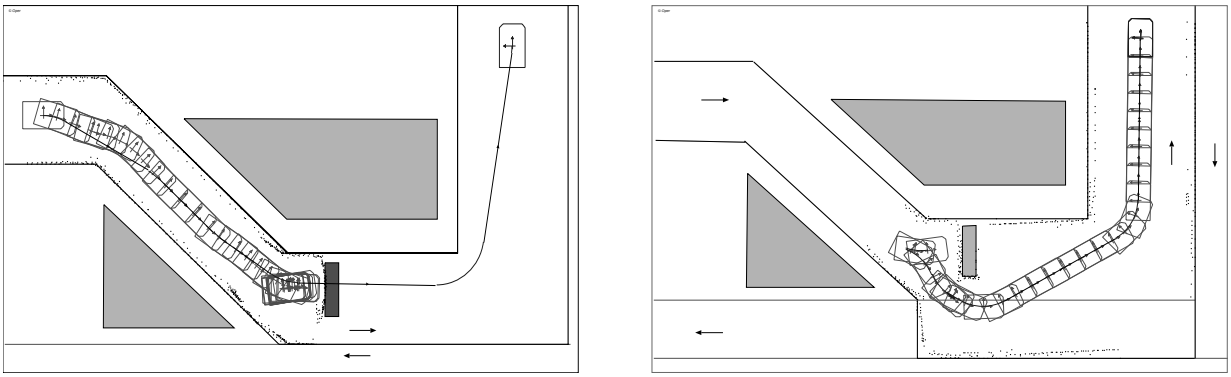

Figure 23: **Fourth situation:** The robot is once again blocked, but no trajectory was found to reach the goal while remaining inside the allocated corridor: an extended cell (from the other one-way corridor) has been negotiated and allocated by the robot.

control capabilities. He later integrated these LSS in purely behavioral robot systems.

In [44] Stewart *et al.* emphasize the concept of reusable software. A framework, organized on a global database, is proposed for integrating real-time software modules. Module connections are based on automata theory. A modular organization supervised through an automaton is also proposed in [40]. The ORCCAD [42, 19] modular system, focused on the scheduling of servo-actions, is based on ESTEREL synchronous language and permits to demonstrate some logical and temporal properties.

All these systems, initially dedicated to manipulators, share the idea of a set of functional modules implementing the basic robot capabilities which are combined to execute more complex tasks. The connections miss more or less some programmability or even reconfigurability due to a lack of an independent and generic organization of the modules or a limited accessibility to those modules from the different entities of the global architecture, if any. These limitations make difficult the integration of reasoning capabilities.

Several authors addressed the problem of building a complete control architecture for autonomous agents. Most approaches emphasize either a reactive or a centralized structure. A few resulted in structures trying to integrate deliberation with reaction.

At one extreme Brooks [14] builds stimuli driven "robot beings". The well known subsumption architecture organizes the robot system into layers of behaviors with an inhibition mechanism. The robots built with this architecture are clearly the data driven part (although not necessarily in an efficient manner), but have no capacity of action planning.

This bottom-up organization of finite automata is criticized by several authors, for example in [43] where a "Supervenience architecture" is proposed instead: it adds a goal-driven mechanism enabling a hierarchical task-network planner to adapt the behavior to the context and to new goals. Similarly, Arkin's AuRA (Autonomous Robot Architecture) binds a set of reactive behaviors to a simple hierarchical planner that chooses the appropriate behaviors in a given situation [5].

Firby [21] introduced *reactive action packages* (RAP) which are programs running until either the goal is reached or a failure occurs. The RAP interpreter refines tasks into more

primitive commands and controls activation and conflict resolution. RAPs describe how to achieve a given task. Their semantics are strictly oriented towards task achievement and sequencing subtasks. RAP somewhat compares with PRS (§5.2). One difference is that selection of the appropriate method within a given RAP does not consider general strategies, as implemented in our case in meta-procedures.

Going further towards the integration of planning and reacting, Lyons and Hendriks [38] developed a system, called RS, where planning is seen as a permanent adaptation of a reactive process. The later is a set of rules which is modified according to the context and goals. This has been tested for the design of an assembly robot [37]. Similarly, Bresina [12] proposes in the ERE system a planner which is able to synthesize new situated control rules when a failure situation is met.

3T [10] is a three-layer architecture. It comprises a set of "skills" (comparable in a way to our modules) and uses the RAP system in one of its layer to sequence them. One layer is a planning system that reasons on goal achievement including timing constraints. As a global architecture, 3T is comparable to our own. However, when examining the details of each layer and their interactions, many differences appear. 3T is mainly based on a planner/sequencer hierarchy. The planner-supervisor structure which is the rationale of our decision level is very different because it enables reaction *during* planning.

The Task Control Architecture developed by R. Simmons [41] organizes processing modules around a central controller which coordinates their interactions. A goal is decomposed into a task tree with subgoals which are achieved by the decentralized modules. All communication is supported by the centralized control. TCA provides control mechanisms for task decomposition and takes into account temporal constraints in task scheduling. It is probably the closest work to PRS (§5.2). TCA is however, to our knowledge, not integrated into a global framework for action planning and real-time control as our own system.

More recently, other research laboratories have found interest in using the PRS like approach for mobile robot applications. In [36], the authors describe an implementation of procedural reasoning (called UM-PRS) to control an outdoor environment vehicle.

All these approaches have their own advantages and drawbacks, in terms of the principles on which they rely. But we believe the architecture presented in this paper is the more complete and integrated.

8 Conclusion

This paper proposes a generic architecture for autonomous robots. The architectural concepts guiding our design choices have been described and justified with respect to the properties that we consider as being required in an autonomous robot. Autonomy in a rational behavior can be evaluated by the robot's efficiency and robustness in carrying out various tasks in a partially known environment. It calls for properties such as programmability, reactivity, adaptability, or evolutiveness. Above all, it requires a suitable interaction between deliberation and action.

Following a discussion of the architectural concepts and the rationale supporting the

proposed organization, the paper describes in details the representations and tools, practical and conceptual, that we have developed or chosen for designing and implementing the components at the various levels of the architecture. These representations and tools are essential for making the generic architecture feasible and for endowing it with the required properties, in particular: the programming and integration easiness; the consistency of the implemented or synthesized code; and the reactivity of the critical components, at different constraint and guaranty levels.

The paper exemplifies and analyzes the proposed architecture through one of its instances that we developed and integrated within our robot platforms for a complex multi-robot cooperation in transportation problems within a structured environment. Various experiments are described to illustrate, through the behavior of the robots, some of the nice properties of the architecture.

Finally, let us stress one of the quality exhibited by the proposed architecture and the accompanying tools, that is the ease of design, programming and integration. Indeed, research platforms like ours are particularly demanding for such a quality since several workers interested in different robotics aspects, ranging from sensing, environment modeling and interpretation, motion, and manipulation, have been able to develop their ideas, to integrate their algorithms and to test them within the proposed architecture.

In conclusion, the evidence from our experience on the integration of a wide range of work, on several mobile platforms, supports our belief that the architecture proposed here is able to fulfill the requirements foreseen for the design of intelligent robots. However, the knowledge representations and tools actually implemented and available within this architecture are far from being complete with respect to our ambition for such robots. Several extensions have been mentioned along the technical presentation of the 3 levels, such as the partial synthesis of the executive rules from the formal description of the functional modules. Let us stress here three open issues to which significant research needs to be devoted, that is more extensive management of uncertainty, learning capabilities, and man-robot interactions.

Currently, we mostly deal with uncertainties in world models and sensor data. We are able to plan motion strategies taking into account such inaccuracies [11, 31] and part of its further implications (e.g., a way that could have been crossed is now blocked [18]). However, hypothetical deliberation (on the basis for example of non-deterministic supervision procedures or planning tasks) would improve the robustness of the robot behavior. The corresponding extensions are certainly compatible with our architecture.

The learning capabilities of our robots are today limited, mainly to environment modeling [16]. Ease of programming and evolutiveness would require self improvement capabilities from experience. One may think about the synthesis of new executive rules, new supervision procedures or task models. Mechanisms for logging in and analyzing past experiences have to be added to the architecture, eventually learning goals have to be considered and planned for. Many learning techniques for abstract environments are known, and can be useful in robotics, but wide areas of research are open.

Finally, the autonomous robot has to interact friendly with roboticiens, with users, with persons active in the environment. A broad range of capabilities are also needed,

many of which not specific to robotics (e.g. MMI through speech and vision, or interactive problem solving), others, such as extended reality, coordinated control or task cooperation are active issues in our field.

References

[1] R. Alami. A multi-robot cooperation scheme based on incremental plan-merging. In *Hirzinger, Giralt (Eds), Robotics Research: The Seventh International Symposium, Springer Verlag*, 1996.

[2] R. Alami, R. Chatila, and B. Espiau. Designing an intelligent control architecture for autonomous robots. In *ICAR'93*, pages 435–440, Tokyo, Japan, November 1993.

[3] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert. Multi Robot Cooperation in the Martha Project. *IEEE Robotics and Automation Magazine: Projects funded by the Commission of the European Union*, ?(?), March 1998. To appear.Also available as LAAS/CNRS Technical Note 96392.

[4] R. Alami, J.P. Laumond, and T. Siméon. Two manipulation planning algorithms. In *The First Worshop on the Algorithmic Foundations of Robotics, A.K. Peters Pub., Boston, MA*, 1994.

[5] R. C. Arkin. Motor Schema-Base Mobile Robot Navigation. *International Journal of Robotics Research*, 1990.

[6] P. Baroni, G. Guida, S. Mussi, and A. Vetturi. A distributed architecture for control of autonomous mobile robots. In *ICAR'95*, pages 869–877, Spain, September 1995.

[7] A. Benveniste and P. LeGuernic. Hybrid dynamical system theory and the signal language. *IEEE Trans. Automatic Control*, 35(5):535–546, May 1990.

[8] G. Berry and G. Gonthier. The synchronous programming language esterel: design, semantics and implementation. Technical Report 327, INRIA, 1985.

[9] S. Betgé-Brezetz, P. Hébert, R. Chatila, and M. Devy. Uncertain map making in natural environment. In *IEEE ICRA'96, St Paul, (USA)*, 1996.

[10] R. P. Bonasso, D. Kortenkamp, D. Miller, and M. Slack. *Experiences with an Architecture for Intelligent.* Michael Wooldridge, Joerg P. Mueller, and Milind Tambe, Springer-Verlag, 1995.

[11] B. Bouilly, T. Siméon, and R. Alami. A numerical technique for planning motion strategies for a mobile robot in presence of uncertainties. In *IEEE ICRA'95, Nagoya (Japan)*, May 1995.

[12] J. L. Bresina. Design of a reactive system based on classicla planning. In *Foundations of automatic Planning*, pages 5–9, 1993.

[13] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[14] R. A. Brooks and A. Flynn. Robot beings. In *IEEE IROS'89, Tsukuba (Japan)*, 1990.

[15] R. F. Camargo, R. Chatila, and R. Alami. Hardware and software architecture for execution control of an autonomous mobile robot. In *International Conference on Industrial Electronics, Control, and Instrumentation (IECON'92), San Diego (USA)*, 1992.

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

[17] R. Chatila, R. Alami, B. Degallaix, and H. Laruelle. Integrated planning and execution control of autonomous robot actions. In *IEEE ICRA'92, Nice, (France)*, 1992.

[18] R. Chatila, S. Lacroix, T. Siméon, and M. Herrb. Planetary exploration by a mobile robot : Mission teleprogramming and autonomous navigation. *Autonomous Robots Journal*, 2(4):333–344, 1995.

[19] E. Coste-Maniere, B. Espiau, and D. Simon. Reactive objects in a task level open controller. In *IEEE ICRA'92, Nice, (France)*, 1992.

[20] 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'93*, pages 267–274, Atlanta (USA), May 1993.

[21] R. J. Firby. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems, Chicago IL*, June 1994.

[22] S. Fleury, M. Herrb, and R. Chatila. Design of a modular architecture for autonomous robot. In *IEEE ICRA'94, San Diego California, (USA)*, 1994.

[23] F. Garcia and P. Laborie. Hierarchisation of the search space in temporal planning. In *Proceedings EWSP-95*, pages 235–249, 1995.

[24] M. Ghallab and H. Laruelle. Representation and Control in Ixtet, a Temporal Planner. In *Proceedings AIPS-94*, pages 61–67, 1994.

[25] M. Ghallab and A. Mounir-Alaoui. Managing Efficiently Temporal Relations Through Indexed Spanning Trees. In *Proceedings IJCAI*, 1989.

[26] J.-M. Hasemann. Robot control architectures – application requirements, approaches, and technologies. In *XIV Intelligent Robots and Computer Vision: Algorithms, Techniques, Active Vision, Material Handling*, Philadelphia, USA, October 1995.

[27] T. C. Henderson, C. Hansen, and B. Bhanu. A framework for distributed sensing and control. In *9th International Joint Conference on Artificial Intelligence (IJCAI), Los Angeles, California (USA)*, 1985.

[28] F. F. Ingrand, R. Chatila, R. Alami, and F. Robert. Prs: A high level supervision and control language for autonomous mobile robots. In *IEEE ICRA'96, St Paul, (USA)*, 1996.

[29] F. F. Ingrand and V. Coutance. Real-Time Reasoning using Procedural Reasoning. Technical Report 93-104, LAAS/CNRS, Toulouse, France, 1993.

[30] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert, Knowledge-Based Diagnosis in Process Engineering*, 7(6):34–44, December 1992.

[31] M. Khatib, B. Bouilly, T. Siméon, and R. Chatila. Indoor navigation with uncertainty using sensor-based motions. In *IEEE Int. Conf. on Robotics and Automation, Albuquerque (USA)*, May 1997.

[32] C. A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings AAAI-90*, pages 923–928, 1990.

[33] P. Laborie and M. Ghallab. Planning with Sharable Resource Constraints. In *Proceedings IJCAI-95*, pages 1643–1649, 1995.

[34] S. Lacroix, P. Grandjean, and M. Ghallab. Perception planning for a multisensor interpretation machine. In *IEEE ICRA'92, Nice, (France)*, 1992.

[35] J. P. Laumond, P. E. Jacobs, M. Taix, and R. M. Murray. A motion planner for nonholonomic mobile robots. *IEEE Transactions on Robotics and Automation*, 10(5):577–593, 1994.

[36] J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications. In *Proceedings of the Conference on Intelligent Robotics in Field, Factory, Service, and Space*, pages 842–849, Houston, Texas (USA), March 1994.

[37] D. M. Lyons. Representing and analysing action plans as networks of concurrent processes. *IEEE Trans. Robotics and Automation*, 9(3):241–256, June 1993.

[38] D. M. Lyons and A. J. Hendriks. Testing incremental adaptation. In Hammond, editor, *Proc. 2nd AIPS*, pages 116–121, 1994.

[39] P. Moutarlier and R. Chatila. Incremental free-space modelling from uncertain data by an autonomous mobile robot. *IEEE IROS'91, Osaka (Japon)*, November 1991.

[40] S. Schneider, V. Chen, and G. Pardo-Catellote. The ControlShell components-based real-time programming system. In *IEEE ICRA'95, Nagoya (Japan)*, 1995.

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

[42] D. Simon, B. Espiau, E. Castillo, and K. Kapellos. Computed-aided design of a generic robot controller handling reactivity and real-time control issues. *IEEE Transaction on Control Systems Technology*, 1(4), 1993.

[43] L. Spector and J. Hendler. The use of supervenience in dynamic-world planning. In Hammond, editor, *Proc. 2nd AIPS*, pages 158–163, 1994.

[44] D. Stewart and P. Khosla. The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects. *Int. Journal of Software Engineering and Knowledge Engineering*, 6(2):p. 249–277, June 1996.

[45] A. Tate, B. Drabble, and R. Kirby. *O-Plan2: An Architecture for Command, Planning and Control.* Intelligent Scheduling. Morgan-Kaufmann Publishing, 1994.

[46] C. E. Thorpe and M. Hebert. Mobile robotics: Perspectives and realities. In *ICAR'95*, volume 1, pages 497–506, Spain, September 1995.

[47] T. Vidal, M. Ghallab, and R. Alami. Incremental mission allocation to a large team of robots. In *IEEE ICRA'96, St Paul, (USA)*, 1996.