Available online at www.sciencedirect.com

ScienceDirect

Artificial Intelligence 172 (2008) 392–412

Artificial Intelligence

www.elsevier.com/locate/artint

Learning how to combine sensory-motor functions into a robust behavior ☆

Benoit Morisset [a,*], Malik Ghallab [b]

[a] *SRI International, Artificial Intelligence Center, Menlo Park, CA, USA*
[b] *INRIA, Rocquencourt, France*

Received 29 March 2005; received in revised form 23 July 2007; accepted 27 July 2007

Available online 2 October 2007

Abstract

This article describes a system, called ROBEL, for defining a robot controller that learns from experience very robust ways of performing a high-level task such as "navigate to". The designer specifies a collection of *skills*, represented as *hierarchical tasks networks*, whose primitives are sensory-motor functions. The skills provide different ways of combining these sensory-motor functions to achieve the desired task. The specified skills are assumed to be complementary and to cover different situations. The relationship between control states, defined through a set of task-dependent features, and the appropriate skills for pursuing the task is learned as a finite observable *Markov decision process* (MDP). This MDP provides a general policy for the task; it is independent of the environment and characterizes the abilities of the robot for the task.

Keywords: Robot behavior; Sensory-motor function; Skill; Planning; Learning

1. Introduction

Programming a robot for robustly carrying out various tasks in changing, open-ended environments is highly challenging. The technical literature is rich in controversial discussions about the merits and limitations of approaches ranging from automata-based, purely reactive techniques, to deliberative approaches with explicit models and reasoning capabilities, as well as approaches extending popular programming paradigms to robotics, such as synchronous programming, object programming, agent programming, logic programming or constraint programming. A pragmatic view of robot programming, that is widely shared today, states that, (i) there is a clear need for consistently integrating a wide spectrum of representations, (ii) deliberation is needed but the abstract representations of deliberation techniques cannot handle alone, and directly, the low-level feedback loops from sensors to actuators, and (iii) learning is required for mapping human advice and specifications into robot programs.

We describe here an approach along that view. The approach involves two steps:

☆ The authors collaborated for this work when both where with LAAS-CNRS, University of Toulouse, France.

* Corresponding author.
E-mail addresses: morisset@ai.sri.com (B. Morisset), Malik.Ghallab@inria.fr (M. Ghallab).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2007.07.003

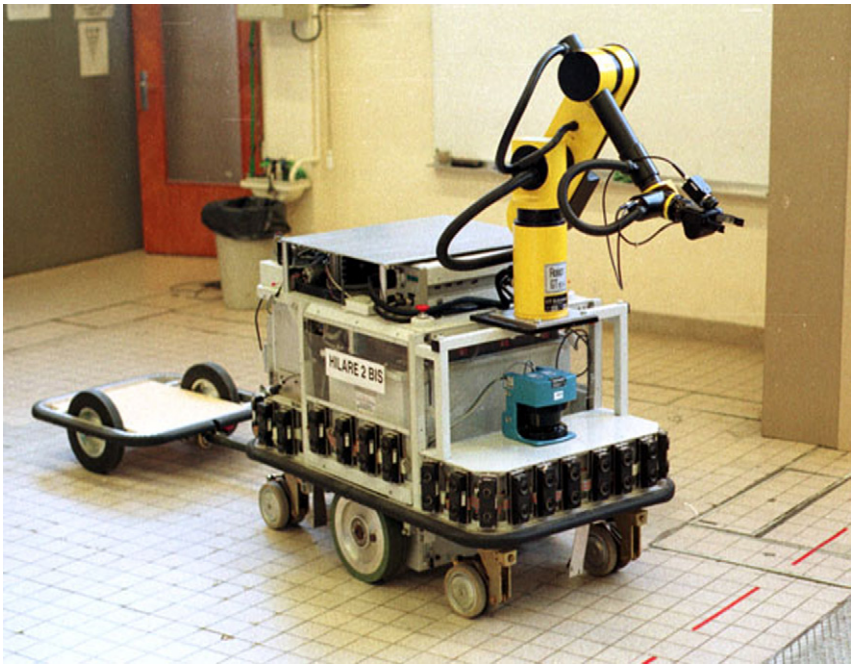

Fig. 1. Hilare 2.

(1) specifying offline and synthesizing a collection of complex plans for achieving a robot task with a given set of sensory-motor functions, and
(2) learning a controller for adequately using this collection of plans.

The learning step is the main focus and original contribution of this paper.

Let us assume that we are given a robot equipped with several sensors—sonar, laser, vision—and actuators—wheels, arm—for moving around and handling objects (Fig. 1). This robot is endowed with various sensory-motor functions, such as localization, map building and updating, motion planning, motion control along a planned trajectory, obstacle avoidance, and arm and grasping control. Let us further assume that this robot has several redundant ways of implementing the same sensory-motor function, eventually using different combinations of sensors and actuators. There are advantages in having several redundant hardware and software means for achieving a sensory-motor function. Redundancy is required for coping with possible hardware failures. Furthermore, no single method or sensor has universal coverage. Each has its weak points and drawbacks. A diversity of means for achieving a sensory-motor function allows for building up robust behavior.

Robustness also requires deliberation capabilities in order to combine several such means into plans appropriate for the current task and context and to revise those plans when needed. Planning techniques usually rely on relational models of tasks with abstract preconditions and effects. The abstract description of a task such as "navigate to" can be useful for mission planning, but not for robustly performing the navigation task with low-level sensory-motor functions. To achieve the deliberation capabilities needed at this level, our approach relies on two automated planning techniques, namely *hierarchical tasks networks* (HTNs) and *Markov decision processes* (MDPs).

For each high-level task the designer specifies offline a collection of HTNs that are complex plans, called *skills*. The primitives of these HTNs are modules implementing sensory-motor functions. Each skill is a possible way of combining a few of these functions to achieve the desired task. A skill has a rich context-dependent control structure. It includes alternatives whose selection depends on the sensor data. The degree of suitability of each skill for pursuing the task can vary depending on the current context. It is assumed that the skills specified by the designer for the task are complementary and provide good coverage of the diversity of execution contexts of that task.

The controller must choose dynamically, at each moment, the right skill for pursuing the task.[1] This choice is far from being obvious since the execution context is difficult to characterize and to map to skills at the robot design stage. However, it is shown here that the relationship between control states and skills can be adequately represented as an *observable finite MDP* and learned through experience. The *action space* of this MDP is the available set of skills for the task; its state space is the robot *control space*.

The control state is defined through the discretized values of a set of task-dependent features describing the current context. For the navigation task these features are, for example, the current estimated uncertainty in the robot location, or the level of cluttering of the perceived local environment. The values of these features are maintained and updated at about 10 Hz. A transition occurs in the control space when a change in the control state is observed. These non-deterministic transitions are labeled by the current skill—since skills are the action space. The probability and cost distributions associated with each transition are estimated during a learning stage. The learned MDP characterizes the robot's abilities for that task.

The controller relies on this learned MDP. It applies a closed-loop control. But it does not follow a universal policy computed offline, as is usually done with MDP controllers in decision-theoretic planning [7]. Our approach is more akin to the *receding horizon control*, as discussed in [3], in which a closed-loop policy is obtained by repeated online planning from the current state, and by the application to that state of the first action (in our case the first skill) of the obtained plan. Usually, in the receding horizon control, each online planning step provides an open-loop policy, that is, a sequence of actions. In our case however, each online planning step provides a closed-loop policy, obtained by decision-theoretic planning, taking into account the objectives specific to that step, as perceived in the control space.

We selected this MDP controller approach for quite natural reasons. First, we need a stochastic mapping from the execution contexts to skills. Decision trees, for example, can be easily learned, but do not handle conveniently a stochastic mapping. In contrast, this is very natural with MDPs. Furthermore, we were able to choose finite observable MDPs because our MDP state space was not *a priori* given. We designed it, as a part of the controller design, by choosing and discretizing task-dependent features that are observable and that characterize well the execution context. This choice of observable finite MDPs is motivated by obvious complexity reasons: it drastically simplifies the learning stage as well as the policy planning stage. Finally, our choice of a particular receding horizon control, by the repeated *online* computation of a closed-loop policy, is more subtle but also very natural. A universal closed-loop policy computed offline is a feasible approach only when the control objectives, as well as the model of the system to be controlled, are available at the design stage. In our case, the control objective—that is, the goal to reach in a navigation task—cannot be expressed, once and for all, in the control space. One must either rely on environment-dependent MDPs, with obvious drawbacks in genericity and complexity, or recompute online closed-loop policies at each control state for the current objective. The latter choice has great benefits in genericity and efficiency, and also in robustness. The controller is not dependent on the convergence properties of an offline computed universal policy. It is more reactive and more focused on the current goal through the repeated online planning of a closed-loop policy applied to the current control state.

This is certainly not the first contribution that relies on a planning formalism and on plan-based control in order to program an autonomous robot. For example, the "*Structured Reactive Controllers*" [4] are close to our concerns and have been demonstrated effectively on the Rhino mobile robot. The efforts for extending and adapting the Golog language [22] to programming autonomous robots offer another interesting example from a quite different perspective, that of the Situation Calculus formalism [30], eventually in combination with HTNs [13] or MDPs [8]. The "*societal agent theory*" of [23] offers another interesting approach for specifying and combining sequentially, concurrently or in a cooperative mode, several agent-based behaviors. The CDL language used for specifying agent interactions is similar to our programming environment. HTNs are used in [6] to specify navigation plans for the Rhino robot and to improve by plan transformation the set of specified HTNs. Let us also mention the "*dual dynamics*" approach of [17] that permits the flexible interaction and control of several skills. These are typical examples of possible architectures for designing controllers for autonomous robots (see [2] for a more comprehensive survey). Our approach relies on the use of HTNs for the specification and synthesis of skills as task networks. The approach is effective because of the intermediate position of this representation between programming and automated planning. In one of our implementations we relied on SHOP [28], a domain independent HTN planner, in order to synthesize *offline* skills

[1] Unless stated otherwise, throughout this paper the word *control* refers to this high-level process of choosing a skill and modifying the robot physical behavior according the sensory-motor functions of the chosen skill.

from generic specifications [26]. This programming approach is very flexible and allows for robustness because of the consistent use of several redundant HTN skills.

The main contribution of this paper is an original approach for learning, from the robot experience, an MDP-based controller that enables a dynamic choice of a skill appropriate to the current context for pursuing the task. Learning a policy—that is, a mapping from a state space to an action space—is a well-known problem with widely studied techniques such as the $TD(\lambda)$ [32] or the Q-learning [36] algorithms, and a broad variety of reinforcement learning methods [33]. These approaches are well founded and have good properties inherited from those of dynamic programming, such as convergence toward optimal policies [19]. Our approach relies on dynamic programming and the value iteration algorithm. However, because of the receding horizon control, it is different from the usual reinforcement learning. We do not aim at reaching a stationary value function $Q(s, a)$ from which an optimal universal policy is derived. Instead, at each decision step a new value function and a new policy are computed online, based on local conditions and control objectives. We deal here only with the so-called *indirect* learning problem [3], that is, estimating the probability and cost distributions of the MDP controller.

Reinforcement learning is quite popular in robotics; it may be used at several levels [31], including the control of navigation tasks. Several authors have directly expressed Markov states as cells of a navigation grid and addressed navigation through MDP algorithms such as, for example, value iteration [10,21,35]. Learning systems have been developed in this framework. For example, XFRMLEARN extends these approaches further with a knowledge-based learning mechanism that adds subplans from experience to improve navigation performances [5]. Other approaches considered learning at very specific levels, for example, to improve path planning capabilities [9,15].

Our approach stands at a more generic level. Its purpose is to acquire autonomously the mapping from an abstract state space, which is the control space, to the action space of redundant skills. We have proposed a convenient control space. We have also introduced a new and effective receding horizon control, by the repeated *online* planning of a closed-loop policy applied to the current state with respect to the updated current objectives. The approach relies on an original search mechanism that repeatedly projects a topological route into the control space, allowing for a precise and local expression of the navigation goal within the control space. The learning of this automaton relies on simple and effective techniques. Because of the abstract state space, the learned MDP is independent of a particular environment; it characterizes the robot capabilities.

This contribution is backed up by an implementation of the three components of the approach—sensory motor functions, HTN skills, and MDP controller—within a mobile robot, and by extensive experiments and evaluation for indoor navigation tasks.

These three components are successively described in the following three sections. The experiments and results are given in a subsequent section, followed by a conclusion with an assessment of the approach.

2. Sensory-motor functions

Let us denote a software module implementing a sensory-motor function as an *sm* function; *sm* functions are the low-level primitives available for programming the robot. Each *sm* function is coded as one or more modules in our software engineering environment, called Genom [2,12]. Each *sm* function offers a set of services that can be requested by another *sm* function or by the controller. Each call creates a process managed according to a finite automaton composed of states such as Idle, Initializing, Interrupted, Executing, or Failed (Fig. 2). The transition events are issued by the controller (noted "event/-" on Fig. 2) or by the execution engine (noted "-/event").

A call to an *sm* function returns a report either indicating the end of a normal execution (the state of the process switches to Idle), or giving additional information about nonnominal execution. If an error occurs during the execution, a specific report corresponding to the error is issued and the activity switches to the state Failed.

As underlined in the introduction, it is important to have a redundant set of *sm* functions, several of them implementing, in complementary ways, localization, mapping, or motion control. To be able to specify skills using *sm* functions, it is also very important to have a good knowledge of these functions, of their weak and strong points, and of how they complement each other. This section details the *sm* functions that we have implemented or used in our experiments. It briefly describes the geometric and topological representations of space used within the *sm* functions.

We rely on a geometric model of the environment learned and maintained by the robot. This model is a 2D map of obstacle edges acquired from the laser range data. A simultaneous localization and mapping (SLAM) technique is used to generate and maintain this map of the environment [27].

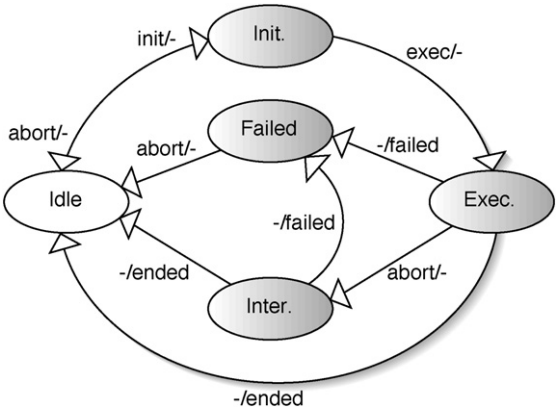

Fig. 2. Genom activity automaton.

A labeled topological graph of the environment is associated to the geometric 2D map. Cells are polygons that partition the metric map. Each cell is characterized by its name and a *color* that corresponds to navigation features such as Corridor, Corridor with landmarks, Large Door, Narrow Door, Confined Area, Open Area, Open Area with fixed localization devices.[2] Edges of the topological graph are labeled by estimates of the metrical length of the path from one cell to the next and by heuristic estimates of how easy the traversal of such a path is.

2.1. Segment-based localization

This function relies on the map maintained by the robot from the laser range data. The SLAM technique uses a data estimation approach called *extended Kalman filtering* in order to match the local perception with the previously built model [27]. It offers a continuous position-updating mode, which is used when a good probabilistic estimate of the robot position is available. This *sm* function explicitly maintains the *inaccuracy* of the robot position estimate as an ellipsoid of uncertainty. When this estimated inaccuracy is above a threshold, the robot is assumed to be lost; a relocalization mode is be performed. A constraint relaxation on the position inaccuracy extends the search space until a good matching with the map is found.

This *sm* function is generally reliable and robust to partial occlusions, and much more precise than odometry. However, occlusion of the laser beam by obstacles gives unreliable data. Occlusion occurs when dense unexpected obstacles are gathered in front of the robot. Moreover, in long corridors the laser obtains no data along the corridor axis. The inaccuracy estimate of the robot position increases along the corridor axis. Restarting the position updating loop in a long corridor can prove to be difficult. A feedback from this *sm* function can be a report of bad localization, which warns that the inaccuracy of the robot position has exceeded an allowed threshold. The robot stops, turns on the spot, and reactivates the relocalization mode. This can be repeated to find a nonambiguous position in the environment to restart the localization loop.

2.2. Localization on visual landmarks

This function relies on a calibrated monocular vision to detect known landmarks such as doors or wall posters [16]. It derives from the perceptual data a very accurate estimation of the robot position. Setting up is simple: a few wall posters and characteristic planar features on walls are learned in supervised mode. However, landmarks are available and visible only in a few areas of the environment. Hence this *sm* function is mainly used to update from time to time the last known robot position. A feedback from this *sm* function is a report of a potentially visible landmark, which indicates that the robot enters an area of visibility of a landmark. The robot stops and turns toward the expected

[2] Existing environment modeling techniques enable automatic acquisition of such a topological graph with the cells and their labels, e.g., [34]. However, in the implementation referred to here, the topological graph is hand-programmed.

landmark; it searches the landmark by using the pan-and-tilt mount. A failure report discloses that the landmark was not identified. Eventually, the robot retries from a second predefined position in the landmark visibility area.

2.3. Absolute localization

The environment may have areas equipped with calibrated fixed devices, such as infrared reflectors or cameras, or even areas where a differential GPS signal is available. These devices permit very accurate and robust localization [11], but the *sm* function works only when the robot is within a covered area.

2.4. Elastic band for motion control along a planned trajectory

This *sm* function updates and dynamically maintains a flexible trajectory as an *elastic band* or a sequence of configurations from the current robot position to the goal [20]. Connection between configurations relies on a set of internal forces that are used to optimize the global shape of the path. External forces are associated with obstacles and are applied to all configurations in the band in order to dynamically modify the path to take it away from obstacles. This *sm* function takes into account the planned path, the map, and the online input from the laser data. It gives a robust method for long-range navigation. However, the band deformation is a local optimization between internal and external forces; the techniques may fall into local minima. This is the case when a mobile obstacle blocks the band against another obstacle. Furthermore, it is a costly computational process that may limit the reactivity in certain cluttered, dynamic environments. This also limits the band length.

The feedback may warn that the band execution is blocked by a temporary obstacle that cannot be avoided (e.g., a closed door, an obstacle in a corridor). This obstacle is perceived by the laser and is not represented in the map. If the band relies on a planned path, the new obstacle is added to the map. A new trajectory taking into account the unexpected obstacle is computed, and a new elastic band is executed. Another report may warn that the actual band is no longer adapted to the planned path. In this case, a new band must be created.

2.5. Reactive obstacle avoidance

This *sm* function provides a reactive motion capability toward a goal without needing a planned path [24]. It extracts from sensory data a description of free regions. It selects the region closest to the goal, taking into account the distance to the obstacles. It computes and tries to achieve a motion command to that region.

The function offers a reactive motion capability that remains efficient in a cluttered space. However, like all the reactive methods, it may fall into local minima. It is not appropriate for long-range navigation. Its feedback is a failure report generated when the reactive execution is blocked.

2.6. Path planner

A path planner [29] may be seen as an *sm* function from the viewpoint of a high-level navigation controller. Note that a planned path does not take into account environmental changes and new obstacles. Furthermore, a path planner may not succeed in finding a path. This may happen when the initial or goal configurations are too close to obstacles: because of the inaccuracy in the estimated robot position, the corresponding configurations may not be found within the free part of the configuration space. The robot must move away from the obstacles by using a reactive motion *sm* function before a new path can be planned.

3. Skills

A navigation task such as `(Goto x y` θ`)` given by a mission planning step requires an integrated use of several *sm* functions among those presented earlier. Each consistent combination of these *sm* functions is a particular plan called a *skill*. A navigation skill is one way of performing the navigation task. A skill has specific characteristics that make it more appropriate for some contexts or environments, and less for others. We will discuss later how the controller chooses the skill appropriate for the context. We exemplify some of such skills for the navigation task before giving the detail of the HTN representation for skills.

Skill A_1**.** This skill uses three *sm* functions: the path planner, the elastic band for the dynamic motion execution, and the laser-based localization. When A_1 is chosen to carry out a navigation, the laser-based localization is initialized. The robot position is maintained dynamically. A geometric path is computed to reach the goal position. The control along that path is carried out by the elastic band *sm* function. Stopping the skill interrupts the band execution and the localization loop; it restores the initial state of the map if temporary obstacles have been added to it. Suspending the skill stops the band execution. The path, the band and the localization loop are maintained. A suspended skill can be resumed by restarting the execution of the current elastic band.

Skill A_2**.** This skill uses three *sm* functions: the path planner, the reactive obstacle avoidance, and the laser-based localization. The path planner provides waypoints (vertices of the trajectory) to the reactive motion function. Despite these waypoints the reactive motion can be trapped in local minima in cluttered environments. Its avoidance capability is higher than that of the elastic band *sm* function. However, the reactivity to obstacles and the attraction to waypoints may lead to oscillations and to a discontinuous motion that confuses the localization *sm* function. This is a clear drawback for A_2 in long corridors.

Skill A_3**.** This skill is like A_2 but without path planning and with a reduced speed in obstacle avoidance. It starts with the reactive motion and the laser-based localization loop. It offers an efficient alternative in narrow environments like offices, and in cluttered spaces where path planning may fail. It can be preferred to A_1 in order to avoid unreliable replanning steps if the elastic band is blocked by a cluttered environment. Navigation with this skill is reactive, hence it may fall into a local minima problem. The weakness of the laser localization in long corridors is also a drawback for A_3.

Skill A_4**.** This skill uses the reactive obstacle avoidance together with the odometer and the visual landmark localization *sm* functions. The odometer inaccuracy can be locally reset by visual localization when the robot goes by a known landmark. Reactive navigation between landmarks allows the robot to cross a corridor without accurate knowledge of its position. Typically skill A_4 can be used in long corridors if visual landmarks are available. However, the growing inaccuracy can make it difficult to find out the next landmark. The visual search method allows for some inaccuracy on the robot position by moving the cameras but this inaccuracy cannot exceed 1 m. For this reason landmarks should not to be too far apart with respect to the required updating of odometry estimates. Furthermore, the reactive navigation of A_4 may fall into local minima.

Skill A_5**.** This skill relies on the reactive obstacle avoidance *sm* function and the absolute localization *sm* function when the robot is within an area equipped with absolute localization devices.

Skills are represented as hierarchical task networks (Fig. 3). The HTN formalism is adapted to skills because of its expressiveness and its flexible control structure. HTNs offer a middle ground between programming and automated planning, allowing the designer to express search control knowledge, which is available in our case. In HTN planning, the planner is given this search control knowledge as a set of *methods*. A method is a formal description of how to decompose a task into a set of subtasks, together with the conditions and ordering constraints of this decomposition. Several alternative methods may be available for decomposing a given task. Planning proceeds by decomposing *non-primitive* tasks recursively into smaller and smaller subtasks, until only *primitive* tasks are reached. These can be performed directly using available actions (see for details [14], Chapter 11).

HTNs can be very general. For ROBEL we need only the restricted class of *ordered simple task networks*. Graphically, such an HTN is an And/Or tree. An internal node of the HTN tree is a task or a subtask that can be pursued in different ways depending on the context. These alternate ways are specified by *methods* for decomposing the task or subtask into an ordered conjunction of subtasks. The leaves of the tree are primitive actions, each corresponding to a unique query sent to the Genom module representing the *sm* function.

A root task is dynamically decomposed, according to the context, into a set of primitive actions organized as concurrent or sequential subsets. Execution starts as soon as the decomposition process reaches a leaf, even if the entire decomposition process is not complete. A primitive action can be *blocking* or *nonblocking*. In blocking mode, the control flow waits until the end of this action is reported before starting the next action in the sequence flow. In nonblocking mode, actions in a sequence are triggered sequentially without waiting for a feedback. A blocking

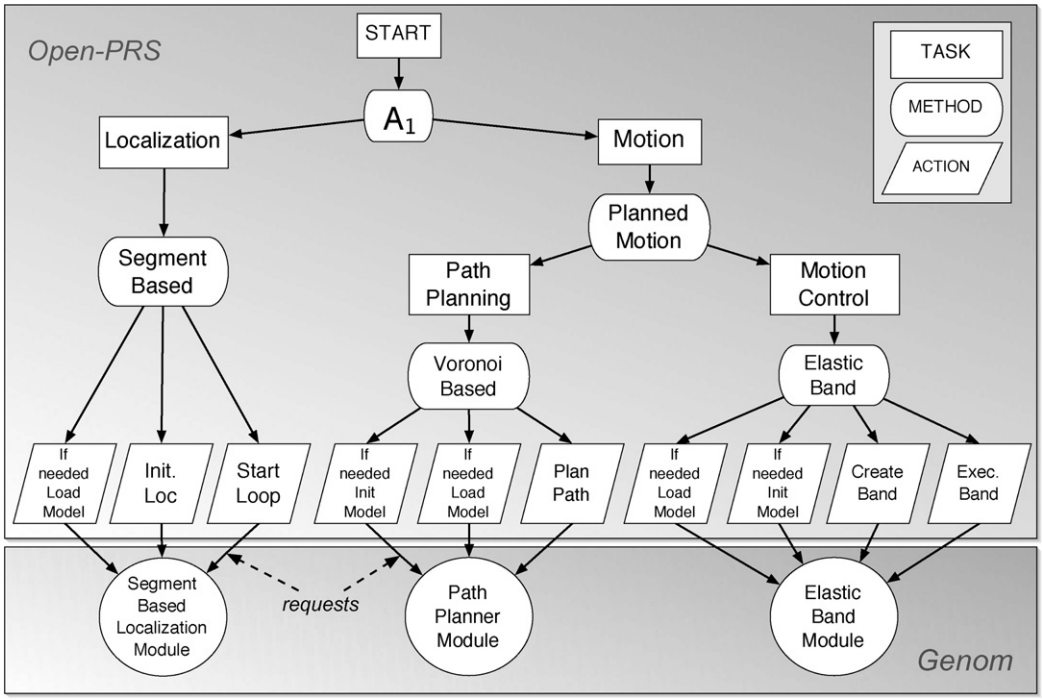

Fig. 3. START task of skill A_1.

primitive action is considered ended after a report has been issued by the *sm* function and after that report has been processed by the control system. The report from a nonblocking primitive action may occur and be processed after an unpredictable delay. A nonblocking primitive can be seen as a background process whose execution may affect the skill, but it does not change the decomposition process. For example, the segment-based localization loop is started in nonblocking mode.

Each skill corresponds to six different HTN tasks: `start, stop, suspend, resume, succeed`, and `fail`. The `start` task represents the nominal skill execution. The `stop` task stops the skill, restores the neutral state, and is characterized by the lack of any *sm* function execution. It also enables the environment model modified by the skill execution to recover its previous form. The `suspend` and `resume` tasks are triggered by the recovery system described below. The `suspend` task stops the execution by freezing the state of the active *sm* functions. The `resume` task restarts the skill execution from such a frozen state. The `fail` (resp. `succeed`) task is followed when the skill execution reaches a failure (resp. a success) end. These tasks are used to restore the neutral state and to allow certain executions required in these specific cases.

The implementation of the five skills presented above involves several methods for each `start` task. For example, the localization task is decomposed with two methods: the segment-based method used by skills A_1, A_2, A_3 and A_5, and the vision-based method used by A_4. The motion task is decomposed with two methods: the planned motion method used by A_1, A_2 and A_4, and the reactive motion method used by A_3 and A_5.

In a first implementation of ROBEL [25], no particular planner has been used to synthesize the five skills presented above. For each skill, each task is decomposed by a unique predefined method. This deterministic decomposition scheme has been directly implemented in Open-PRS (Procedural Reasoning System) [18]. The decomposition takes place online when a skill is started, stopped, suspended, or resumed. In a subsequent implementation [26], the offline synthesis of a set of skills from specified formal methods has been performed interactively using the Shop-2 planner [28].

4. The resource and recovery manager

A skill triggers several parallel activities that share resources. It receives reports back from its own *sm* functions. These feedback reports and the shared resources are managed by the *Resource and Recovery Manager* (R&RM).

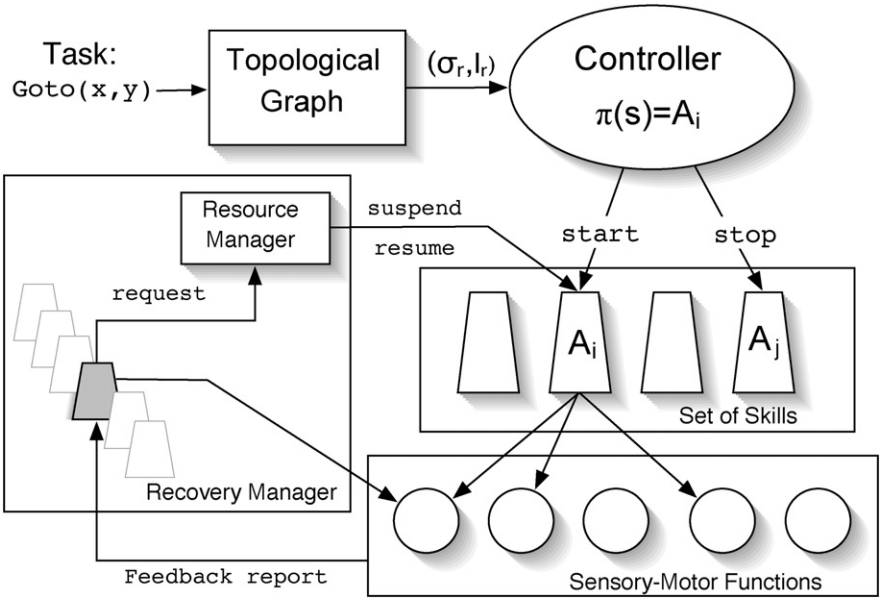

Fig. 4. The ROBEL system.

The R&RM catches and reacts appropriately to reports emitted by *sm* functions (Fig. 4). The Resource and Recovery Manager is also specified and implemented through a set of HTN tasks, called R-tasks.

A nonnominal report from a skill signals a particular type of *sm* function execution. The aim of the corresponding R-task is to recover to a nominal execution of the skill. The following two example illustrates the recovery mechanisms:

- A report from a localization function signaling that the robot is lost triggers a relocalization mode. The robot turns on the spot, trying to catch in the environment some new feature to reinitialize the localization loop.
- The path planner may fail if the robot is too close to an obstacle. In that case a specific report is issued. The R-task corresponding to this report starts a clearance procedure and moves the robot away from the obstacles. If the clearance is successful, the resource manager resumes the execution of the current skill.

Some nonnominal reports can be nonrecoverable failures. In these cases, the corresponding R-task sends a fail message to the skill pursuing this *sm* function. Nominal reports may disclose the success of the global task. In this case, the success alternative of the skill is activated.

Resources to be managed are either physical and nonsharable (e.g., motors, cameras, pan-and-tilt mount) or logical (the environment model that can be temporally modified). The execution of a set of concurrent nonblocking actions can imply the simultaneous execution of different *sm* functions. These concurrent executions may generate a resource conflict. For example, when a report signaling a bad localization is issued, the R-task corresponding to this report must lock the resource "motor" to stop the robot and to start the re-localization procedure. To allow this recovery procedure, the skill under execution must unlock the resource "motor" in conflict.

The R&RM manager organizes the resource sharing with semaphores and priorities.

- Each nonsharable resource is semaphorized. The request for a resource takes into account the priority level of each consumer. This priority level is specified by the designer.
- The execution of an R-task is not interruptible. If another HTN task requires a resource already in use by a control execution, the message activating this task (either skill or control) is added to a spooler according to its priority level.
- An R-task has a priority higher than those of `start` and `suspend` tasks but lower than those of `stop` and `fail` tasks.

When a nonnominal report is issued by a Genom module, the R-task corresponding to this report starts its execution. It requests the resource it needs. If this resource is already in use by a skill the manager sends to this skill a suspend message, and leaves a resume message for the skill in the spooler according to its priority. The suspend alternative is executed, freeing the resource and enabling the R-task to be executed. If the R-task execution succeeds, waiting messages are removed and executed until the spooler becomes empty. If the R-task execution fails, the resume message is removed from the spooler and the fail alternative is executed for the skill.

5. The controller

5.1. The control space

The controller must choose the skill that is the most appropriate to the current state for pursuing the task (Fig. 4). A set of *control features* must provide control information. The choice of these control features is an important design issue. For example, in the navigation task the control features that characterizes the navigation conditions are the following:

- The *cluttering* of the environment, which is defined as the weighted sum of the distances to nearest obstacles perceived by the laser range finder, with a dominant weight along the robot motion axis. This information is important in establishing the execution conditions of the motion and the localization *sm* functions.
- The *angular variation* of the profile of the laser range data which characterizes the robot's local environment. Close to a wall, the cluttering value is high but the angular variation remains low. In an open area the cluttering value is low while the angular variation may be high.
- The current *inaccuracy of the robot position*, a value that is estimated from the co-variance matrix maintained by the localization *sm* function.
- The current *confidence in the robot position* estimate. The inaccuracy is not sufficient to qualify the localization. Each call to the localization *sm* function supplies a heuristic estimate of the confidence in the last processed position.
- The navigation *color* of the current area. When the robot position estimate falls within some labeled cell of the topological graph, the corresponding labels are taken into account, for example, *Corridor, Corridor with landmarks, Large door, Narrow door, Confined area, Open area, Area with fixed localization*.
- The current *skill*. This information is essential to assess the control state and possible transitions between skills.

Let $f_1, \ldots, f_m$ be a set of control features that range over *finite* sets, $f_i \in D_i$. A control state is defined by the values of these control features. Hence, the control space is provided by the Cartesian product $\prod_{i=1}^{m} D_i$.

To have a finite state space we need finite ranges D_i. For that, continuous features are discretized over a few intervals. In our implementation, the choice of these intervals was made manually by analyzing statistical data. We recorded the values of the features for the robot in different navigation contexts. For instance, the cluttering feature is useful for characterizing the obstacle avoidance conditions. To determine the intervals for this feature, the robot was placed in significantly different situations regarding avoidance constraints: open spaces, doorways, offices, each with several densities of obstacles. For each situation, the values of the cluttering feature were recorded. These values, labeled by the situations we wanted to discriminate, were analyzed. Because we were controlling the experiment by choosing the more appropriate situations to record, it was easy to define thresholds and intervals separating the situations of interest. In more complex cases, classical clustering techniques could have been used.

In our experiments, the cluttering feature, the angular variation, the inaccuracy of the pose estimate and the confidence in the pose estimate are discretized respectively over four, three, four and three intervals. The navigation color of the area ranges over seven values; we have five skills. Hence, $\prod_{i=1}^{m} |D_i| = 4 \times 3 \times 4 \times 3 \times 7 \times 5 = 5040$.

The values of these five features are maintained and updated at about 10 Hz. A transition occurs in the control space when a change in the value of a feature is observed. The 10 Hz rate is sufficiently high with respect to the speed of the robot to guarantee that every change in the control space will be noticed, including every change in the color of the traversed area.

For convenience, we consider an additional global failure state that is reached whenever the recovery and resource manager of a skill reports a failure. We finally end up with the finite control space $S = \{\text{failure}\} \cup \prod_{i=1}^{m} D_i$, from which we define a *control automaton*.

5.2. The control automaton

Unpredictable external events may modify the environment and consequently the values of control features. For example. someone passing by may change the value of the cluttering feature, or the localization inaccuracy feature. Even without external events, that is with the assumption of a static environment, the transition from one control state to the next is not fully predictable unless one assumes complete and precise models of the environment and no uncertainty in the sensing functions. These assumptions are clearly unrealistic. Hence we relied on nondeterministic transition models.

The nondeterministic control automaton is defined as the tuple $\Sigma = \{S, A, P, C, R\}$ where

- $S = \{\text{failure}\} \cup \prod_{i=1}^{m} D_i$ is the finite control space.
- A is the finite set of skills.
- $P : S \times A \times S \to [0, 1]$ is a probability distribution on the state-transition function; $P_a(s'|s)$ is the probability that the execution of skill a in state s leads to state s'.
- $c : A \times S \times S \to \Re^+$ is a cost function; $c(a, s, s')$ is the cost of performing the transition from s to s' when executing the skill a; $C(a, s)$ is the average over all possible transitions with a from s: $C(a, s) = \sum_{s'} c(a, s, s') P_a(s'|s)$.
- $R : S \to \Re$ is a reward function.

The utility of a pair (s, a) is defined as $U(s, a) = R(s) - C(a, s)$.

A and S are given by design from the definition of the set of skills and the set of control features. In the navigation system illustrated here, $A = 5$ skills and $S = 5041$ states. P and C are obtained from observed statistics during a learning phase. Rewards are computed and updated at each step for the given navigation destination.

The control automaton Σ defines a Markov decision process. As an MDP, Σ could be used reactively on the basis of a *universal policy* π that selects for a given state s the best skill $\pi(s)$ to be executed. However, a universal policy cannot meet different navigation destinations. A more precise approach takes into account explicitly the specified destination g, transposed into Σ as a set S_g of desirable goal states in the control space, with appropriate rewards. This set S_g is given by a repeated look-ahead mechanism based on a search for paths in Σ that may reflect a topological route to the destination g. This mechanism permits a robust receding horizon control with online planning of a closed-loop policy, applied to the current state and with respect to the current objectives, updated for that state.

5.3. Goal states in the control space

Given a navigation destination g, let r be a route in the topological graph from the current robot position to g. r is a sequence of topological cells; it is given by a search in the topological graph, eventually taking into account estimated metrical lengths of edges between cells. Let us characterize a route r by the sequence $\sigma_r = \langle c_1 c_2 \dots c_k \rangle$ of the distinct colors of traversed cells such that $c_i \neq c_{i+1}$ for $i = 1$ to $k - 1$; that is, c_i corresponds to one or several consecutive cells of the same color that are traversed by route r.

It is not possible to map directly the navigation destination g into a control goal state s_g because we are using an abstract control space Σ that does not represent explicitly spatial positions. But this abstract control space does take into account the color of the traversed area. Hence, it is possible to map a route r leading to g into a set of paths in Σ with corresponding colors, and to entail from it appropriate rewards to control states in these paths.

Let p be a path in Σ between two states. Path p can be characterized by its color signature σ_p, that is, the sequence of colors of control states along path p. A path p in Σ is color compatible with a route r in the topological graph, denoted $\text{color}(p, r)$; when σ_p corresponds to the same sequence of colors as σ_r with possible repetition factors, that is

$$\text{color}(p, r) \quad \text{iff} \quad \sigma_p = \langle c_1^{i_1} c_2^{i_2} \dots c_k^{i_k} \rangle \text{ for } \sigma_r = \langle c_1 c_2 \dots c_k \rangle \text{ and } i_1 > 0, \dots, i_k > 0 \text{ are integers.}$$

color(p, r) requires that path p in Σ is traversing control states having the same color as the planned topological route r. A repetition factor corresponds to the number of control states, eventually several but at least one, required for traversing one or several consecutive topological cells of identical colors.

Let r be a route in the topological graph from the current robot position to the destination g, and $\sigma_r = \langle c_1 \dots c_i c_{i+1} \dots c_k \rangle$. Let s_0 be the current control state; its color is by definition c_1. The set of paths $\{p \in \Sigma \mid \text{color}(p, r) \text{ holds}\}$ is given by a tree T, rooted in s_0, and defined recursively as follows. Let s be an interior node of T whose color is c_i, for $i = 1$ to $k - 1$, s' is a successor of s in T iff the following conditions hold:

- s' is a successor of s in Σ, i.e., there is an action a such that $P_a(s'|s) \neq 0$.
- The color of s' is either c_{i+1} or c_i. In the latter case s' is required to be different from s and from any of the immediate predecessors of s that have the same color c_i,
- When $i = k - 1$ and the color of s' is c_k, then s' is a leaf node in T.

This definition allows for paths in Σ with loops, but it forbids infinite loops over a subset of states with the same color. The tree T is obviously finite since each node has a finite number of immediate successors, and the depth of T is bounded by k and by the condition of distinct consecutive states with the same color.

The tree T is given by a straightforward depth-first search in Σ, keeping only paths that reach a leaf node. In the worst case, T can be of exponential size. In practice, however, the sequence σ_r focuses T to a very narrow search with few color-compatible paths. We may even have an empty tree; that is, no complete path meets the above conditions. In that case we can either look for another topological route r' to the given destination and repeat the procedure, or we can take the deepest path in T found along the depth-first search and consider it to be a good-enough approximation of a color-compatible path for the purpose of the heuristics needed to choose the next skill.

Having defined T, the reward function in Σ with respect to the destination g, to the current position and current control state of the robot is computed in the following way.

- Initially, $R(s) \leftarrow -R_0$ for every $s \in \Sigma$, R_0 being a positive integer.
- $R(s') \leftarrow R_0$ for every leaf node s' in T.
- $R(s) \leftarrow 0$ for every interior node s in T.

This procedure gives positive rewards to goal states in S_g, neutral rewards to intermediate states leading to S_g, and negative rewards to other states. It is important to notice that this set S_g of control states is a *heuristic projection* of the planned topological route to the destination g.[3] There is no guarantee that following blindly (i.e., in an open-loop control) a path p in Σ that meets color(p, r) will lead to the destination, and there is no guarantee that every successful navigation to the destination corresponds to a sequence of control states that meets color(p, r). This is an efficient and reliable way of focusing the MDP utility function with respect to the navigation destination, to the current position and control state, and to the planned route. It is quite robust since we are using it in a receding horizon control procedure with online replanning of a closed-loop policy at each step.

5.4. Updating the control policy

At this point we must find the best skill to apply to the current state s_0 in order to reach a state in S_g, given the probability distribution function P and the utility function U: $U(s, a) = R(s) - C(a, s)$.

A simple adaptation of the *Value Iteration* algorithm solves this problem (Fig. 5). One must solve the usual Bellman equation: $E(s) = \max_{a \in A} Q(s, a)$, where $E(s)$ is the expected utility in a state s, and $Q(s, a)$ is the expected value of executing the skill a in a state s: $Q(s, a) = U(s, a) + \gamma \sum_{s' \in S} P_a(s'|s) E(s')$, where γ is the discount factor, $0 < \gamma < 1$, to reduce the contribution of distant rewards and costs to the current state. The dynamic programming approach for solving the Bellman equation leads to the iterative update:

$$E_k(s) \leftarrow \max_{a \in A} \left\{ U(s, a) + \gamma \sum P_a(s'|s) E_{k-1}(s') \right\}$$

[3] We also experimented with a heuristics that extends the color-compatible condition with a constraints on the metrical length of r with respect to an estimate of the minimal length of p.

```
Value-Iteration (Σ,U,γ)
for each s ∈ S do
  if s ∈ S_g then E(s) ← 0
  else E(s) ← ∞
k ← 1
while k < maximum number of iterations do
  for each s ∈ S do
    for each a ∈ A do
      Q(s,a) ← U(s,a) + γ Σ_{s'∈S} P_a(s'|s) E_{k−1}(s')
    E_k(s) ← max_{a∈A} Q(s,a)
    π(s) ← arg max_{a∈A} Q(s,a)
  k ← k + 1
return(π)
end
```

Fig. 5. Value iteration.

The stopping criterion for this iterative update can be either a fixed upper bound on the number of iterations, as stated in the algorithm, or a more flexible criterion of close convergence toward the fixed point given by the Bellman equation:

$$\max_{s\in S}\left|E_n(s) - E_{n-1}(s)\right| < \epsilon \tag{1}$$

The output of the algorithm is not a fixed policy that will be used throughout, but an updated policy for the current state and current estimated objectives.

5.5. The control loop

The receding horizon closed-loop controller uses the updated policy π given by the previous procedure as follows:

- The computed skill $\pi(s_0)$ is triggered; that is, the `start` HTN task for $\pi(s_0)$ is executed.
- The robot keeps observing the current control state; when a new state s is observed, it performs the following.
  - It updates the route r, the set S_g of goal states, and the reward function R with respect to s.
  - It finds the new policy π with respect to this context and the skill to apply to s.

The closed loop is repeated until the control reports a success or a failure. Recovery from a failure state consists in trying from the parent state an untried skill. If none is available, a global failure of the task is reported. This receding horizon control loop is fairly robust sensing noise, to imprecise and partial models of the environment and to external events that may change the control state since we restart a new planning of the policy from the updated state.

5.6. Estimating the distributions of the control automaton

During the learning stage, a sequence of randomly generated navigation goals is given to the robot. Along its navigations, new control states are met and new transitions are recorded or updated. Each time a transition from s to s' with skill a is performed, the traversed distance and time are recorded, and the average values for this transition are updated. The cost of the transition $c(a, s, s')$ can be defined as a weighted average of the traversal time for this transition, taking into account the eventual control steps required during the execution of the skill a in s together with the outcome of that control. The statistics on $a(s)$ are recorded to update the probability distribution function.

During the learning stage, a failure is used to extend the scope of untried skills: we move back to the parent state and try a new skill. If none is available or if a new failure is recorded, the robot is manually moved to another part of the environment and the learning of Σ is resumed from this new location. The current state can be a state previously visited before the failure, or a new state in Σ.

Several strategies can be defined to learn P and C in Σ. For example:

Fig. 6. A cluttered environment.

- A skill is chosen randomly for a given task; this skill is pursued until either it succeeds or a fatal failure is notified. In this case, a new skill is chosen randomly and is executed according to the same principle. This strategy is used initially to expand Σ.
- Σ is used according to the normal control except in a state on which not enough data has been recorded; a skill is randomly applied to this state in order to augment known statistics, for example, the random choice of an untried skill in that state.

The former is a pure exploration strategy, and the latter is a blending of exploration and exploitation.

6. Experimental results

To assess the contributions proposed here, we wanted to qualify (i) the multiskill approach for navigating in different types of indoor environments and (ii) the learning technique and the control automata. In the first stage (Section 6.1) we wanted to assess the coverage of a reasonably small set of five skills for coping with a spectrum of indoor environments as wide as possible. In the second stage (Section 6.2) the purpose was to evaluate how possible and how easy it is to generate a controller Σ for handling two skills, A_1 and A_3.

6.1. Coverage of skills

The five skills described earlier have been fully implemented on our Diligent Robot [1] and extensively experimented with. To characterize the usefulness domain of each skill we measured in a series of navigation tasks, the success rate and other parameters such as the distance covered, the average speed, the number of retries, and control actions. Various types of navigation conditions have been considered with several degradations of the navigation conditions,[4] obtained by

- Modifying the environment: cluttering an area (Fig. 6) and hiding landmarks (Fig. 7).
- Modifying navigation data: removing essential features from the 2D map.

[4] Note that usually experimental setups in robotics simplify the environment, while here we are making more complex.

Fig. 7. Degradation of a long corridor.

Fig. 8. A complete occlusion of 2D edges.

Five different types of experimental conditions have been distinguished:

- *Case* 1: very long range navigation in nominal environment and map, e.g., turning around a circuit of 250 m of corridors several times; minimum distance between obstacles is greater than 1 m.
- *Case* 2: traversal of a highly cluttered area (as in Fig. 6); minimum distance between obstacle is greater than 0.75 m.
- *Case* 3: traversal of the area covered by fixed cameras with the occlusion of the 2D characteristic edges of that area (Fig. 8).
- *Case* 4: traversal of very long corridors with no obstacles.

Skill	case 1	case 2	case 3	case 4	case 5
A_1	$\#r = 20$ $SR = 100\%$	$\#r = 5$ $SR = 0$ to 100%	$\#r = 5$ $SR = 0\%$	$\#r = 20$ $SR = 100\%$	$\#r = 20$ $SR = 5\%$
A_2	$\#r = 12$ $SR = 80\%$	$\#r = 5$ $SR = 0$ to 100%	$\#r = 5$ $SR = 0\%$	$\#r = 12$ $SR = 80\%$	$\#r = 0$ $SR \approx 5\%$
A_3	$\#r = 10$ $SR = 0$ to 100%	$\#r = 10$ $SR = 100\%$	$\#r = 0$ $SR \approx 0\%$	$\#r = 0$ $SR \approx 80\%$	$\#r = 0$ $SR \approx 5\%$
A_4	$\#r = 0$	$\#r = 0$	$\#r = 0$	$\#r = 20$ $SR = 95\%$	$\#r = 12$ $SR = 100\%$

Fig. 9. Experimental results.

- *Case* 5: traversal of long corridors with hidden landmarks and a map degraded by removal of several essential 2D features, such as the contour of a very characteristic heating appliance (Fig. 7).

The experimental results are summarized in Fig. 9,

where $\#r$ denotes the number of runs, and *SR* is the success rate of the experiment. In some cases, the performance of a skill varies widely depending on the specifics of each run. Instead of averaging the results, we found it more meaningful to record and analyze the variation conditions. The following comments clarify the experimental conditions and some results:

- Skill A_1: This skill was able to perform 20 loops of 250 m with an average speed of 0.26 m/s with no failure as long as no narrow pass is encountered on the circuit ($SR = 100\%$). We noticed that *SR* drops sharply in *case* 1 if the circuit contains a narrow pass of less than 1 m with sharp angles. In this context, all failures are reported by the elastic band that is unable to handle this situation. Similarly, for *case* 2 the results vary from easy success to complete failure depending on the minimum distance between obstacles ($SR = 0$ to 100%). As expected, A_1 fails completely in *case* 3 if the 2D edges are occluded since the laser-based localization is unable to operate ($SR = 0\%$). Twenty runs have been successfully performed in *case* 4 ($SR = 100\%$). However, we observed that too many obstacle avoidances performed in the middle of corridors may reduce *SR* because of a failure of the laser-based localization. In *case* 5 longer corridors or imprecise initial localization sharply reduces *SR* ($SR = 5\%$). All the failures recorded in these cases come from the laser-based localization.
- Skill A_2: Because of better avoidance capability of the reactive obstacle avoidance module over the elastic band, A_2 is significantly more robust than A_1 in narrow and intricate passes (*case* 1). A_2 is also slightly faster than A_1 for this case (0.28 m/s). However, the greater reactivity of the obstacle avoidance generates more rotations of the robot. These rotations may lead to a less-precise laser-based localization, which explains lower values of *SR* ($SR = 80\%$ for *case* 1 and $SR = 80\%$ for *case* 4 as opposed to 100% for A_1 in both cases). As for A_1, a wide spectrum of results is obtained in *case* 2 depending on the environmental conditions. We noticed that if the combination of the reactive obstacle avoidance with the path planner is useful for *case* 1, it may create some local minima for *case* 2: the reactive obstacle avoidance starts oscillating between the repulsion of a close obstacle and the attraction of the next waypoint. This situation generates a systematic failure ($SR = 0$ to 100%). For *case* 3, 100% of failure is obtained because of the inability of the laser-based localization. For *case* 5, since A_2 uses the same localization function as A_1 and since the obstacle avoidance method is not essential in the traversal of a corridor, we conclude that no difference exists between A_1 and A_2.
- Skill A_3: Since this skill does not use a path planner, A_3 cannot perform the circuit of *case* 1 ($SR = 0\%$). We manually introduced several waypoints along the circuit to guide the navigation. Waypoints too far apart lead to a failure when the reactive obstacle avoidance reaches a local minimum. On the other hand, A_3 is very successful in *case* 2 ($SR = 100\%$) even when the distance between obstacles is as low as 0.75 m, but A_3 navigates at a slow speed (average speed = 0.14 m/s). *Case* 3 has not been tested because A_3 uses the same laser-based localization as A_1 and A_2. Consequently we can predict that A_3 will fail in that case. In a straight corridor with no local minima, we infer that the performance of A_3 for *case* 4 and *case* 5 will be approximately the same as A_2.

- Skill A_4: For *case* 1, *case* 2, and *case* 3, the corresponding environments were not covered with the needed visual landmarks. Only the long corridors were equipped. For *case* 4 and *case* 5, A_4 is not sensitive to the length of the corridor or the degradation of its map as long as it contains enough landmarks.
- Skill A_5 (not shown in the table): This skill fails everywhere except in *case* 3, since it is the only case where the robot can be externally localized.

An interesting view of our results is that for each case there is at least one successful skill. These are the following:

– for *case* 1: A_1 or A_2
– for *case* 2: A_3
– for *case* 3: A_5 or A_4 if the area is equipped with visual landmarks
– for *case* 4: A_1, A_2 and A_4
– for *case* 5: A_4

This clearly supports the assumption that the skills specified by the designer for the navigation task can be complementary and provide good coverage of the diversity of execution contexts of that task. It also supports the proposed approach of a controller that switches from one skill to another according to the context through the repeated online planning of a closed-loop policy applied to the current control state.

The above results correspond to navigation experiments using a single skill at a time. Four skills, A_1, A_2, A_3 and A_4, have been demonstrated together [1] using a set of selection rules specified manually. The fragile results obtained with the rule-based controller triggered our work on learning a robust control system.

6.2. Learning the controller

Our purpose here is to assess the approach for learning a controller Σ. To simplify the experiment, we restricted the controller to handling only two skills, using a reduced control space of just four features. We started with an empty automaton and two complementary skills:

- Skill A_1, composed of path planning, elastic band, and laser-based localization.
- Skill A_3 that works reactively, without path planning, with reactive obstacle avoidance, and laser localization as with A_1.

The velocity of the reactive obstacle avoidance is twice as slow as the velocity of the elastic band. This, together with the path planner, makes A_1 more efficient in large and open environments. On the other hand, the limited avoidance capabilities of the elastic band make A_3 better adapted to highly cluttered environments.

In this experiment, the learning strategy favors the exploration, that is, the completion of each transition. If a skill has not been tried for the current state (untried skill), this skill is automatically chosen by the controller without any computation of π.

The control state space is defined with the following four features: the *cluttering* of the environment (ranging from C0 to C3), the *angular variation* of the laser profile (ranging from P0 to P2), the navigation *color* (OA for "open area" and CA for "confined area"), and the skill being executed when the state is encountered.

A sample of four transitions is represented in Fig. 12. The probability P and the cost C are given for each transition as they appear at the end of the experiment. The cost C is the inverse of the average speed recorded for each transition.

The experiment involves a total of 173 navigations over three phases with, respectively 83 navigations (noted from $n1$ to $n83$), 30 navigations (from $n84$ to $n114$) and 58 navigations (from $n115$ to $n173$).

Phase 1. The learning starts with a series of 83 navigations in a large open environment. During these navigations, 86 states and 159 transitions are created in Σ (left part of Fig. 10). After the 53rd navigation the number of new transitions encountered by the system tends to be stable. Between $n53$ and $n83$, the computation of π returns A_1 for any state encountered except for two states (in $n60$ and $n70$) whose transitions with A_3 were still untried (Fig. 11). The constant selection of A_1 during the last 30 navigations shows that the controller correctly learned the superiority of A_1 over A_3 for the open environments.

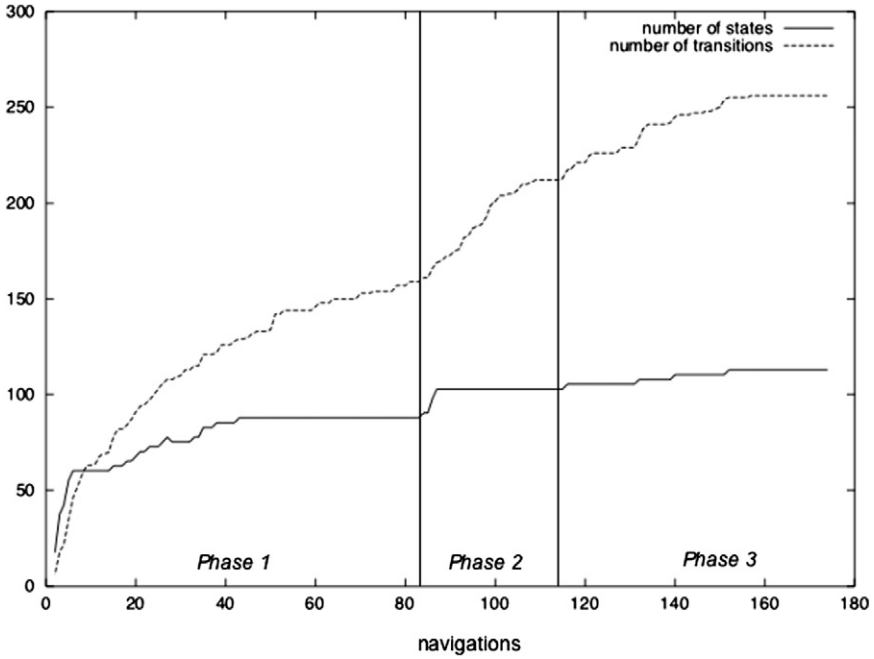

Fig. 10. Evolution of the size of the graph.

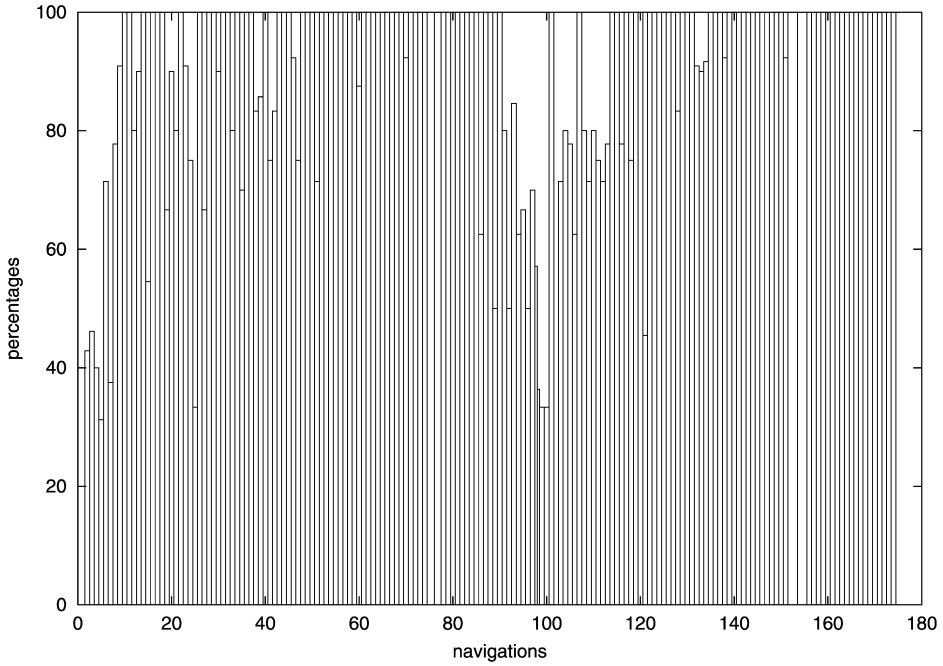

Fig. 11. Percentage of choice of A_1.

Two typical transitions in open environments are presented in Fig. 12(a) and (b). Figure (a) shows that continuous execution of A_1 gives the lowest average cost $C = 2.96$, whereas continuous execution of A_3 in figure (b) yields an average cost of $C = 11.43$. The skill transitions $A_1 \rightarrow A_3$ and $A_3 \rightarrow A_1$ produce higher costs because the robot must stop before transitioning to a new skill.

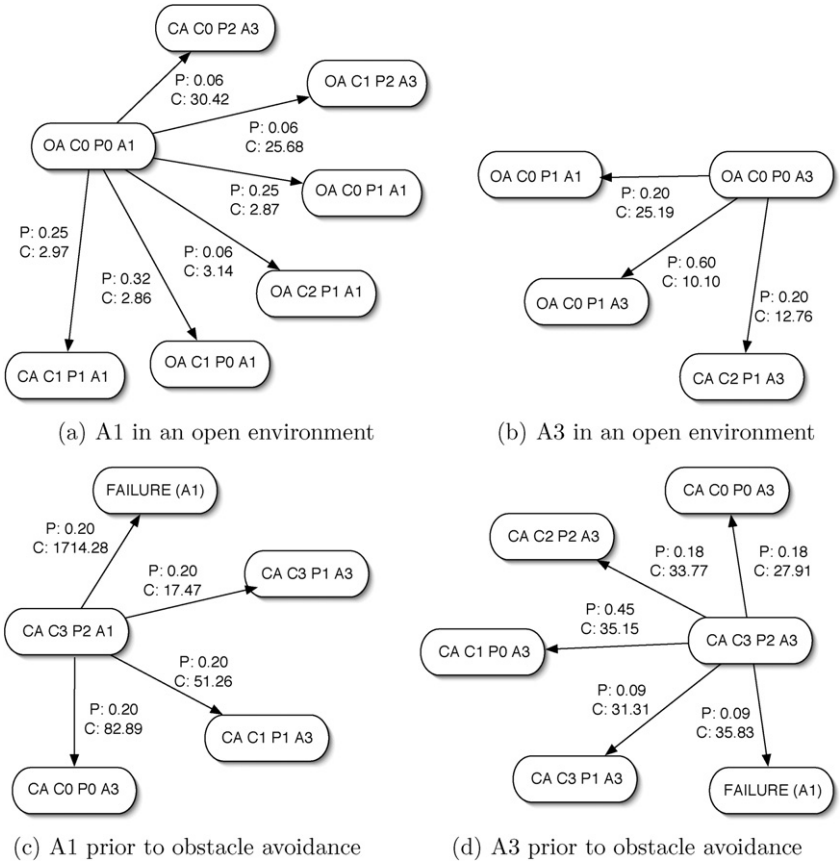

Fig. 12. Example of transitions.

Phase 2. Here 30 navigations are run in a modified environment where several obstacles are added. This new situation generates 14 new states between $n84$ and $n87$ and 10 new transitions are tried until $n93$. During these nine first navigations, six failures are recorded for A_1, each time from a state with a high level for the clutter feature. After $n101$ and until $n114$, each time a state with a high level for the clutter feature is encountered, the execution of A_1 is stopped by the controller and the obstacle avoidance is systematically performed with A_3. No more failures are recorded with A_1. As soon as the clutter feature returns to a low level, the computation of π switches back to a selection of A_1. A_1 is then kept as long as the value of the clutter feature stays low. Whereas in the previous phase A_1 was more appropriate than A_3, in this second phase the system is able to learn within 30 navigations the better efficiency of A_3 in cluttered environments.

Figs. 12(c) and (d) represent two transitions encountered immediately prior to obstacle avoidance. The current states for these transitions are characterized by the confined area navigation color (CA), the highest level of clutter ($C3$) and the most variable laser profile ($P2$). Figure (c) shows that the only way to avoid a failure for this transition is to switch to A_3. The example (d) shows that no failure is generated when A_3 is kept. The only failures reported in this situation occur when A_1 is executed.

Phase 3. The goal of the third step is to verify that the learning of the second phase (avoidance) did not corrupt the learning of the first phase (open navigation). The obstacles are removed to recover the same environment as in Phase 1, and 58 more navigations are performed. This new step shows that the learning in Phase 1 was not complete: 10 new states are created and 23 untried transitions are completed. Despite these untried transitions, between $n114$ and $n152$, 30 navigations are performed with 100% selection of A_1 by π. After $n151$, A_1 is constantly selected during the last 21 navigations. This last step shows that despite incomplete learning, the efficiency of A_1 in Phase 1 has not been forgotten after the learning in Phase 2.

7. Conclusion

ROBEL, the robot programming system proposed here, involves three components for achieving in a robust way a high-level task:

- A set of redundant *sensory-motor functions*, implemented through software modules within the Genom software engineering environment.
- A set of *skills* specified as HTN plans whose primitives are *sm* functions. These skills provide different ways of combining some of the available *sm* functions to achieve the desired task. Skills are complementary and provide good coverage of the diversity of execution contexts of that task.
- An observable finite MDP controller, learned through experience, which chooses dynamically at each control state the right skill for pursuing the task. It implements an original receding horizon control, by the repeated online planning of a closed-loop policy applied to the current state with respect to the updated current objectives.

This system has been fully implemented and deployed on an indoor mobile platform and experimented with in navigation tasks within a wide laboratory environment. The approach is fairly generic and illustrates the use of planning techniques in robotics, not for the synthesis of mission plans but for achieving a robust execution of the high-level steps of such mission plans. The learned controller is fairly generic and independent of the specifics of a particular environment. It characterizes the robot capabilities for the task. In principle it can be incrementally augmented with new *sm* functions and new skills.

The HTN planning technique used for specifying detailed skills to be followed by a controller for decomposing a complex task into primitive actions is fairly general and powerful. It can be widely applied in robotics because it can take into account closed-loop feedback from sensors and primitive actions. It extends significantly and can rely on the capabilities of the rule-based or procedure-based languages for programming reactive controllers, as in the system described here. In a first implementation, these HTNs were hand-programmed in Open-PRS. In a subsequent implementation, the offline synthesis of a set of skills from specified formal methods has been achieved using the Shop-2 planner [26]. The designer selects from this set the most relevant skills for which learning is performed.

The MDP planning technique relies on an abstract dedicated space, namely the space of control states for the navigation task. This abstract space is defined through a set of task-dependent features. It is independent of the robot environment. The size of the control space is just a few thousand states. Consequently, MDP algorithms can be used efficiently to replan online a closed-loop policy at each current state for the updated current objectives, as perceived in the control space. Furthermore, the learning stage of the probability and cost distributions in Σ is feasible at a reasonable cost. The drawback of these advantages is the *ad hoc* definition of the control space through task-dependent features, which requires very good knowledge of the sensory-motor functions and the navigation task. While in principle the system described here can be extended by the addition of new skills for the same task, or for other tasks, it is not clear how easy it would be to update the control space or to define new spaces for other tasks.

In addition to future work directions mentioned above, an important test of ROBEL will be the extension of the set of tasks to manipulation tasks such as "*open a door*". This significant development will require the integration of new manipulation functions, the design of redundant behaviors for these tasks and their associated control, and the extension of the control state. We believe ROBEL to be generic enough to support and permit such developments.

References

[1] R. Alami, I. Belousov, S. Fleury, M. Herrb, F. Ingrand, J. Minguez, B. Morisset, Diligent: Towards a human-friendly navigation system, in: IROS'2000, Takamatsu, Japan, 2000.
[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, F. Ingrand, An architecture for autonomy, International Journal of Robotics Research 17 (4) (1998) 315–337.
[3] A.G. Barto, S.J. Bradtke, S.P. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72 (1–2) (1995) 81–138.
[4] M. Beetz, Structured reactive controllers—a computational model of everyday activity, in: 3rd Int. Conf. on Autonomous Agents, 1999, pp. 228–235.
[5] M. Beetz, T. Belker, Environment and task adaptation for robotics agents, in: ECAI, 2000.
[6] T. Belker, M. Hammel, J. Hertzberg, Learning to optimize mobile robot navigation based on HTN plans, in: ICRA, 2003, pp. 4136–4141.
[7] C. Boutilier, T. Dean, S. Hanks, Decision theoretic planning: Structural assumptions and computational leverage, Journal of Artificial Intelligence Research 11 (1999) 1–94.

[8] C. Boutilier, R. Reiter, M. Soutchanski, S. Thrun, Decision-theoretic, high-level robot programming in the situation calculus, in: Proc. 17th National Conference on Artificial Intelligence (AAAI'00), Austin, Texas, 2000, pp. 355–362.
[9] S. Buck, U. Weber, M. Beetz, T. Schmitt, Multi robot path planning for dynamic environments: A case study, in: Proc. IEEE Intl. Conf. on Intelligent Robots and Systems, 2001.
[10] T. Dean, M. Wellman, Planning and Control, Morgan Kaufmann, 1991.
[11] S. Fleury, T. Baron, M. Herrb, Monocular localization of a mobile robot, in: IAS-3, Pittsburgh, USA, 1994.
[12] S. Fleury, M. Herrb, R. Chatila, Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture, in: IROS, 1997, pp. 842–848.
[13] A. Gabaldon, Programming hierarchical task networks in the situation calculus, in: AIPS'02 Workshop on On-line Planning and Scheduling, Toulouse, France, April 2002.
[14] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Morgan Kaufmann Publishers, 2004.
[15] K.Z. Haigh, M. Veloso, Learning situation-dependent costs: Improving planning from probabilistic robot execution, in: 2nd Int. Conf. on Autonomous Agents, 1998.
[16] J.B. Hayet, F. Lerasle, M. Devy, Planar landmarks to localize a mobile robot, in: SIRS'2000, Berkshire, England, July 2000, pp. 163–169.
[17] J. Hertzberg, H. Jaeger, Ph. Morignot, U.R. Zimmer, A framework for plan execution in behavior-based robots, in: ISIC-98, Gaithersburg, MD, 1998, pp. 8–13.
[18] F. Ingrand, R. Chatila, R. Alami, F. Robert, PRS: A high level supervision and control language for autonomous mobile robots, in: IEEE ICRA, St Paul (USA), 1996.
[19] T. Jaakkola, M. Jordan, S. Singh, On the convergence of stochastic iterative dynamic programming algorithms, Neural Computation 6 (6) (1994) 1185–1201.
[20] H. Jaouini, M. Khatib, J.P. Laumond, Elastic bands for nonholonomic car-like robots: Algorithms and combinatorial issues, in: 3rd Int. Workshop on the Algorithmic Foundations of Robotics (WAFR'98), 1998.
[21] L.P. Kaelbling, A.R. Cassandra, J.A. Kurien, Acting under uncertainty: Discrete Bayesian models for mobile-robot navigation, in: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 1996.
[22] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, R. Scherl, GOLOG: A logic programming language for dynamic domains, Journal of Logic Programming 31 (1997) 59–83.
[23] D.C. MacKenzie, R.C. Arkin, J.M. Cameron, Multiagent mission specification and execution, Autonomous Robots 4 (1) (1997) 29 V52.
[24] J. Minguez, L. Montano, Nearness diagram navigation (ND): A new real time collision avoidance approach, in: IROS, Japan, 2000, pp. 2094–2100.
[25] B. Morisset, M. Ghallab, Learning how to combine sensory-motor modalities for a robust behavior, in: M. Beetz, J. Hertzberg, M. Ghallab, M.E. Pollack (Eds.), Advances in Plan-Based Control of Robotic Agents, in: Lecture Notes in Artificial Intelligence, vol. 2466, Springer, 2002, pp. 157–178.
[26] B. Morisset, G. Infantes, M. Ghallab, F. Ingrand, Robel: Synthesizing and controlling complex robust robot behaviors, in: ECAI 2004, 2004.
[27] P. Moutarlier, R. Chatila, Stochastic multisensory data fusion for mobile robot location and environment modelling, in: Proc. Int. Symp. on Robotics Research, Tokyo, 1989.
[28] D.S. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, F. Yaman, SHOP2: An HTN planning system, Journal of Artificial Intelligence Research 20 (2003) 379–404.
[29] S. Quinlan, O. Khatib, Towards real-time execution of motion tasks, in: R. Chatila, G. Hirzinger (Eds.), Experimental Robotics 2, Springer, 1992.
[30] R. Reiter, Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems, MIT Press, 2001.
[31] W.D. Smart, L.P. Kaelbling, Effective reinforcement learning for mobile robots, in: Proc. IEEE Int. Conf. on Robotics and Automation (ICRA 2002), 2002, pp. 3404–3410.
[32] R.S. Sutton, Learning to predict by the methods of temporal differences, Machine Learning 3 (1988) 9–44.
[33] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, Cambridge, MA, 1998.
[34] S. Thrun, Learning metric-topological maps for indoor mobile robot navigation, Artificial Intelligence 99 (1) (1998) 21–71.
[35] S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Frohlinghaus, D. Henning, T. Hofmann, M. Krell, T. Schmidt, Map learning and high speed navigation in Rhino, in: D. Kortenkamp, R.P. Bonasso, R. Murphy (Eds.), AI-based Mobile Robots: Case Studies of Successful Robot Systems, MIT Press, 1998.
[36] C. Watkins, P. Dayan, Q-learning, Machine Learning 8 (1992) 279–292.