A Modular, Hierarchical, Reconfigurable Controller for Autonomous Robots

Denis Remondini and Alessandro Saffiotti

***Abstract*— Behavior-based systems are one of the most popular paradigms for building controllers. Although this paradigm is intrinsically tied to the notion of modular design, most existing behavior-based controllers do not fully satisfy the important desiderata of modularity, hierarchical structure, and reconfigurability. In this paper we discuss these desiderata, and we describe a fuzzy controller that satisfies them. To illustrate the functioning of our controller, we show experiments run on both a simulator and a real robot.**

Keywords: Mobile robots, behavior-based robotics, fuzzy logic, intelligent control.

I. Introduction

As service robots begin to leave the research laboratories and enter real home, public and industrial environments, a number of important technological issues must be addressed. One of those is that the control system for these robots should be designed in such a way as to make it easy to customize a robot for a specific task and/or environment, by re-using the existing components as much as possible. In particular, the control system should be highly *modular*, so that specific behaviors can easily be added, removed, or replaced without having to redesign the other behaviors of the entire control system. The system should have a *hierarchical* structure, which simplifies the construction of increasingly complex, modular behavioral components by the aggregation of simpler ones. Finally, the system should be *reconfigurable*, that is, it should be possible to change the overall behavior of the system on-line by dynamically changing some of its component behaviors, or the way they are combined together.

Behavior-based systems [1] are the *de-facto* standard for building controllers for autonomous robots. The behavior-based approach makes a significant step in addressing the three requirements above. As we discuss in the next section, however, most existing behavior-based controllers do not fully satisfy these requirements in practice.

In this paper, we propose a hierarchical controller for autonomous robots based on fuzzy logic, that satisfies the requirements of modularity, hierarchy and dynamic reconfigurability. The two key points in our development are as follows:

This work was partially supported by the Swedish KK Foundation.

D. Remondini is with Dip. di Elettronica e Informazione, Politecnico di Milano, Italy (email: *remondin@elet.polimi.it*).

A. Saffiotti is with AASS Mobile Robotics Lab, University of Örebro, Sweden (email: *asaffio@aass.oru.se*).

a) each behavior generates a rich representation of its *preferences* among the possible control values, instead of a single control value; preferences of different behaviors are fused to reach a consensus between them;
b) behaviors are written in Java, and a *factory method* pattern is used to allow the dynamic creation and modification of behaviors.

As we discuss below, point (a) enables the *modularity* and *hierarchical organization* of our behaviors, while point (b) improves *reconfigurability* by allowing the run-time creation and modification of behaviors, either manually by the developer or automatically by, e.g., a planner.

II. Meeting the Desiderata

Before we embark on the description of a controller that meets the above desiderata of modularity, hierarchical structure and reconfigurability, it is useful to discuss these desiderata to a deeper extent. In particular, the following three questions should be answered: (1) Are these desiderata really desirable? (2) Aren't they satisfied by the current systems? (3) How can we satisfy them?

Modularity means that each behavior in the controller can be designed independently from the others, possibly by different designers, and still we can add, modify, and remove it from the controller without the need to change the other behaviors. This is obviously a desirable property, which simplifies maintenance of the robot systems, reusability of behaviors, and sharing of behaviors among robots and designers.

As noted by Bonarini *et al* [2], real modularity is only partially achieved in most current behavior-based architectures. For example, in subsumption-based architectures [3] behaviors must be organized in a carefully designed suppression/inhibition network, whose structure depends on the hierarchical and causal relationships among the behaviors. This means that an independently designed behavior cannot be included in the behavior network in a plug-and-play fashion. A similar problem occurs with the architectures based on potential fields [7], [1], in which the relative gains of the behaviors depend on the specific nature of those behaviors.

One of the reasons why modularity breaks down in these approaches is that, when we combine behaviors together, we only use a small part of the information that was available to each behavior. For instance, an avoidance behavior confronted with an obstacle will usually consider the geometry of the obstacle in order to suggest a turning action. This information, however, is not visible outside the behavior, and

cannot be used to make an informed combination with, let's say, a goal achieving behavior. Several authors have proposed to solve this problem by allowing each behavior to produce a rich, multi-modal output [9], [11]. In our proposal below, we use the technique proposed by [11]. We also insist that the behaviors are self-contained and logically isolated, e.g., by only using local variables inside each behavior.

Hierarchical structure simplifies the top-down process allowing the designer to first concentrate on higher-level behaviors that implement a given strategy, and then on lower-level behaviors that realize the necessary actions. This structure also simplifies the bottom-up tuning and debugging, allowing the designer to first focus on the tuning of individual simple behaviors and then debug the behavior resulting from the combination of several simple behaviors.

Hierarchical structures are not uncommon in behavior-based systems. Already in the subsumption architecture [3], behaviors were organized in a hierarchy of increasing competences. This approach, like many others, has the drawback that the hierarchy was hard-wired, seriously limiting its flexibility. A more flexible structure, in which behaviors can call other sub-behaviors without restrictions, has been proposed in [12]. This is the approach that we follow in our system.

Reconfigurability means the possibility to dynamically modify the behaviors and the way they are combined at run-time. This has two important uses. First, it allows a much shorter tuning and debugging cycle when developing new behaviors or adapting existing ones to a new platform, task, or environment. Second, it allows the robot to autonomously generate the right behavior, or behavior configuration, for a given task and a given execution environment — e.g., using a task planner.

In most current systems, behaviors are written and compiled prior to execution, and they cannot be dynamically created or modified at run time. In some cases, a script language is used to define behaviors [4], [8], [13]. This, however, leads to either have to live with a limited language in the design of a behavior, or to develop an interpreter with a power comparable to the one of C. In the system described in this paper, we use a *behavior factory* combined with the class reloading feature provided by Java. This allows us to use a very expressive language inside a behavior and also to create and modify a behavior at run-time without restarting the experiment from scratch.

The Factory Method pattern is one of the *creational* design patterns described in the GoF catalog [6]. Creational design patterns abstract the instantiation process in order to make the system independent of the way in which its objects are created, composed and represented. Their main goal is to give flexibility in choosing how and when objects are created and which objects create them. We use the factory method pattern to have one single point where a behavior is loaded and to provide a great modularity to the controller. The factory receives as input the name of the requested behavior and returns an instance of the class that implements the behavior. As each behavior is a specialization of the abstract behavior class (see Fig. 3 below), we can use the polymorphism feature to execute a behavior without knowing how and when it was developed (it could be developed not only before, but also during the controller execution).

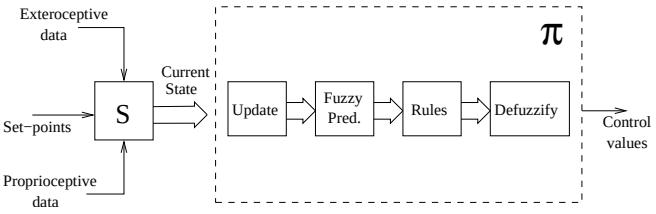

Fig. 1. The structure of a behavior.

III. Controller Design

We now introduce the behavior structure, its life cycle and the general structure of the controller that we have designed to satisfy the above desiderata.

A. Behavior structure

The internal structure of our behaviors is shown in Fig. 1. The data coming from the (exteroceptive and proprioceptive) sensors, together with the set-points coming from the higher layer, are collected in an internal structure S, called *state space*. Whatever data are represented in S, they can be used by the control policy π to generate a control $u \in U$, where U is the space of possible controls. At each control cycle, the S state is updated and a control value is decided via π.

A behavior contains a set of fuzzy rules that cover the relevant situations where the behavior will be used [11]. There are two kind of rules used in our controller: ground rules and meta rules. The difference between them concerns the rule consequent: in a ground rule the consequent is a fuzzy set, hence the rule evaluation gives directly the desirability of a value for a control variable. An example of ground rules can be seen in the *Face* behavior: its aim is to turn the robot towards a position or an object that is the behavior target. The behavior has the following rule set:

IF (NOT (OR TargetRight TargetLeft)) **THEN** turnStraight
IF (TargetLeft) **THEN** turnLeft
IF (TargetRight) **THEN** turnRight
IF (AlwaysTrue) **THEN** goStill

For each rule, the antecedent is evaluated and its truth value is used to weight the fuzzy sets contained in the rule consequent; the obtained result is the preference of the rule.

In a meta rule the consequent is a sub-behavior and the rule evaluation implies the activation of the sub-behavior that will generate a desirability function for each control variable. Examples of meta rules are:

IF (AND FacingDoor AlignedToDoor) **THEN** CrossDoor
IF (InsideCorridor) **THEN** FollowCorridor

We can use meta rules to build complex behaviors combining their preferences using the context-dependent blending (CDB, [10]) mechanism. This mechanism can be briefly characterized as follows:

i) at each state *s*, each behavior generates a *preference ranking* of possible controls from the perspective of promoting that behavior's objective;
ii) each behavior has a *context* of activation, representing the situations where it should be used;
iii) the preferences of each behavior are *discounted* by the truth value of its context in the current state *s*;
iv) the preferences of all behaviors, properly discounted, are *blended* to form a collective preference ranking;
v) one control is *chosen* from this collective preference, and sent to the effectors.

It is important to notice that the resulting choice is meant to reflect the *consensus*, rather than a tradeoff, between the preferences expressed by those behaviors.

In practice, our framework allows to have more powerful structures than pure CDB, such as a hierarchy of behaviors that contains a mixture of meta rules and ground rules, as follows:

IF (AND AngledRight LeftClear) **THEN** turnLeft
IF (AND AngledLeft RightClear) **THEN** turnRight
IF (AND (NOT AngledRight)
(NOT AngledLeft)) **THEN** turnStraight
IF (AlwaysTrue) **THEN** KeepVelocity

This gives the behavior designer more flexibility to design a set of behaviors that can be used to solve particular situations.

B. Behavior's life cycle

The *life cycle* of a behavior is composed of two different phases: behavior creation and behavior execution. Fig. 2 shows a diagram of a behavior life cycle.

The creation of a new behavior is made only once and it is the place where the rule set of a behavior is built. This process consists of parsing each rule and creating the required fuzzy predicates.

The execution of a behavior takes place at each control cycle of the controller and it is composed of three parts: (i) fuzzy predicate updating, (ii) rule evaluation, and (iii) generation of the behavior preference.

The first phase calculates the truth value of fuzzy predicates used in a behavior: they are a representation of the internal state of the behavior, that is, they represent information about the part of the physical world which is relevant to the behavior. The rule evaluation phase computes

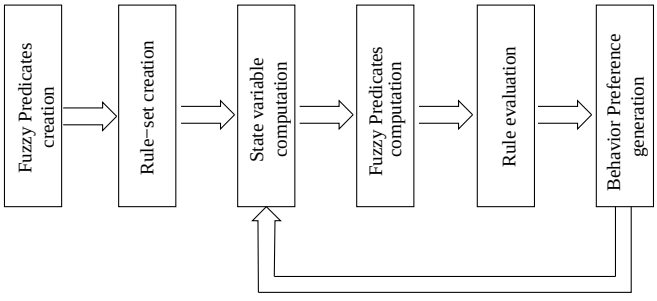

Fig. 2. The flow diagram of the behavior's life cycle.

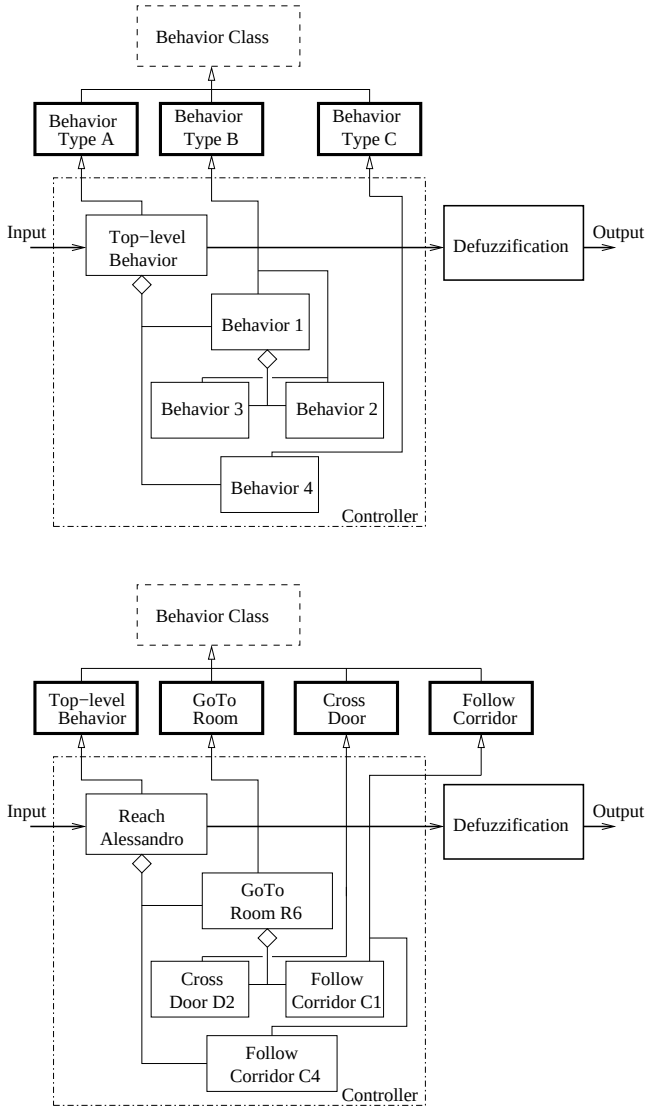

Fig. 3. The overall structure of the controller. (Top) Generic example. (Bottom) A concrete example.

the preference given by each rule of a behavior as seen above. Finally, once we have the preference of every rule, the overall behavior preference is given by a *combination* of the rule preferences.

C. Controller structure

The overall structure of our behavior-based controller is shown in Fig. 3, where we can also see the structure of the controller in a practical example (the robot must reach a person and to do that it needs to follow two corridors and then enter a door).

The top behavior is the one directly executed by the controller and it is composed of fuzzy meta rules: rules that have a behavior as consequent. In this way, we can build a hierarchical behavior that allows us to handle many situations with a simple tree structure. The top-level behavior execution implies the evaluation of each sub-behavior. Each sub-behavior generates a full desirability function over the

space of control values; once we have a desirability function for each active behavior, we use the CDB mechanism to obtain the overall set of command preferences to be applied. Finally, we use the defuzzification step to get the crisp values to send to the robot's effectors.

Notice that CDB allows us to add or remove behaviors without having to modify the coordination mechanism and hence it enhances the controller modularity.

IV. Behavior Design

The above structure allows the designer to write complex behaviors by instantiating and combining basic behaviors. Interesting, and thanks to the rule-based format of the behavior combination rules, this structure also allows the generation of a complex behavior for a given task to be done automatically, e.g., by a task planner.

A. Writing basic behaviors

In behavior-based approaches, it is always suggested to consider a behavior as a simple unit of control: if a complex control policy is needed, it should be obtained by composing a number of different basic behaviors.

A basic behavior should implement a simple control policy which respects these ideal requirements:

1) needs to maintain little or no internal state;
2) can be computed in bounded time;
3) pursues one single objective;
4) is designed to work in a relatively small set of environmental conditions.

We assume that any behavior has an objective: it is the purpose for which a behavior has been built. A "good" behavior must have a set of rules with some important properties: completeness, consistency, continuity.

A set of if-then rules is *complete* if there is at least one rule active in each situation that is relevant for the behavior. A set of if-then rules is *consistent* if it does not contain contradictions. A set of rules contains contradictions if at least one couple of rules has a non-empty intersection of their antecedents and an empty intersection of their consequents (see [5]). A set of if-then rules is *continuous* if the control function provided by the behavior is continue. This means that small variations of input variable values produce small variations of control values generated by the behavior execution.

Unfortunately, no methods exist yet to verify the above properties for a given set of fuzzy rules. In practice, however, the properties of completeness and consistency can be tested empirically on a reasonably large range of situations using a monitoring tool like the one discussed below (see Fig. 5). Continuity is typically guaranteed by the nature of fuzzy logic, provided that no crisp predicates are used in the rule antecedents.

B. Writing complex behaviors

Complex behaviors are built by combining simpler behaviors together using the CDB mechanism. We use meta rules to combine simpler behaviors in a hierarchical structure: in this way it is possible to reuse behaviors as black box units inside a more complex behavior. This structure gives us the chance to use the *divide-et-impera* paradigm, letting us to decompose a task into simpler parts, develop behaviors that run each single part, and then combine them to execute the original task.

This structure enhances the modularity, in fact each behavior (acting as a black box unit) inside a hierarchical structure can be replaced by another one that is functionally equivalent without the need to modify the other behaviors.

Rule antecedent	***Rule consequent***
(AT Goal)	Still()
(AND (AT Room1) (AND (ORIENTED Goal) (NOT (AT Goal))))	GoTo(Goal)
(AND (AND (AT Corridor1) (NEAR Door4)) (AND (ORIENTED Door4) (NOT (AT Room1))))	Cross(Door4)
(AND (NEAR Door4) (AND (NOT (AT Room1)) (AT Corridor1)))	Face(Door4)
(AND (AT Corridor1) (NOT (NEAR Door4)))	FollowCorridor(Corridor1)

Fig. 4. An example of B-plan used in our framework.

C. Plan-based behavior generation

In our framework we can write the top-level behavior by hand or we can create it automatically. The first method can be useful in some particular cases, for example when we need to test some behaviors in a specific situation, so we can create a specific environment and write the relative top-level behavior that allows us to make our test.

An alternative option is to create the top-level behavior automatically. This is especially useful when the robot has a suite of behaviors, and it must achieve a goal that is known only at runtime. In our work, we assume that there is a process able to create a plan in the form of a set of *situation-action* rules that specify the course of action in each situation, in order to achieve a given goal. Such a set of rules, called a *B-plan* in [10], can be directly translated into a top-level behavior of the type that can be executed by our controller.

Fig. 4 shows an example of B-plan. We can identify several rules, and each of them is composed of: an activation context, the name of the behavior that has to be activated in that context, the name of the object necessary for the behavior to do its work, and some parameters (not shown). To create the top-level behavior that executes a given B-plan, we follow these steps: (1) parsing the B-plan; (2) creation of the behavior rules; (3) association of arguments and parameters to the sub-behaviors.

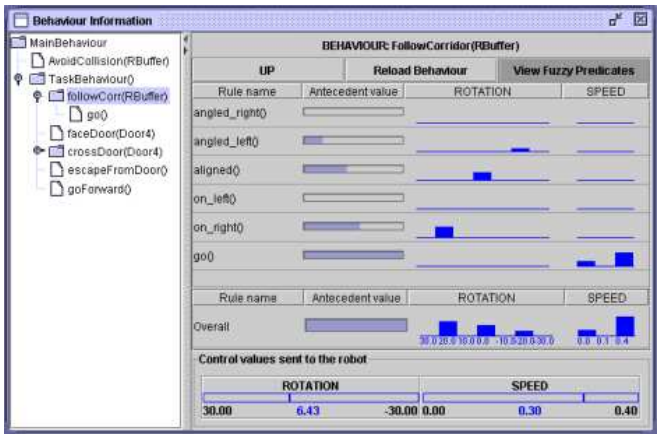

Fig. 5. Behaviors monitoring tool

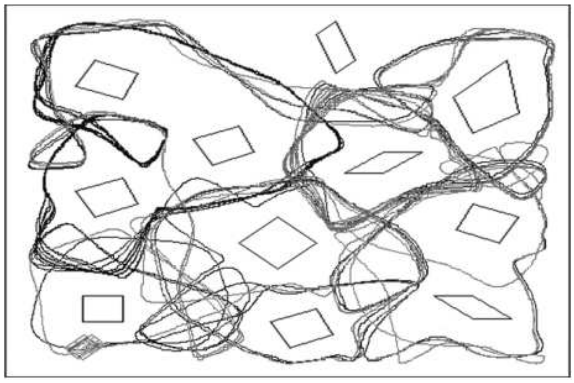

Fig. 6. Wander behavior on simulator.

V. An experimental system

Our fuzzy hierarchical behavior-based controller is a general stand-alone controller, which can be integrated in whatever framework once a suitable interface to the sensors and actuators is provided. In order to validate it, however, we have integrated it inside a specific robot architecture: the Thinking Cap II. In this section, we show a few qualitative tests run on both a real robot and a simulator.[1] Thinking Cap II is based on the fuzzy architecture described in [10], and has been implemented in Java. In our experiments we noted that, although Java is not so performant as C or C++, it can afford a control cycle of 100 ms, allowing fluent robot movements.

A. Using the controller

Fig. 5 shows the main window of the *monitoring tool*, the main interface by which we can inspect and debug the activity of our controller. It is easy to identify four different areas:

1) on the left there is a tree that shows the hierarchy of behaviors executed by the controller;
2) on the top right there are three buttons: the first one on the left is used to go up in the hierarchy, the second one sends the "Reload Behavior" command to the controller, and the third one opens a window containing the information about the fuzzy predicates used in the monitored behavior;
3) on the bottom right there is the visualization of the control values sent to the robot's effectors;
4) the main body of the panel on the right displays all the information about a specific behavior.

The "Reload Behavior" feature is particularly interesting. This is typically used during testing, when it is discovered that a change is needed in a behavior; the designer modifies the behavior and then he asks to the controller to reload it: the controller loads the behavior without stopping its execution. This feature makes behavior development much faster.

The reload mechanism works as follows: once our controller gets the request for a behavior reload, it uses a behavior factory that compiles the requested behavior at run-time, loads and integrates it into the controller. This is possible due to utilization of the Factory Method pattern and of the Java dynamic class loading feature.

B. Simulated experiments

The first experiments to test our controller have been made using the simulator: for example, we have extensively tested the Avoid Obstacle behavior in simulated environments with static obstacles whose positions were unknown to the robot. In this experiment we used the Wander behavior as top-level behavior. The Wander behavior is composed of two sub-behaviors, AvoidObstacles and KeepVelocity, and its goal is to go around avoiding obstacles and trying to maintain a constant velocity.

The rule set of the Wander behavior is composed of the following two meta rules:

IF (NOT nearObstacles) **THEN** KeepVelocity
IF (nearObstacles) **THEN** AvoidObstacle

Fig. 6 shows a run of the Wander behavior in a simulated environment with static obstacles whose positions were unknown to the robot. In this run, the robot covered an overall distance of 1.32 Km at an average speed of about 200 mm/sec without a single collision.

C. Real robot experiments

We have also tested our system on a real robot in an indoor office environment.[2] The robot used was an iRobot ATRV-Jr. This is a robot for mixed indoor-outdoor use with four driving wheels and skid steering. It is equipped with a rich set of sensors, including a ring of 17 sonars, a SICK laser range finder, a GPS receiver, a stereo camera mounted on a pan-tilt unit, and an inertial navigation system. It has a maximum linear velocity of 1.7 m/s and maximum rotational velocity is 120 degrees per second.

We report here one of the experiments made to test a complex behavior designed to execute a task consisting of

[1] Thinking Cap II is a joint effort between the University of Örebro, Sweden, and the University of Murcia, Spain. See http://robolab.inf.um.es/tc2/.

[2] Videos made during our experiments with the real robot are available on http://www.rabotat.org/MScThesis.html

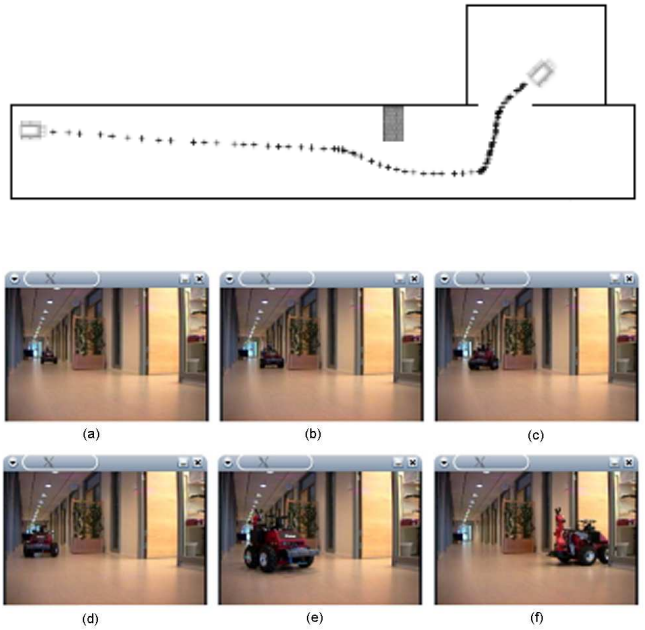

Fig. 7. Top: path generated during the real robot experiment. Bottom: image sequence taken by an external camera, situated on another robot, during execution.

following a corridor, facing and entering a door situated on the left side of the corridor. The robot has only the position of the door as a priori information.

Fig. 7 shows the path generated by the robot (top), and a sequence of the images taken by an external camera (bottom), during the execution of the task. The corridor in which the robot executes the task is a standard corridor with many offices, some glasses, and also a flower vase situated near the door that the robot must enter. Fig. 8 shows the activations of the behaviors over time: at the beginning only the FollowCorridor behavior is active and this situation persists until the robot approaches the door, then the Face and the CrossDoor behavior are activated in sequence to enter the door. Finally, the EscapeFromDoor and GoForward rules are activated to bring the robot into the new room beyond the door.

VI. Conclusions

Our approach allows a designer to build complex controllers that meet the three desiderata of modularity, hierarchical structure, and reconfigurability. The hierarchical structure is based on meta rules that provide a flexible way to define increasingly complex behaviors, and that consider behaviors as black box units that can be replaced by other functionally equivalent behaviors. Black-box modularity is enhanced by definition of self-contained behaviors and by the CDB mechanism for combining behaviors, which makes more of the internal information in a behavior available to the external combination mechanism. Finally, reconfigurability is attained using a behavior factory combined with the class reloading feature provided by Java.

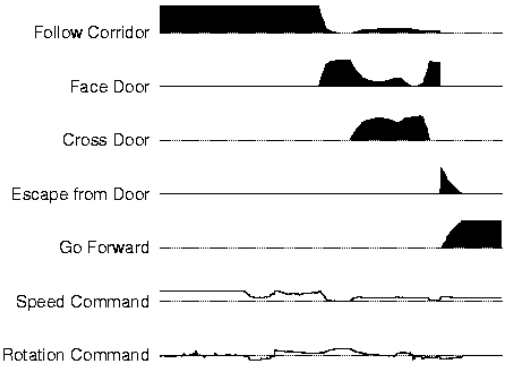

Fig. 8. The temporal evolution of the activation level of the rules of the complex behavior used to execute the real robot experiment.

While these principles have been used and demonstrated in the implementation of a specific fuzzy controller, we are confident that they are general enough to inspire others to develop behavior-based systems which better satisfy the three desiderata above.

VII. Acknowledgment

The authors are grateful to Humberto Martínez Barberá for his help with the Thinking Cap.

References

[1] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[2] A. Bonarini, M. Matteucci, and M. Restelli. A novel model to rule behavior interaction. In *Proc of the Int Conf on Intelligent Autonomous Systems (IAS)*, pages 199–206, Amsterdan, NL, 2004.

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

[4] R. A. Brooks. The behavior language: User's guide. Technical Report AIM-1227, MIT, 1990.

[5] D. Driankov and A. Saffiotti, editors. *Fuzzy Logic Techniques for Autonomous Vehicle Navigation*, volume 61 of *Studies in Fuzziness and Soft Computing*. Springer-Verlag, Berlin, Germany, 2001.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Int Journal of Robotics Research*, 5(1):90–98, 1986.

[8] K. Konolige. COLBERT: A language for reactive control in sapphira. In *KI - Kunstliche Intelligenz*, pages 31–52, 1997.

[9] J. K. Rosenblatt and D. W. Payton. A fine-grained alternative to the subsumption architecture for mobile robot control. In *Procs of the IEEE Int. Conf. on Neural Networks*, volume 2, pages 317–324, Washington, DC, 1989.

[10] A. Saffiotti, K. Konolige, and E. H. Ruspini. A multivalued-logic approach to integrating planning and control. *Artificial Intelligence*, 76(1-2):481–526, 1995.

[11] A. Saffiotti, E.H. Ruspini, and K. Konolige. Blending reactivity and goal-directedness in a fuzzy controller. In *Proc of the 2nd IEEE Intl Conf on Fuzzy Systems*, pages 134–139, San Francisco, CA, 1993.

[12] A. Saffiotti and Z. Wasik. Using hierarchical fuzzy behaviors in the robocup domain. In D. Maravall C. Zhou and D. Ruan, editors, *Autonomous Robotic Systems*, pages 235–262. Springer-Verlag, 2003.

[13] A. Skarmeta, H. Martínez Barberá, and M. Alonso. A fuzzy agents architecture for autonomous mobile robots. In *Procs of the World Congress of the Int Fuzzy Systems Association (IFSA)*, Taipei, TW, 1999.