Specifying Failure and Progress Conditions in a Behavior-Based Robot Programming System

(Extended Abstract)

Froduald Kabanza and Khaled Ben Lamine
University of Sherbrooke
Sherbrooke, Quebec, J1K 2R1 Canada
{Kabanza, Khaled.Ben.Lamine}@usherbrooke.ca

Abstract. Behavior-based robot programs are designed by splitting the overall robot decision making process among concurrent processes, each of them being in charge of a well defined simple goal-oriented behavior. However, the combination of simple behaviors into more complex ones often incur global interactions that cannot be debugged by just considering individual behaviors. The normal way of dealing with global interactions that may cause failure in the robot execution is to program additional heuristic processes that monitor behaviors to check that they progress normally towards their intended goals. In this paper, we explain how Linear Temporal Logic (LTL) can be used as a declarative language for specifying an interesting class of such monitoring processes. This simplifies the task of writing robot control programs, increases the design modularity by clearly separating the control components from the monitoring ones, and augments the reliability of control programs thanks to a precise and clear LTL semantics.

1 INTRODUCTION

Behavior-based robot programming systems follow the principle that the overall robot decision making process should be split among several concurrent processes, each of them being in charge of a well defined simple goal-oriented behavior; it is then the combination of theses processes that achieves more complex task-oriented behaviors [1, 7, 10, 15]. An interesting feature that explains in part the increasing popularity of this approach is undoubtedly the modularity of robot programs that are designed along this principle; one designs a program that controls one behavior, without having to get into details of interacting behaviors that control the other robot aspects. For instance, one can program a *goto* behavior that combines with *obstacle-avoidance* behaviors to make a robot go to a desired destination, yet without getting involved into the code of obstacle-avoidance processes.

The split of behaviors among many concurrent processes introduces, however, all the known hurdles of concurrent processes. Individual behaviors that are tested or proven to work correctly by assuming local conditions may no longer work perfectly when combined with other behaviors. In fact, most complex tasks require heuristic processes that monitor the robot behaviors to check whether they progress normally towards their intended goals, and trigger failure-avoidance or failure-recovery behaviors whenever necessary [3, 11, 13, 16]. However, programming such robot monitoring behaviors still remains a difficult task, because of the complexity of process interactions, which is often exacerbated by the intrinsic imprecision in robot sensors, actuators and positioning devices or the uncertainty in the robot environment.

In order to facilitate the programming of robot monitoring processes, we are trying to develop a general framework for specifying declarative robot monitoring conditions using Linear Temporal Logic (LTL) [12]. We use LTL to specify conditions that a normal, satisfactory robot execution should satisfy; then we monitor those statements in the background of the robot execution to notify of their violation. Processes waiting on such notifications can then be activated to avoid anticipated failures or to recover from them. LTL is expressive enough to handle an interesting subset of monitoring conditions, such as waiting until a particular sequence of state conditions has been observed.

With this approach, the implementation of a monitoring process will not require elaborate knowledge of the internal program structure of the processes being monitored. This approach also seems quite flexible, allowing the specification of both light-weight synchronous monitoring processes as well as more complex asynchronous ones. It is quite simply implemented, without any explicit storage of the robot history; instead, the relevant state features of the history are automatically conveyed by the update of monitoring conditions which are added to the robot internal state. These features make our approach fit naturally with the behavior-based architecture that we use, thus requiring little learning effort from robot software programmers to use our approach for coding logic-based monitoring processes.

The remainder of the paper is organized as follows. In the next section we discuss SAPHIRA [10, 15], the robot programming system used to implement our approach. Then, we discuss the use of LTL to express robot monitoring conditions; we discuss the basic algorithms that are used to handle those formulas in robot monitoring activities, as well as the integration of these algorithms into the SAPHIRA architecture. We conclude with a discussion on most related work and on future directions of this research.

2 SAPHIRA ARCHITECTURE

SAPHIRA is a programming system for mobile robots, developed by Konolige and his team at SRI [10, 15]. It includes libraries for controlling a mobile robot at different levels of complexity, going from low-level (e.g., moving a given distance, turning the wheels a given angle, or acquiring raw sensor data) to more complex navigation behaviors (e.g., obstacle avoidance, map registration, path planning) and tasks (e.g., sequencing behaviors using arbitrary C++ programs). Figure 1 shows an abstract view of the SAPHIRA architecture. SAPHIRA processes can be synchronous or asynchronous. They all have access to the robot state, which consists of the state reflector (position encoder readings, wheel orientation, sonar readings), and the environment representations (map information, sonar interpretation data). State information is updated automatically from robot sensors and from SAPHIRA synchronous or asynchronous processes.

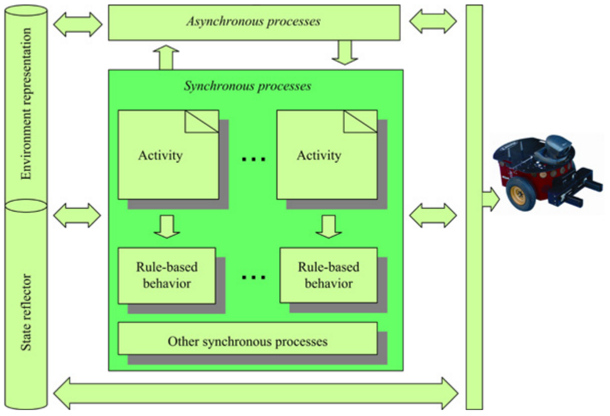

Figure 1. SAPHIRA architecture

Synchronous processes are normally used to implement lower-level behaviors that require immediate action on the part of the robot, such as obstacle avoidance or object tracking; they are managed by a SAPHIRA specific scheduler. Asynchronous processes on the other hand are normally used to implement higher-level behaviors (e.g., a task planner deciding on the order objects are picked and delivered by a robot in an office delivery application); they are managed by the host computer operating system (e.g., Linux or Windows).

The synchronous scheduler iterates through each synchronous process every 100 milliseconds. At each cycle, it runs each process, one at time, from the execution point where it suspended it last time to the next suspension point; a priority mechanism is used to resolve conflicts among processes that simultaneously affect the same robot control parameters.

The rationale behind the 100 milliseconds cycle becomes obvious by considering the obstacle-avoidance behavior; if we code an avoidance strategy consisting in turning left each time the sensor readings indicate an obstacle on the right, we would like the time between the detection of the obstacle (one block of instruction executed during one cycle) and the turning of the robot wheel and change of its speed (another block of instruction executed in the next cycle) to last at most one 100 milliseconds, so as the robot's reaction is fast enough before it hits the obstacle. Hence, when writing synchronous processes, one must ensure that the code between two suspension points is fast enough to be executed within the 100 milliseconds cycle.

Synchronous processes are programmed using either rule-based C++ libraries, a declarative C-like language called Colbert, or finite state machines (FSM) which are actually internal representations of processes written using the rule-based C++ libraries and are rarely convenient to code in directly. Asynchronous processes are normally written in higher-level languages (e.g., C++ or Java). In SAPHIRA terminology, rule-based processes are called behaviors, whereas those written using Colbert are called activities. In this paper, we adopt a formal software engineering terminology [13]; a behavior is a set of possible execution sequences that a given SAPHIRA process may go through when interacting with other processes.

3 MONITORING PROCESSES

A robot task is programmed by combining several concurrent SAPHIRA processes. For instance, moving a robot to a goal location can be done by combining one behavior that avoids obstacles and one behavior that attracts the robot towards the goal location; the obstacle avoidance behavior is often further split into avoiding close obstacles and staying away from remote obstacles. Adding corridor following, object-seeking, object-grasping and object-releasing processes makes the robot become an object delivery system. Although such a splitting of behaviors is a purely reactive, heuristic one, in many cases it is sufficient to move the robot fast towards its target, while avoiding obstacles. However, in unusual obstacle configurations, the robot can get trapped, oscillating between the obstacle avoidance and goal-tracking behaviors.

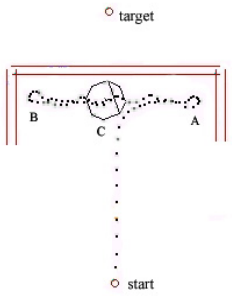

Figure 2. Example of navigation failure

Figure 2 illustrates this situation with a U-shape obstacle configuration. The robot is shown as a circle, with a crossing line indicating its heading direction. The dotted curve indicates the robot trajectory so far. Using rule-based C++ goal-reaching and obstacle avoidance processes provided with the SAPHIRA distribution, we reproduced an experiment made by Xu [17]. The robot starts from the indicated point on the figure with the goal of reaching the indicated target position. At the start, the U-shape obstacle is too far away to have an impact on the obstacle avoidance behavior; hence the robot moves in a straight line towards the target only under the effect of the goal-reaching behavior. As it gets closer to the obstacle (bottom of the U-shape), the obstacle avoidance veers the robot to the right to avoid the obstacle (it could also have veered left). The robot continues moving along the bottom of the obstacle, towards the right. On point A, because of the obstacle in front, the robot veers left (being attracted by the goal-reaching behavior), then because of the obstacle still on the left, abruptly veers right, making its heading direction opposite to the target direction. There are no more obstacles in front, and the robot becomes attracted again towards the target, bringing it back to point C. Since it approaches the bottom slightly inclined on its right, it will tend to veer left this time, making a move that is a mirror to the previous one, this time with critical point B playing the role of critical point A. The robot keeps on oscillating this way, between points A, C and B endlessly.

This is kind of situations is not limited to behavior-based robots. It is actually a problem for any navigation approach based on local decisions such potential field methods [11]. If the obstacle configurations are known, we can use path-planning to escape from such situations. For unpredictable obstacles (e.g., in office delivery environments people can move freely and objects may be displaced without notice), the robot behaviors have to be coupled with monitoring processes to detect such U-shaped obstacle configurations in order to activate recovery strategies.

3.1 Program-based monitoring processes

We can detect and escape from U-shape obstacles by using the virtual target approach by Xu [17]. The idea is to monitor the occurrence of the above A and B points in the robot behaviors and then to temporally set a virtual escape target for the robot (see Figure 3).

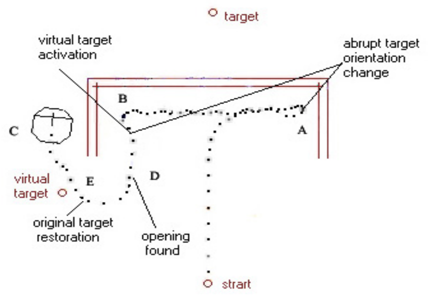
target
virtual target activation
abrupt target orientation change
B
C
A
D
E
virtual target
original target restoration
opening found
strart

Figure 3. Virtual target approach

To experiment with this idea, we wrote a Colbert process (i.e., more precisely a Colbert activity, named *DetectUshape*) which sets a global Boolean variable (*UshapeDetected*) when such points A and B have occurred over a period of time; another Colbert process (*SetVirtualTarget*) waits on *UshapeDetected* becoming true to change temporally the robot target to a virtual target opposite to the current one and to activate another Colbert process (*DetectOpening*), which moves the robot along one side of the U-shape (much like a corridor following behavior), trying to detect an opening; once an opening is detected (point D), this is notified to *SetVirtualTarget***,** which restores the original target. Colbert includes useful primitive for programming the above processes, C-like *while* and *if* flow control instructions, instructions for suspending and resuming processes, instructions for acquiring process states, global variable definitions for inter-process data sharing, and a very useful *waitFor* instruction allowing a process to be suspended until a Boolean condition becomes true.

The above solution still remains simple in many aspects (it's not difficult to trap the robot with a more complex obstacle configuration), nevertheless it illustrates that even very lower-level robot control can deal with logical patterns of behaviors which could be abstracted over using declarative statements.

3.2 LTL-based monitoring processes

The process *DetectUshape* monitors a failure condition for robot navigation that can be declaratively expressed as "*eventually, the robot's heading direction keeps on changing abruptly from left to right, or from right to left, immediately followed with the heading opposite to the target*". We want to simplify the program by replacing the process *DetectUshape* by a simple *wait* instruction on such a declaratively stated condition.

This statement specifies something that should not happen in a normal execution; it's a failure condition. We can also express this from a perspective of something that should be maintained true in a normal behavior: "*the robot's heading direction is never continually changing abruptly from left to right, or from right to left, immediately followed with the heading opposite to the target*"; this is a progress condition. Failure conditions and progress conditions are dual, but it's nice to have both of them in a tool for monitoring behaviors, since some execution properties are better captured as failures while others are better expressed as progress conditions.

By integrating declarative failure and progress conditions as Colbert and C++ primitives, we can use them to code succinct monitoring processes, to develop prototypes using them in a development phase and later replace them by Colbert or C/C++ code, or to test the correctness or performance of a robot. For instance, we can simulate a robot control program on randomly generated object delivery requests using a statement like "always when a delivery request is received, it is fulfilled within 20 seconds". We can also express things like "when grasping and delivering object, the robot should wait until an object is visible for five consecutive SAPHIRA cycles, before approaching it;" or "the robot must wait in the corridor until door to room B is opened."

3.2.1 Specifying LTL conditions

The execution of a robot process produces a sequence of robot states. We can thus express failure and progress conditions as declarative statements over sequences of robot states. In Colbert and C/C++ processes, we can already write conditions relevant to just one state using Boolean expressions (in Colbert they have the same syntax as in C/C++). Normal Colbert or C++ Boolean expressions form the basic case for LTL conditions, called *state conditions*. For instance, if x is an integer declared in Colbert, then *(x== 1||x==2)* is an LTL condition. All Boolean variables are also LTL conditions.

LTL conditions are expressed from the perspective of the current robot state. A state condition is true at a current point of execution if the corresponding Boolean expression evaluates to true. *Backward conditions* express properties with respect to the execution history of the current state, whereas *forward conditions* express properties with respect to what will happen from the current state.

A backward condition is a condition involving the logical operators **L** (last), **S** (since), **P** (previous) and **G** (all the time) to refer to the history states. The syntax and intuitive semantics are quite simple. If **c** is a state condition or a backward condition, then **(L c)** is a backward condition, which is true in the current state if **c** is true in the preceding state. If **c** and **d** are state conditions or backward conditions, then **(c S d)** is a backward condition, which is true in the current state if **c** has been true in each previous state since when **d** was true. If **c** is a state condition or a backward condition, then **(P c)** is a backward condition, which is true in the current state if **c** is true in some previous state. If **c** is a state condition or a backward condition, then **(G c)** is a backward condition, which is true in the current state if **c** is true in all previous states. Finally, backward conditions can be combined using the usual Boolean operators **II** (or), **!** (negation) and **&&** (and). The operator **L** is only used in synchronous processes; by preceding state, it then means the state available at the preceding SAPHIRA cycle. For asynchronous processes, SAPHIRA states are sampled at arbitrary periods.

Conditions can be nested arbitrary, making the resulting semantics a recursive one, with a double basic case on state conditions and on the start state of the execution. This gives us a quite powerful language for expressing behavior properties. For example, we can express the abrupt direction change in the U-shape obstacle failure by using the condition

(P (targetRealLeft && (L targetRealRight))) ||
(P (targetRealRight && (L targetRealLeftt))),

where *targetRealLeft* and *targetRealRight* are Boolean conditions over the robot current position and the target, expressing respectively that, the target is on the rear left of the robot, or on the real right; thus the disjunct *(P (targetRealLeft && (L targetRealRight))* expresses that there is a previous state in which the target was on the rear left and on the rear right in the state just before.

A forward condition is a condition involving the logical operators **N** (Next), **U** (until), **E** (eventually) and **A** (always) to refer to the history states. The syntax and intuitive semantics are also simple. If **c** is a state condition or a forward condition, then **(N c)** is a forward condition, which is true in the current state if **c** is true in the next state. If **c** and **d** are state conditions or forward conditions, then **(c U d)** is a forward condition, which is true in the current state if **c** true in all forward states preceding the first state, if any, where **d** is true. If **c** is a state condition or a forward condition, then **(F c)** is a forward condition, which is true in the current state if **c** is true in some future state. If **c** is a state condition or a forward condition, then **(A c)** is a forward condition, which is true in the current state if **c** is true in all future states. Forward conditions can also be combined using the usual Boolean operators. The operator **N** is only used in synchronous processes; by next state, it then means the state available at the next SAPHIRA cycle.

Forward conditions are like mirror conditions to backward conditions with respect to the current state. In fact, in our case, any property expressed as a backward condition can also be expressed as a forward condition and vice-versa.[1] The previous U-shape example can be expressed forwardly as

(F (targetRealRight && (N targetRealLeft))) ||
(F (targetRealLeft && (N targetRealRight))).

Even though backward conditions and forward conditions have the same expressive power, it is useful to have them both because some behavior properties are better specified by telling what should happen or not happen for any future execution sequence seen from the start state (i.e., the current state in interpreting the condition), whereas other properties are better expressed by stating what bad sequence of states must have occurred in the past, before concluding in a failure or normal progress in the current state.

[1] This equivalence holds because for **(F c),** c is not required to eventually become true (what is required is that if this ever happens, then we must be able to detect it); similarly, in **(c U d)**, **d** is not required to hold eventually. Without these conditions, forward conditions are slightly more expressive than backward conditions.

3.2.2 Progressing conditions

The above example and the intuitive semantics of this language suggest that we would have to store explicitly the sequence of SAPHIRA states in order to evaluate the truth of LTL conditions. In fact, we need not. Given an LTL formula and a current state, it is possible to tell whether the formula is true in the current state, whether it is false, or whether none of these two situations can be decided yet, by simply computing an update of the condition to be evaluated, state by state. The function that computes such an update condition for backward or forward formula is called a *condition progress function*; it progresses a condition along a sequence of execution on the fly until being able to establish its validity or falsity.

The technique for progressing forward formulas is well-known and has been used in many problems, including robot perception planning [5] and robot monitoring [3]. The update rules are actually quite easily derived from the forward recursive syntax rules and recursive semantics of LTL.

For backward conditions a different technique is needed. We can track the truth of a backward condition over a history, on the fly, forwardly, starting from the initial state of an execution, yet without actually keeping an explicit record of the execution trace. Instead, the necessary information for evaluating the backward condition at a given point of execution will be conveyed by a set of past sub-conditions that are updated at every step of execution.

The idea is first to realize that the truth value of a backward condition is completely determined by the truth value of its sub-conditions. Initially, we determine the sub-conditions that are true in the initial state. For example, **(L c)** is initially false since there is no previous state; a Boolean condition **c** is initially true if it holds in the initial state; **(s S d)** is initially true if **d** is true in the initial state; similarly for **P** and **G** conditions. It takes a constant time to compute the initial set of sub-conditions (exactly one run over the condition).

Next, we pass the set of sub-conditions true in the current state to the next state of execution; we say that we have progressed the set of sub-conditions. Given this, we can determine whether a backward formula is true, false or none of these yet, by evaluating Boolean conditions as usual in the new state and backward sub-conditions using their membership in the progressed set of sub-conditions. This also takes a constant time.

3.2.3 Integration into SAPHIRA

Asynchronous processes are programmed in C/C++ (or other high-level languages that allow dynamically linked libraries). On the other hand SAPHIRA Colbert synchronous processes can invoke arbitrary C/C++ functions and have access to C/C++ structures via dynamically linked libraries. Thus, once one has implemented the LTL progression algorithms for forward conditions and backward conditions, it is not hard to make an interface between them and Colbert or with C/C++. Currently we have only the forward progression functions implemented, and only for Colbert processes.

We programmed in C++ a structure for an LTL condition, indicating the type (backward or forward, although only forward ones are supported at the time being), its mode (failure condition or progress condition) the original condition, the progressed condition (by the LTL progression function, this slot becomes true in state where it is satisfied, false in states where it falsified, an LTL condition otherwise) and other few bookkeeping attributes.

The declaration of an LTL condition initializes the appropriate structure and returns a pointer to it. This pointer can then be used in a *waitLTLCondition*. If **c** is a failure condition, then the instruction *waitLTLCondition(c)* in a Colbert process blocks the process until **c** is progressed to false (which means it is made false in the current state). If **c** is a progress condition, then the instruction *waitLTLCondition(c)* in a Colbert process blocks the process until **c** is progressed to true (which means it is made true in the current state). After the process has passed the wait condition, it can execute user-specified code for handling the condition. Figure 4 illustrates the integration into the SAPHIRA architecture.

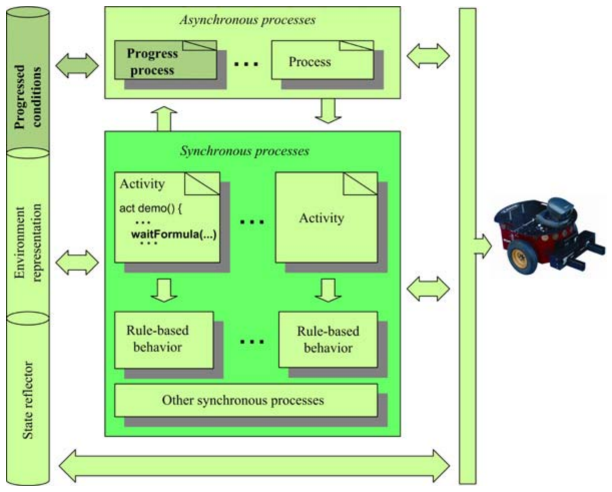

Figure 4. SAPHIRA architecture with LTL Progress

4 RELATED WORK

Earlier steps of this work were reported in [3,4]. At that time, our system only supported asynchronous monitoring processes, required to record a history of the robot execution (this increases the monitoring complexity), and only handled forward LTL conditions. Even though handling backward conditions does not modify the theoretical expressive power, this allows easier specifications in many cases, particularly when coding robot behaviors based upon past observed states.

Many task-level control languages have been proposed allowing flexible specifications of robot tasks. Among them, the Task Definition Language (TDL) of Simmons and Apfelbaum is a C++ extension [16] with various useful synchronization primitives that facilitate the specification of robot monitoring processes. The language does not support however a declarative specification of behavioral properties such as forward or backward conditions; these have to be encoded directly as programs. Other frameworks that involve formal methods in robot monitoring include the Reactive Plan Language in [2], the probabilistic perception action planner in 54], the extraction of symbolic facts from a history of behaviors' activations [9], and robot monitoring of Golog robot control programs [5]. Both RPL and Golog appear to be more flexible for task-level problems. The work in [11] is much closely related to the framework we propose in that it provides a mechanism for extracting fluents from behavior activation levels; a connection could then be made with our approach by using such fluents as the basis for propositions in monitored progress conditions.

5 CONCLUSION

This work is being continued along many avenues, including the following. With synchronous processes, LTL conditions are progressed by Colbert activities. This increases the size of the SAPHIRA stack, consuming precious time of the 100 milliseconds cycle. Since not all conditions need update at every SAPHIRA cycle, we can progress non critical conditions asynchronously, still allowing access of SAPHIRA synchronous processes to the result of this progression.

It also seems feasible to add timing conditions by introducing a global clock variable and having it involved in Boolean conditions that compose progress conditions. For instance, if the clock is initialized to 0 at the start of the robot executions, with units in seconds, we can have a formula like *F=(A(nearTarget || clock < 600))*, where *nearTarget* is a Boolean expression expressing a desired nearness between the robot position and a target position. If this is declared as a failure condition in a process, the instruction *waitLTLCondition(F)* would block the process until 600 seconds have elapsed before the robot is near the target.

6 REFERENCESS

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

[2] M. Beetz. 'Structured reactive controllers: controlling robots that perform everyday activity.' *Agents*, 228-235, 1999.

[3] K. Ben Lamine and F. Kabanza. Reasoning about robot actions: a model-checking approach. In *Advances in Plan-Based Control of Robotic Agents*. LNAI 2466, pages 123-139, 2002.

[4] K. Ben Lamine and F. Kabanza, 'History checking of temporal fuzzy logic formulas for monitoring behavior-based mobile robots'. *Proc. of the 12th IEEE International Conference on Tools with Artificial Intelligence*, 312–319, 2000.

[5] M. Broxvall, L. Karlsson and A. Saffiotti. 'Steps toward detecting and recovering from Perceptual Failures.' *Proc. of the 8th Int. Conf. on Intelligent Autonomous Systems, 2004*.

[6] G. De Giacomo, R. Reiter and M. Soutchanski. 'Execution monitoring of high-Level robot programs.' *Proc. of Principles of Knowledge Representation and Reasoning,* 453-465, 1998.

[7] E. Gat. 'On three-layer architecture.' *Artificial Intelligence and Mobile Robots*, **2**, 1622–1627, 1994.

[8] M. Grabisch. 'Temporal scenario modelling and recognition based on possibilistic logic', *Artificial Intelligence Journal*, **148**(1-2) , 261–289, August 2003.

[9] J. Hertzberg, F. Schönherr, M. Cistelecan and T. Christaller. 'Extracting situation facts from activation value histories in behavior-based robots', *KI-2001:: Advances in Artificial Intelligence, LNAI 2174,* 305–319, 2001.

[10] K. Konolige. Colbert: A language for reactive control in SAPHIRA. In *KI: Advances in Artificial Intelligence*, LNAI, pages 31–52, 1997.

[11] J.C. Latombe. '*Robot Motion Planning*.' Kluwer Academic Press, 1991.

[12] K. Madhava and P. Krishna. ‘Perception and remembrance of the environment during real-time navigation of a mobile robot.’ *Robotics and Autonomous Systems*, 37(1) :25–51, 2001.

[13] Z. Manna and A. Pnueli. ‘The Temporal Logic of Reactive and Concurrent Systems.’ Springer-Verlag, 1991.

[14] F.G. Pin and S.R. Bender. ‘Adding memory processing behaviors to the fuzzy Behaviorist-based navigation of mobile robots.’ In *ISRAM’96 Sixth International Symposium on Robotics and Manufacturing*, 27-30 1996.

[15] E.H. Ruspini, K. Konolige, K. L. Myers and A. Saffiotti. ‘The Saphira architecture: A design for autonomy,’ *Journal of Experimental and Theoretical Artificial Intelligence*, **9**(1):215–235, 1997.

[16] R. Simmons and D. Apfelbaum. ‘A task description language for robot control.’ Proc. of Conference on Intelligent Robotics Systems, 1998.

[17] W. L. Xu. A virtual target approach for resolving the limit cycle problem in navigation of a fuzzy behaviour-based mobile robot. *Robotics and Autonomous Systems*, **30**(4) :315–324, 2000.