TECHNICAL RESEARCH REPORT

The MDLe Engine: a Software Tool for Hybrid Motion Control

by D. Hristu-Varsakelis, P.S. Krishnaprasad, S. Andersson, F. Zhang, P. Sodre, L. D'Anna

CDCSS TR 2000-8
(ISR TR 2000-54)

The Center for Dynamics and Control of Smart Structures (CDCSS) is a joint Harvard University, Boston University, University of Maryland center, supported by the Army Research Office under the ODDR&E MURI97 Program Grant No. DAAG55-97-1-0114 (through Harvard University). This document is a technical report in the CDCSS series originating at the University of Maryland.

Web site http://www.isr.umd.edu/CDCSS/cdcss.html

The MDLe Engine: a Software Tool for Hybrid Motion Control*

D. Hristu-Varsakelis†, P. S. Krishnaprasad‡, S. Andersson‡, F. Zhang‡ , P. Sodre§and L. D'Anna§
{hristu, krishna, sanderss, fuminz, sodre, ldanna}@glue.umd.edu
University of Maryland,
College Park, MD 20742

Abstract

One of the important but often overlooked practical challenges in motion control for robotics and other autonomous machines has to do with the implementation of theoretical tools into software that will allow the system to interact effectively with the physical world. More often than not motion control programs are machine-specific and not reusable, even when the underlying algorithm does not require any changes. The work on Motion Description Languages (MDL) has been an effort to formalize a general-purpose robot programming language that allows one to incorporate both switching logic and differential equations. Extended MDL (MDLe) is a device-independent programming language for hybrid motion control, accommodating hybrid controllers, multi-robot interactions and robot-to-robot communications. The purpose of this paper is to describe the "MDLe engine", a software tool that implements the MDLe language. We have designed a basic compiler/software foundation for writing MDLe code. We provide a brief description of the MDLe syntax, implementation architecture, and functionality. Sample programs are presented together with the results of their execution on a set of physical and simulated mobile robots.

1 Introduction

The significant volume of work produced to date on various aspects of intelligent machines has arguably not yet resulted in a workable, unified framework that effectively integrates features of modern control theory with reactive decision-making. This is due in part to the scope and difficulty of the problem, in part to our incomplete understanding of the interaction between individual dynamics and the environment as well as interaction between machines, and in part to the limited expressive power of current models. Consequently it is not surprising that most, if not all, software tools that allow one to incorporate discrete logic and differential equation-based control laws into a program that will interact with the environment, result in hardware-specific code that is difficult to maintain and not reusable without modifications.

The need for a "standard" language for motion control is becoming urgent as modern control theory is challenged to address hybrid control systems of increasing complexity with embedded and/or distributed components (robots, groups of vehicles, smart structures, and MEMS arrays). One approach to such a language began over a decade ago with the the "Motion Description Language" developed in [3, 4, 5] which provided a formal basis for robot programming using behaviors and at the same time permitted incorporation of kinematic and dynamic models of robots in the form of differential equations. The work in [13, 11, 12] (upon which this paper builds) extended the early ideas to a version of the language known as "extended MDL" or MDLe. Language-based descriptions of control tasks use abstractions for simple low-level control primitives and compose such abstractions into programs which - by construction - have at least some chance of being universal. A programming language suitable for motion control should be able to encode hybrid controllers,

*This research was supported in part by a grant from the National Science Foundation Learning and Intelligent Systems Initiative Grant CMS9720334, by the Army Research Office under the ODDR&E MURI97 Program Grant No. DAAG55-97-1-0114 to the Center for Dynamics and Control of Smart Structures (through Harvard University), and by the Office of Naval Research under the ODDR&E MURI97 Program Grant No. N000149710501EE to the Center for Auditory and Acoustics Research.

†Mechanical Engineering
‡Electrical and Computer Engineering and Institute for Systems Research
§Computer Science

allowing for "classical" differential equation-type control interrupted by reactive decision-making. In order to manage complexity and allow one to write reusable programs the language should support hierarchical levels of encoding with programs put together from simpler programs, all the way down to hardware-specific functions.

Our purpose here is to describe a set software tools that enables the rapid development of motion control programs which will operate across machines. For other relevant work on layered architectures for motion control see [7], [6], [2], [9], [13] and references therein.

The paper is structured as follows: In the next section we give a brief summary of the MDLe formalism, structure, and syntax. Section 3 discusses some of the implementation details of the MDLe engine, the software which interprets and compiles MDLe programs. Section 4 describes two multi-robot motion control tasks together with the MDLe programs that code those tasks and the results of their execution.

2 The MDLe language

In the following we give a brief outline of MDLe's syntactic structure and features. For a more complete description see [4, 13]. We have in mind that there is an underlying physical system (this paper will consider robots as an example) with a set of sensors and actuators for which we want to specify a motion control program. At the lowest level is the so-called *kinetic state machine* (see Fig. 1) [13], a biologically-motivated abstraction [1] between atoms (the simplest elements of MDLe) and continuous-time control. A kinetic state machine is governed by a differential equation of the form

$$\dot{x} = f(x) + G(x)u; \qquad y = h(x) \in \mathbb{R}^p \tag{1}$$

where $x(\cdot) : \mathbb{R}^+ \to \mathbb{R}^n$, $u(\cdot) : \mathbb{R}^+ \times \mathbb{R}^n \to \mathbb{R}^m$ and G is a matrix whose columns g_i are vector fields in $\mathbb{R}^n$.

Each MDLe **atom** is an evanescent field defined on space-time. Here "space" refers to the state-space or output space of a dynamical system. The lifetime of an evanescent field is at most $T > 0$ and will be reduced by an interrupt. More specifically, an atom is a triple of the form $\sigma = (U, \xi, T)$, where U is as defined earlier, $\xi : \mathbb{R}^k \to \{0, 1\}$ is a boolean interrupt function defined on the space of outputs from k sensors, and $T \in \mathbb{R}^+$ denotes the value of time (measured from the time an atom is activated) at which the atom will "time out". To evaluate the atom σ means to apply the input u to the kinetic state machine until the interrupt function ξ is "low" (0) or until T units of time elapse, whichever occurs first. We note that T is allowed to be ∞. The input u could be an open loop command or could be given by a feedback law of the type $u = u(t, x)$.

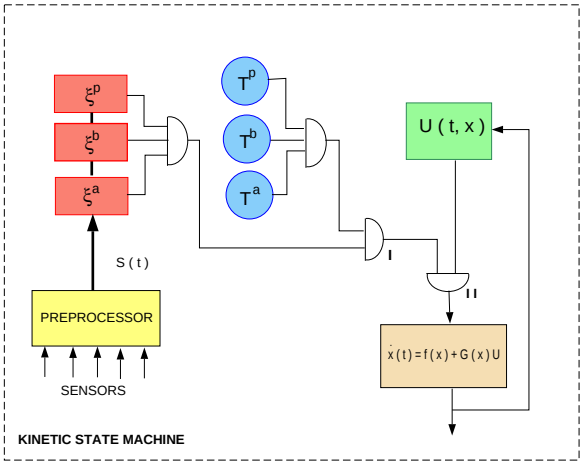


Figure 1: The kinetic state machine (from [13])

A set of atoms can be composed into a string with its own interrupt function and timer. Such strings are called **behaviors**. For example, one could use the atoms $\sigma_1 = (u_1, \xi_1, T_1)$, $\sigma_2 = (u_2, \xi_2, T_2)$ to define the behavior $b = ((\sigma_1, \sigma_2), \xi_b, T_b)$. Evaluating b means evaluating σ_1 followed by σ_2 while the interrupt function ξ_b is "high" and less than T_b units of time have elapsed. Behaviors themselves can be used to form

higher-level structures (partial plans) which in turn can be nested into plans, etc. An example of a plan made from the behavior b and a new atom, might be: $plan_1 = ((b, (u_3, \xi_3, T_3)), \xi_p, T_p)$. Using this LISP-like syntax, MDLe allows for arbitrarily many levels of nested atoms, behaviors, plans, etc. MDLe programs can contain loops, e.g. $b = ((\sigma_1, \sigma_2)^n, \xi_b, T_b)$ denotes the execution of the string (σ_1, σ_2) n times. Though MDLe strings are written in a sequential manner the order of execution of atoms in a program does not have to coincide with their order of appearance in that program. This is a consequence of allowing for interrupts (triggered by external or internal events) as well as loops and it gives MDLe significant expressive power. In particular any hybrid automaton can be easily represented by an MDLe program with atoms encoding "cells" where the system evolves according to a differential equation and with the interrupt functions derived from the automaton's transition rules. A more complete description of MDLe's features will be included in the final version of this paper.

Of course, we are interested in programs that will run on physical hardware. The interface between the kinetic state machine and the hardware is of necessity hardware-specific. This interface, called the "virtual robot" (described in the next section), is charged with translating individual atoms into hardware-specific machine code. The virtual robot encapsulates all hardware-specific functions (e.g. low-level code that interfaces to sensors and actuators) and should be thought of as a device driver for motion control.

3 Reusable motion control programs: The MDLe Engine

One of the driving ideas behind MDLe is the construction of a framework for autonomous robot control and motion planning that separates the user from the low level implementation details of a specific robot. With this in mind we have developed the MDLe engine, a software package implementing an MDLe interpreter, scheduler, and compiler written in C/C++ and currently running under Linux. Figure 2-a shows the block diagram. Programming in MDLe consists of a sequence of four steps:

- writing MDLe code (not unlike the examples in the previous section).
- translating the code to C/C++ and providing "hooks" for hardware-specific device drivers that will execute each atom.
- Creating a number of threads for sensing, actuation and computation and scheduling the execution of those threads in the CPU.
- Compiling the resulting C/C++ program into executable code.

In the following we give brief descriptions of each of these components.

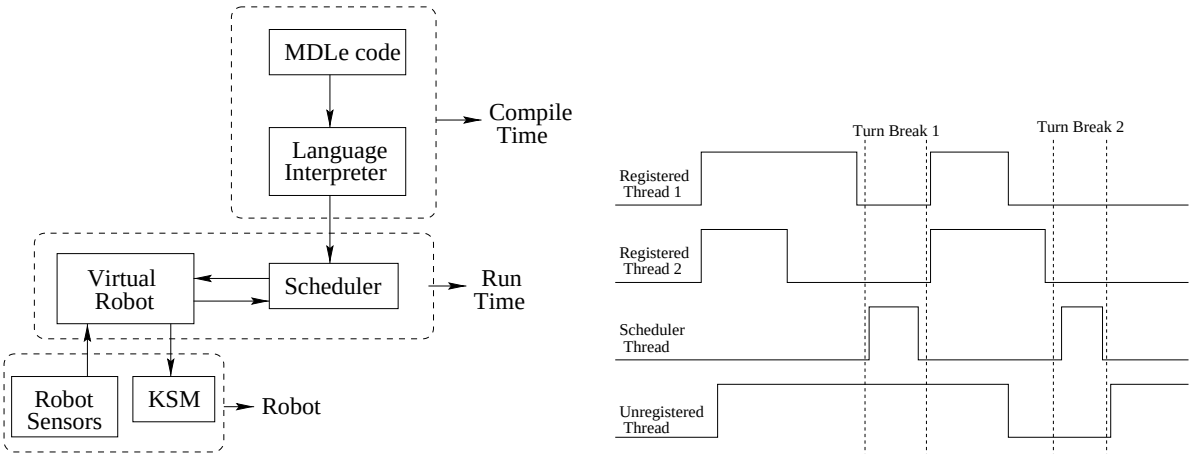


Figure 2: (a) Block diagram of the MDLe engine and (b) Timing diagram in multi-threaded mode

3.1 The MDLe interpreter

Under MDLe plans are built from a dictionary of behaviors and behaviors from an alphabet of atoms. A C/C++ template is provided for the user to develop new interrupts, atoms, and behaviors to add to the dictionary. Plans are specified in a text file using the MDLe language. MDLe code is Lisp-like and details the elements from the dictionary that comprise the behaviors, partial plans, and plans the user would like the robot to follow. See section 4.1.1 for sample MDLe code. The language interpreter reads the MDLe code and generates a corresponding C/C++ file. This code is compiled and linked with supporting libraries into the final executable file.

3.2 The virtual robot

The virtual robot establishes the device-independent nature of MDLe. The virtual robot translates MDLe control commands and data gathering queries into robot-dependent function calls. This module separates the MDLe engine from the hardware layer of the particular robot being used and allows the user to create plans in an abstract manner. Plans that accomplish the same task can then look the same on different robots; only the virtual robot need change. The virtual robot module parses an MDLe program and for each atom "hooks" the machine-specific functions (control law and interrupt) required to execute it. Of course, this means that one must provide a library of all such low-level functions that the MDLe program will require. The standardized structure of atoms allows us to use a C/C++ "template" so that low-level functions can easily be put in the format required by MDLe. The user need only provide pointers to functions which evaluate the control u, interrupt function and value of timer.

3.3 The scheduler

One of the essential features of the MDLe approach is the use of timers and interrupts to enable real time interaction between the robot and its environment. The task of the scheduler is to ensure that the atoms, behaviors, and plans are executed for their specified time intervals and are interrupted under the specified conditions. It is easy to see that prompt response of the MDLe system to interrupts is essential; it is of little use for the robot to sense an imminent collision if it is not able to modify its behaviors swiftly enough. However we are limited by the real time properties of the hardware and operating systems that make up the robot and our goal has been to make MDLe as real time as possible under the system limitations.

The tasks of the various software components are divided based on the complexity of the calculations involved and the speed of response requirements. These tasks are then scheduled as either registered (high priority) or unregistered (low priority) processes. The user has the freedom to select which atoms, behaviors, and partial plans will have high priority. Typical registered processes include direct wheel control, time critical atoms (such as open loop controls or real time feedback control laws), and data critical actions (such as obstacle avoidance). Unregistered processes normally involve high level complex processing of data and include for example, sound localization algorithms, map making algorithms, and target tracking.

3.3.1 Registered processes

The scheduler explicitly controls the timing of the registered processes. It initiates a time slice, referred to as a *turn*, and passes CPU control to the threads associated with the registered processes. Each such process completes its task and then notifies the scheduler.

The scheduler is also responsible for sharing data between all processes, registered and unregistered. Once all registered processes have completed their tasks the scheduler ends the turn and begins the *turn break*. During the turn break the scheduler calls a callback function for each registered process and for each unregistered process that has requested it (see section 3.3.2 below). These callback functions broadcast data to the other processes. In this scheme, then, data is only transferred between processes during the turn breaks. At all other times the tasks run independently.

As the callback functions do nothing more than write a data variable, the turn breaks take a negligible amount of time with respect to the run time of the registered processes tasks. The length of a turn is then determined by the execution time of the registered threads. To ensure near real time operation only tasks that can be run in fast sequential steps should be put into a registered process.

The scheduler can be run either in single- or in multi-threaded mode. In the single thread approach each registered process is run in sequence and the length of the turn is given by the sum of the time for each registered process to complete. In the multi-threaded approach each registered process runs as a separate thread and the length of the turn is given by the maximum time among the registered threads (assuming the overhead imposed by the operating system to achieve multi-threading is negligible with respect to the threads themselves). While the multi-threaded approach adds complexity to the implementation it is necessary to achieve the best performance.

The decision not to use fixed length time slices was driven in large part by the hardware we operate on. Since the robot control system has very different time scales for I/O, sensing, and control the synchronization issues of a fixed length policy can become quite complex. While our method cleanly handles these different times scales it does have an obvious disadvantage; it is at best pseudo-real time. Turn lengths are highly dependent on the particular registered processes. If there are too many critical processes or if any of them require extensive execution time the turn lengths will become unacceptably long. The user is responsible for keeping the registered threads as few and as computationally light as possible. As long as the design of these processes is done properly the control loop times in critical atoms can be guaranteed despite the pseudo-real time nature of our approach.

3.3.2 Unregistered processes

The unregistered processes run independently of the scheduler and therefore do not affect the timing of the critical tasks. Since these processes are independent of the scheduler, however, they cannot simply publish their data to the registered threads because that can only be done during a turn break and the unregistered processes do not know when that event occurs. This problem is handled by providing each unregistered process with an update function that registers a callback function with the scheduler and then blocks the process until the next turn break. At the next turn break the callback function is run by the scheduler, the update function publishes its data to the other processes, the callback function is removed from the scheduler, and the process is unblocked. Similarly, since processes can only read each other's data at turn breaks unregistered processes need a special procedure to get outside information. This is done through an auxiliary callback function that is run by the scheduler at each turn break.

Figure 2-b illustrates the scheduling scheme in the multi-threaded mode. During the first turn multiple registered processes are run simultaneously and the scheduler thread sits dormant. After the longest registered thread has completed its task the first turn break begins. As described above, during the turn break the scheduler runs the callback functions of the registered processes and any unregistered processes that have requested it. When the scheduler has completed its tasks the next turn begins. The diagram clearly shows the dependence of the turn length on the run time of the registered threads. The unregistered threads run independent of the turns unless they want to share data with the other processes. In the diagram the unregistered thread blocks itself at some point after the first turn break and waits until the scheduler has published all the data before becoming active again. The additional delays in the diagram represent operating system overhead for handling the thread switching.

4 Experiments

This section describes a pair of experiments involving a set of coordinated mobile robots. We briefly discuss the motion control task to be performed, give the MDLe programs that implemented that task and show the results of the programs' execution.

The robots used are differentially driven wheeled vehicles outfitted with an array of sonar and touch sensors. The robots are connected through a wireless Ethernet network. While the actual robot controls are the left and right wheel commands, the MDLe atoms were written in terms of the heading and forward controls to allow more intuitive design. As described in Sec. 3 above the virtual robot layer converted the MDLe commands to wheel controls.

4.1 A multi-robot motion control task: learning minimum length paths

Consider a group of vehicles operating on a remote environment without thee benefit of a map describing the terrain. The group has found a (possibly circuitous) path between two fixed locations via exploration and must now find the minimum length path between these points. In [8, 10] a cooperative algorithm known as "local pursuit" is discussed.

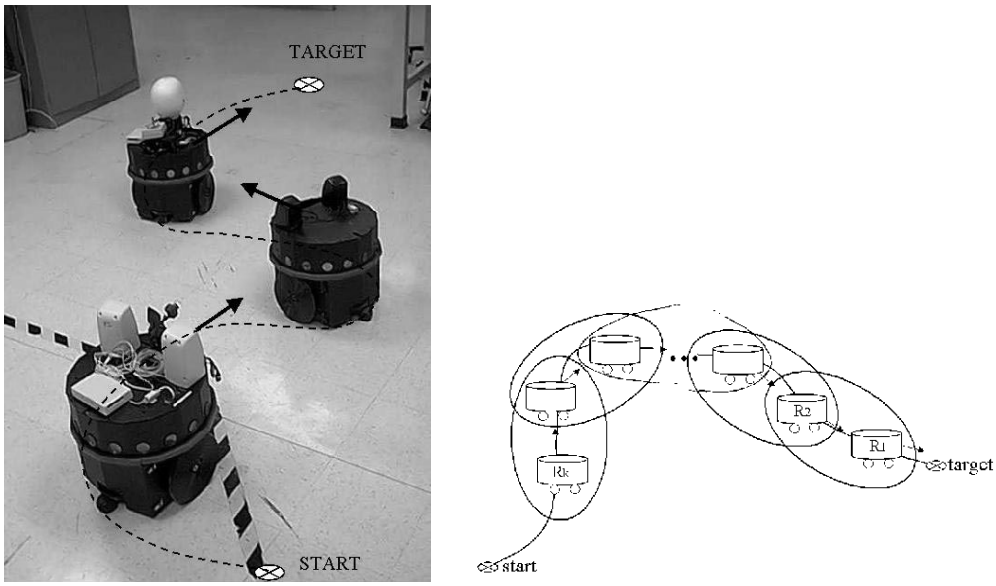


Figure 3: Group of vehicles (a) with Decentralized communication (b)

Local pursuit is an iterative, decentralized algorithm which requires interactions only between neighboring vehicles. Communication needs are thus kept to a minimum. No centralized controller is needed to "guide" the vehicles. According to this strategy a "leader" vehicle departs the starting location and travels towards the target along an initial (suboptimal) path. The next vehicle follows the leader while being followed by another vehicle, etc. It can be proved that the sequence of iterated paths taken by the vehicles converges to a path whose length is locally minimum.

In the following we describe this algorithm, discuss how it was implemented under MDLe, and present the results of our experiment.

We implemented the local pursuit algorithm on a trio of mobile robots, using MDLe to describe the collective task. A coordinate frame was fixed in the room where the robots were located. Starting at the origin one of the robots (designated as the leader) followed a pre-determined path to the goal at $(3.75m, 0.75m)$. The second robot followed the leader using the local pursuit algorithm. The third robot followed the second. Each robot was separated $0.5m$ from the next. When the robots reached the end they moved off the goal point to allow the next robot to complete its path. Once all robots had completed their routes they reversed their order, with robot three taking on the lead role, retracing its path to the origin and robots two and one following in local pursuit. We chose to measure length on the plane using the usual Euclidean metric so that the shortest path was simply a straight line.

4.1.1 An MDLe program for local pursuit

To implement the local pursuit algorithm under the MDLe framework several atoms and behaviors were designed and a module to share state information among the robots was created. In the following the robots are designated R_i, $i = 0, 1, 2, 3, 4$, where R_0, R_4 indicate NULL connections, $u = (u_f, u_\theta)'$ indicates the vector of forward and heading controls, *bumpHit* is an interrupt that fires when any bumper on the robot is depressed, p, p_0 indicate the lab frame coordinates at the current time and at the start of the atom, behavior, or plan, and θ, θ_0 indicate the heading at the current time and at the start of the atom, behavior, or plan.

The following is a list of atoms, behaviors and plans that implemented the pursuit algorithm. The actual MDLe code is syntactically similar to the format used below except that interrupt and timer values are used as arguments with each atom/behavior/plan name (as in the example of Sec.2). Here we choose to present them in tabular form; the final copy will include a listing of the actual MDLe code. We are interested in the device-independent language so the implementation of hardware-specific routines that implement sensing or control operations is not discussed here.

ATOM	Control law	Interrupt	Timer	Comments
antFoll(R_i)	$u = k\frac{\partial g}{\partial s}$ k: constant gain	bumpHit	∞	g: geodesic to R_i s: arclength
path(file)	$u = u(t)$	bumpHit	∞	u(t): read from *file*
pr(d,v,T)	$u_f = v\ u_\theta = 0$	$x = x_0 + d$	T sec	d: meters v: m/s
rotate(ψ)	$u_f = 0\ u_\theta = k(\theta - \psi)$	$\theta = \theta_0 + \psi$	∞	k: constant gain
stop	$u_r, u_l = 0$	-	∞	-
sync(to,T)	$u_r, u_l = 0$	link to *to* ready	T sec	-

BEHAVIOR	Atom List	Interrupt	Timer
trnarnd	rotate(180^0)	bumpHit	10 s
mvOffEnd(i)	stop;sync(R_{i-1},5);pr(0.5,0.25,2);sync(R_{i+1},5)	bumpHit	∞
mv2End(i)	sync(R_{i+1},5);sync(R_{i-1},5);pr(0.5,0.25,2)	bumpHit	∞
mvOffStrt(i)	stop;sync(R_{i+1},5);pr(0.5,0.25,2);sync(R_{i-1},5)	bumpHit	∞
mv2Strt(i)	sync(R_{i-1},5);sync(R_{i+1},5);pr(0.5,0.25,2)	bumpHit	∞

PARTIAL PLAN	Behavior and atom list	Interrupt	Timer
firstPath	sync(R_2,5);path(firstRte);mvOffEnd(1)	bumpHit	∞
lead2Strt	sync(R_2,5);path(lastPath);mvOffStart(3)	bumpHit	∞
lead2End	sync(R_2,5);path(lastPath);mvOffEnd(1)	bumpHit	∞
fol2End(i)	mv2Strt(i);sync(R_{i+1},5);antFoll(R_{i-1}) mvOffEnd(i);sync(R_{i-1},5)	bumpHit	∞
fol2Strt(i)	mv2End(i);sync(R_{i-1},5);antFoll(R_{i+1})) mvOffStrt(i);sync(R_{i+1},5)	bumpHit	∞

PLAN	Partial plan list	Interrupt	Timer
firstRob	firstPath;(trnarnd;fol2Strt(1);trnarnd;lead2End(1))$^{n-1}$	bumpHit	∞
middleRob	(fol2End(2);trnarnd;fol2Strt(2);trnarnd)^n	bumpHit	∞
lastRob	(fol2End(3);trnarnd;lead2Strt(3);trnarnd)^n	bumpHit	∞

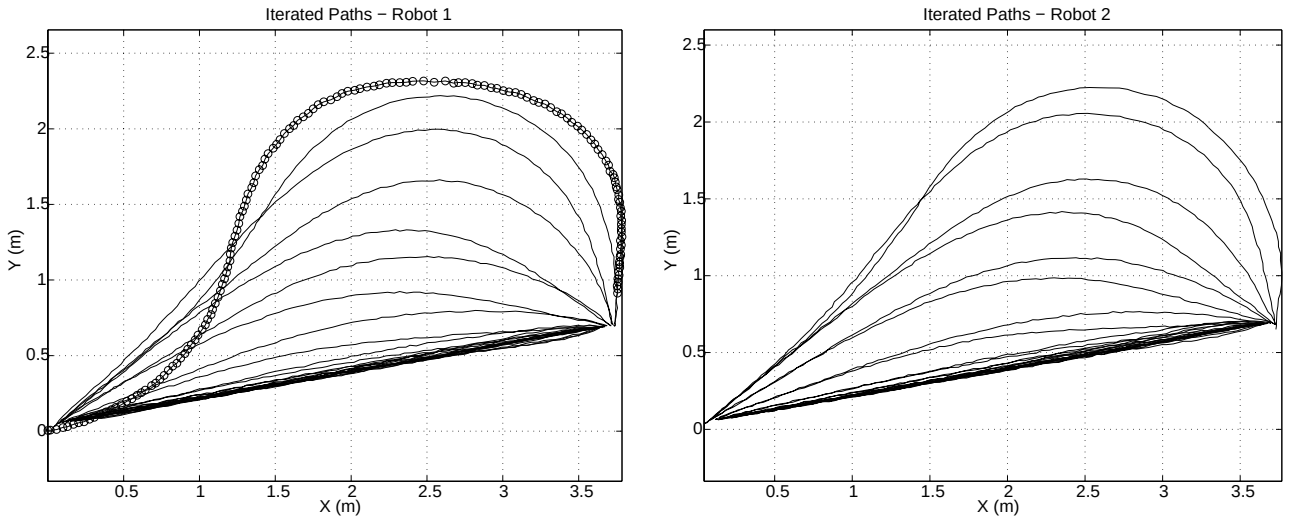
where n is the number of times the robots should shuttle between the end points.

4.1.2 Results

As predicted, each follower traveled less distance than its leader, effectively shortening the path between the origin and the target. Once at the target location, the robots followed each other back to the origin, further reducing the total length traveled. Figure 4 shows the paths traveled by the first (leader) and second robots during seven successive trips between the origin and the target. The curve highlighted with small circles indicates the initial path. As expected, the iterated paths approached a straight line.

4.2 Motion planning with limited sensing: sharing sensors

Consider a group of vehicles moving in a cluttered environment. Each vehicle is equipped with a set of sensors (in our case sonar) with which it can gather data about its environment. Because activating a vehicle's sonar can lead to erroneous measurements being made by nearby vehicles, only one vehicle may activate its sensors at any one time. We designed a decentralized, greedy policy which will allow the vehicles to accomplish their assigned tasks in a safe and efficient manner. According to our policy, a *token* is passed among the robots, so that only the robot currently in possession of the token is allowed to turn on its sensors. Each robot calculates a complexity value which reflects the density of objects near the robot, the rate at which



(a) (b)

Figure 4: Iterated paths created by "following": (a) first and (b) second robots.

the environment is changing, and the length of time since it last had the token, and broadcasts this value over a wireless network. The robot with the largest complexity value wins the token.

4.2.1 Experiment description

We performed a simulation to investigate the feasibility and performance of the greedy policy by giving three simulated robots the task of moving to a goal point through a cluttered and unknown environment. Each robot ran in an identical but separate workspace. Sensor data was used to update a picture of the local environment and the robot controlled itself to the point in the unoccupied region of the sensed area closest to the final goal. While the sensors were disabled the robot continued to move towards the last known intermediate goal. If the goal was reached before the robot got the token it paused until more data could be gathered. Simple proportional feedback on both the heading angle and the position was used on each robot to control to the goals.

4.2.2 An MDLe program for sonar-sharing

To implement this experiment a module to handle the token was created and a single new atom was designed to implement the robot controller. Some atoms designed in the local pursuit experiment were reused.

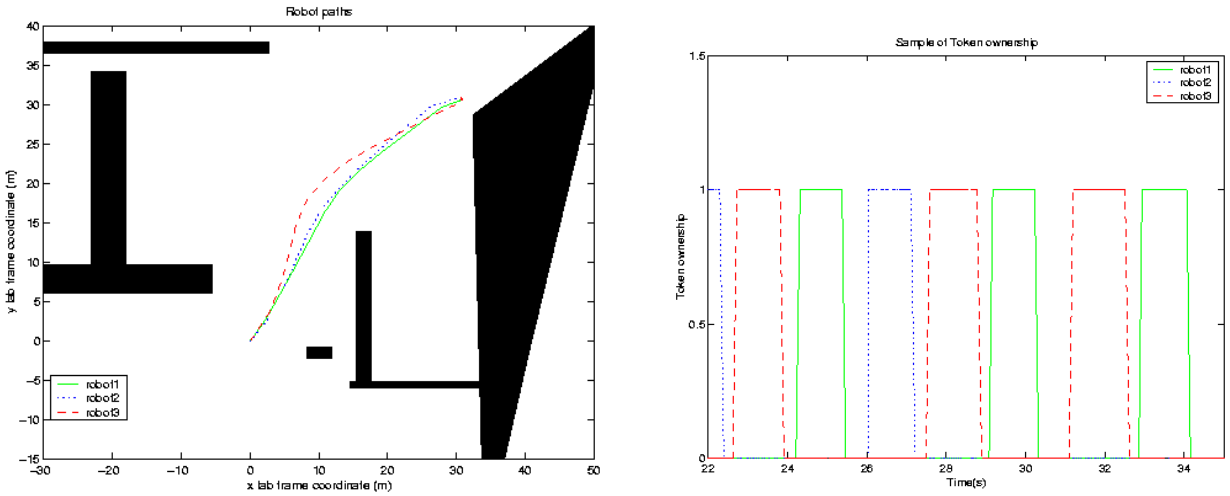
ATOM	Control law	Interrupt	Timer	Comments
navigate(p_g)	$u_f = k\|p_g - p\|^2$ $u_\theta = k\alpha$	bumpHit	∞	k: constant gain α: angle to p_g

BEHAVIOR	Atom list	Interrupt	Timer
bnav(i)	sync(R_{i+1},5);sync(R_{i-1},5);navigate(goal)	bumpHit	∞

PLAN	Behavior list	Interrupt	Timer
robotNav1	bnav(1)	bumpHit	∞
robotNav2	bnav(2)	bumpHit	∞
robotNav3	bnav(3)	bumpHit	∞

4.2.3 Results

As expected all three robots were able to navigate to the goal position without colliding with any of the obstacles. Figure 5a shows the environment the robots moved in and the paths each of them followed. Notice



(a) (b)

Figure 5: (a) Robot paths (c) Token passing

that the paths are not exactly the same. Since the robots are still allowed to move when they do not have the token the actual data for each is different and thus the intermediate goal points are not identical. Figure 5b gives a sample of the token passing during the run. The presence of "dead times" is due in part to network latencies and in part to delays in individual robot processes from picking up the token because of the computational complexity of the simulation.

5 Conclusions

This paper reported on the development of an MDLe compiler which allows one to implement "device independent" motion control programs on physical hybrid systems. Our *MDLe engine* is a software tool that allows the user to write programs which accommodate switching logic, differential equations, hierarchically structured programs, and interaction with hardware. MDLe programs are composed using LISP-like syntax, from a dictionary of "atoms". Atoms are themselves composed from a control law, an interrupt function, and a timer where the last two give the termination conditions for the control law. The MDLe engine translates a program into C/C++ and compiles it to hardware-specific executable code. Hardware-specific routines for actuation and sensing are linked at compile time so that MDLe programs are in principle portable. Of course the performance of an MDLe program depends on the hardware capabilities of the machine which is attempting to execute that program. In general, atoms will require interaction with some (or perhaps all) available sensors and actuators. Depending on the complexity of the data and the control law, varying amounts of pre-processing might be required (for example, an on-board camera supplies an image out of which a few bits of information must be extracted). For this reason we have provided for spawning several threads within an atom in order to meet that atom's communication and computational needs. Scheduling of those threads is turn-based and time-critical control/sensing code fragments may be scheduled at a higher priority. Clearly, the ability of an atom to respond quickly to an interrupt will depend on the computational complexity of that atom.

We described two motion control programs involving hybrid control systems (autonomous mobile robots) along with the MDLe programs that implement them and showed the results of their execution. Future work is directed towards the development of a real-time version of MDLe under RTLinux as well as a set of graphical programming and verification tools. We believe that the flexibility and device independence of MDLe make it a useful tool for hybrid control and we plan to make our software available shortly via the World Wide Web.

References

[1] N. Bernstein. *The Coordination and Regulation of Movement*. Pergamon Press, 1967.

[2] B. M. Blumberg and T. A. Galyean. Multi-level direction of autonomous cratures for real-time virtual environments. In *SIGGRAPH Proceedings*, pages 47–54, 1995.

[3] R. W. Brockett. On the computer control of movement. In *Proceedings of the 1988 IEEE Conference on Robotics and Automation*, pages 534–540, April 1988.

[4] R. W. Brockett. Formal languages for motion description and map making. In *Robotics*, pages 181–93. American Mathematical Society, 1990.

[5] R. W. Brockett. Hybrid models for motion control. In H. Trentelman and J. C. Willems, editors, *Perspectives in Control*, pages 29–51. Birkhauser - Verlag, 1993.

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

[7] R. A. Brooks. Intelligence without reason. Technical Report A.I. Memo No. 1293, MIT, 1991.

[8] A. M. Bruckstein. Why the ant trails look so straight and nice. *The Mathematical Intelligencer*, 15(2):59–62, 1993.

[9] D. Hristu. *Optimal Control with Limited Communication*. PhD thesis, Harvard University, Div. of Engineering and Applied Sciences, 1999.

[10] D. Hristu. Robot formations: Optimizing path length on uneven terrain. In *IEEE Mediterranean Conference on Control and Automation*, 2000.

[11] V. Manikonda, J. Hendler, and P. S. Krishnaprasad. Formalizing behavior-based planning for nonholonomic robots. In *Proceedings 1995 International Joint Conference on Artificial Intelligence*, volume 1, pages 142–9, August 1995.

[12] V. Manikonda, P. S. Krishnaprasad, and J. Hendler. A motion description language and a hybrid architecture for motion plannning with nonholonomic robots. In *Proceedings 1995 IEEE International Conference on Robotics and Automation*, volume 2, pages 2021–8, May 1995.

[13] V. Manikonda, P. S. Krishnaprasad, and J. Hendler. Languages, behaviors, hybrid architectures and motion control. In J.C. Willems J. Baillieul, editor, *Mathematical Control Theory*, pages 199–226. Springer, 1998.