A Motion Description Language for Hybrid System Programming

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

Abstract— **One of the important but often overlooked challenges in motion control has to do with the transfer of theoretical tools into software that will allow an autonomous system to interact effectively with the physical world. In a situation familiar to most control practitioners, motion control programs are often machine-specific and are not reusable, even when the underlying algorithm does not require changes. These considerations point out the need for a formal, general-purpose programming language that would allow one to write motion control programs, incorporating both switching logic and differential equations. The promise held by such a software tool has motivated a research program on the so-called Motion Description Language (MDL) and its extended version MDLe, put forth as device-independent programming languages that can accommodate hybrid controllers, multi-system interactions and agent-to-agent communications. This paper details the syntax, functionality and expressive power of MDLe as well as a software infrastructure that implements the language. We include a set of programming examples that demonstrate the capabilities of MDLe, together with the results of their execution on a group of mobile robots.**

I. 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 may be attributed to: i) the scope and difficulty of the problem, ii) our incomplete understanding of the interaction between individual dynamics and the environment as well as machine-to-machine interactions, and iii) the limited expressive power of current models. Consequently it is not surprising that most control programs (viewed broadly as pieces of computer code that cause a machine to perform a desired task) are machine dependent, highly specialized and not reusable. In fact, we argue that there is a lack of control-oriented software tools that allow one to combine discrete logic and differential equation-based control laws into robust, reusable programs that interact with the environment. We believe that 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). The work presented in this paper is part of a larger effort towards a general-purpose high-level programming language that: i) has as its intended physical layer a control system as opposed to a CPU, and ii) provides a level of abstraction that appears to be critical in managing the specification complexity of motion control programs (see for example [1]).

This research was supported by ODDR&E MURI01 Grant No. DAAD19-01-1-0465, (Center for Communicating Networked Control Systems, through Boston University), by NSF CRCD Grant No. EIA 0088081 and by AFOSR Grant No. F496200110415

*Department of Mechanical Engineering and Institute for Systems Research

† Department of Electrical and Computer Engineering and Institute for Systems Research

‡ Department of Computer Science

§ Corresponding author: 2233 A. V. Williams Bldg., University of Maryland, College Park, MD 20742. Tel: (301) 405-6843, Fax: (301) 314-9920.

A. The Challenges of Control Software Organization

The viewpoint taken in this paper considers control software as a combination of:

- i) a collection of control laws (together with rules for switching between them) which if successfully implemented will cause a dynamical system to perform a desired task, and
- ii) auxiliary software that translates the inputs generated by a control law into hardware-specific commands (i.e. voltages, byte packets, etc). This software can be thought of as being comparable to the device drivers that desktop computers use to interface with their peripherals.

This separation between hardware and software by means of a layer of device drivers is a first step in object oriented programming (OOP), where complex programs are built hierarchically from simpler ones, with a small set of dedicated routines for interfacing with the hardware at the lowest level . Although this model has intuitive appeal and often results in programs with reduced complexity, it has not yet been successful in the area of control systems, at least when compared to general purpose software. The following situation illustrates our point: Suppose that an engineer has designed a set of control laws and switching logic that will allow, say, a wheeled robot to accomplish some task that requires interaction with its environment (e.g. vision-based navigation). The control laws are translated into software, taking into account the kinematics of the robot, the available sensor/actuator suite and available driver software for accessing them. The engineer may even make use of rapid prototyping graphical environments such as DSpace or Simulink to generate the source code which will then be compiled and executed on the robot to produce the desired results. At the end of the software design process, the resulting source code is not “portable”. To be successfully executed on an identical robot with a different image acquisition system, it is likely that the source code will have to be modified to use a different set of hardware drivers for interfacing with its camera, but not much else may need to be changed. But how about running the same code on a robot with different size, inertia or even kinematics? In those cases, it would not be unreasonable to expect that a major re-write of the software would be necessary in order to achieve the intended operation. In fact, changes may be required in the order, number and functional forms of the control laws required to accomplish the same task as before. Moreover, after making these changes, one would likely end up with just as specialized a software as before.

The situation described above is clearly unsatisfactory, yet it is more often the rule rather than the exception. Moreover, there are numerous software design applications in which such problems of portability and reusability have been alleviated by structuring the code according to the dictum of OOP. Some of the larger (in terms of lines of code) and most sophisticated applications include scientific and business software, games, as well as some that interact with the physical world, with printer control drivers/programs being a notable example. There is, of course, a substantial amount of literature on software organization, with no shortage of works emphasizing the power and conceptual advantages of thinking in terms of objects, classes, and reusable modules. We refer the reader to standard references on that subject (e.g., [2]). However, when it comes to the control of electromechanical systems, standard object-oriented methods will take on a somewhat special form because the software must interact with one-of-a-kind hardware and instruction sets. In addition, there are few precedents that could dictate how control software should be organized.

In OOP, portability and conceptual simplicity are obtained by *encapsulating* or *tokenizing* low-level commands and data structures into objects which may then be viewed as new commands and used to build more complex objects. This strategy is particularly effective in managing the complexity of large programs and limits the number of lines of code that must be altered when something changes in the lower levels of the application (e.g. the code is run on a different machine, a different algorithm is used to implement one of the low-level objects, a different printer will be used, etc). Despite its advantages, OOP - or something analogous to it - has not substantially entered into motion control software and most software that drives control systems tends to take the form of difficult-to-maintain, hardware-dependent one-off’s. At first glance, one may be tempted to attribute this to the lack of uniformity in the hardware used to construct real-world control systems. The variety of devices encountered in experimental control systems seems vast when compared with the relatively standardized desktop computing platform. Put differently, those who write desktop PC software have had the benefit of a short list of devices (with universally-defined interfaces) for for which low-level drivers have been written. Nevertheless, this cannot wholly explain the difficulty in writing “universal” control programs: the availability of device drivers for a multitude of actuator and sensor suites is almost irrelevant when the plant dynamics change. What is apparently required is not more low-level software objects, but rather a way to combine the available objects into motion control programs that are interpretable or compilable to the specifics of a machine and that support reconfigurability and replanning when the hardware changes.

We see then that when it comes to programming electro-mechanical systems, there are *not one but two levels of abstraction that must be dealt with*: one has the role of isolating software objects from hardware (by means of appropriate device drivers); the other - equally important - has to do with separating the control laws from the particulars of a system’s kinematics and dynamics. Efforts to facilitate this inclusion of dynamics and kinematics as part of the hardware (such as the MDLe research program) are still at their early stages, however they have the potential for significant impact, perhaps comparable to the leap that was accomplished when programming moved from assembly code to high level languages.

B. Language-based Controllers

The idea behind language-based descriptions of control tasks is to use abstractions for simple low-level control primitives and compose those 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, 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. One approach to such a language began over a decade ago with the “Motion Description Language” (MDL) developed in [3], [4], [5] which provided a formal basis for robot programming using “behaviors” (structured collections of control primitives) and at the same time permitted the incorporation of kinematic and dynamic models of robots in the form of differential equations. The work in [6], [7], [8] (upon which this paper builds) extended the early ideas to a version of the language known as “extended MDL” or MDLe. For other relevant work on layered architectures for motion control see [9], [10], [11], [12], [8], [3],[4], [13] and references therein. An effective control language environment should reflect the complexity of purpose and the possibility of multimodal operation that one expects to find in complex systems. Our goal in this paper is to describe for others the structure of one such environment and to present a sample of the experiments that were facilitated by it. We provide a definition of MDLe as a formal language, discuss its expressive power and describe a set software tools that enable the rapid development of motion control programs which will operate across machines.

The remainder of this paper is structured as follows: In the next section we give a summary of the MDLe formalism and syntax. Section III discusses the implementation and features of a programming environment that enables the execution of

MDLe programs. Section IV describes a collection of multi-robot motion control tasks together with their MDLe code and the results of its execution.

II. The MDLe language specification

This section outlines the syntax and features of MDLe as described in earlier work (see [4], [8]). Here for the first time we distill the early descriptions into a formal language definition and explore the expressive power of the syntax. 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. The physical system is modeled by a so-called *kinetic state machine* (see Fig. 1), [8], which can be viewed as a biologically-motivated abstraction [14] separating the simplest elements of a control language (to be defined) and continuous-time control. A kinetic state machine (KSM) 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 $x(\cdot) : \mathbb{R}^+ \to \mathbb{R}^n$, $u(\cdot) : \mathbb{R}^p \times \mathbb{R}^+ \to \mathbb{R}^m$ may be an open loop command or feedback law of the type $u = u(t, h(x))$, and G is a matrix whose columns g_i are vector fields in $\mathbb{R}^n$.

The simplest element of MDLe is the **atom**, an evanescent vector field defined on space-time. Here "space" refers to the state-space or output space of a dynamical system. The lifetime of an atom is at most $T > 0$ and may be reduced by an interrupt. More precisely, an atom is a triple of the form $\sigma = (u, \xi, T)$, where u is as defined earlier, $\xi : \mathbb{R}^p \to \{0, 1\}$ is a boolean **interrupt** function defined on the space of outputs from p sensors, and $T \in \mathbb{R}^+$ denotes the value of time (measured from the time an atom is initiated) at which the atom will "time out". To *evaluate* or *run* the atom $\sigma = (u, \xi, T)$ means to apply the input u to the kinetic state machine until the interrupt function ξ is "low" (logical `0`) or until T units of time elapse, whichever occurs first. T is allowed to be ∞.

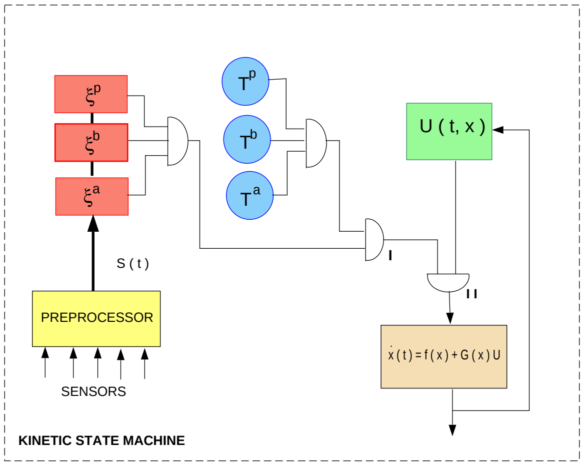


Fig. 1. The kinetic state machine (from [8])

Atoms can be composed into a string that carries 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 until the interrupt function ξ_b returns "low" (logical `0`), or T_b units of time have elapsed, or σ_2 has terminated. Behaviors themselves can be composed to form higher-level strings (named **partial plans**) which in turn can be nested into **plans**[1], etc. An example of a partial plan made from the behavior b and a new atom, might be: $plan_1 = ((b, (u_3, \xi_3, T_3)), \xi_p, T_p)$. MDLe programs can contain loops which are denoted by exponents. For example, $b = ((\sigma_1, \sigma_2)^n, \xi_b, T_b)$ denotes the execution of the string (σ_1, σ_2), n times.

A. MDLe as a formal language

Formally, we let $\mathcal{U} = \{u : \mathbb{R}^p \times \mathbb{R}^+ \to \mathbb{R}^m\}$ be the set of possible control laws (including the trivial $u_{null} = 0$) and $\mathcal{B} = \{\xi : \mathbb{R}^p \to \{0, 1\}\} \cup \{\xi_{null} : \mathbb{R}^p \to 1\}$ the set of boolean functions on p variables (including the null interrupt $\xi_{null} : \mathbb{R}^p \to 1$). Then, define an (finite) alphabet of atoms

Definition 1: $\Sigma \triangleq \{\sigma : \sigma = (u, \xi, T)\} \cup \{\sigma_{null} = (0, 1, \infty)\}, \quad u \in \mathcal{U}, \quad \xi \in \mathcal{B}, \quad T \in \mathbb{R}^+$.
where the symbol σ_{null} is used to denote a special termination atom. Formally, σ_{null} will be the last atom of any MDLe string[2].

A word on notation: for simplicity, we will sometimes "combine" an atom's timer and interrupt function, by re-defining interrupts on $\mathbb{R}^p \times \mathbb{R}^+$, and writing (u, ψ) instead of (u, ξ, T), where $\psi = (\xi \texttt{ AND } (t \leq T))$. Under this notation we will say that an atom is made up of a *control quark* selected from $\mathcal{U}$ and an *interrupt quark* from $\mathcal{B}' = \{\xi : \mathbb{R}^p \times \mathbb{R}^+ \to \{0, 1\}\} \cup \{\xi_{null} : \mathbb{R}^p \times \mathbb{R}^+ \to 1\}$. Of course, this means that we limit ourselves to a finite set for the values of the timer T. This is a necessary step if we are to have a finite alphabet over which we can define a formal language.

Finally,

Definition 2: MDLe is the formal language with valid strings s composed from the alphabet $\mathcal{A} = \mathcal{U} \cup \mathcal{B}' \cup \{(,)\} \cup \{,\}$ using the following rules:

- The set of valid strings V is the union of two classes of strings: Open (O) or Closed (C)
- *Encapsulation* $s \to (s, \xi) \in C$, where $s \in O \cup \mathcal{U}$, $\xi \in \mathcal{B}'$ We refer to s as an *encapsulated* substring, and to ξ as the interrupt *associated* with s.
- *Concatenation* $\{s_1, ..., s_n\} \to s_1 \cdots s_n \in O$, where $s_i \in V$.
- *Looping* $s \to s^n \in O$ if $s \in C$ and $s \to (s)^n \in O$ if $s \in O$, $n \in \mathbb{N}^*_+$.

Using the syntax defined above, MDLe allows for arbitrarily many levels of nested atoms, behaviors, plans, etc. We note that loops (exponentiation in MDLe) shorten the description of an MDLe program but in principle do not contribute to the expressive power of the language. In fact, it will sometimes be convenient to *expand* all loops in a string by replacing each

[1]In the following, we will sometimes use the word "plan" to mean a generic MDLe string independently of the number of nested levels it contains.

[2]We will omit writing σ_{null} when it is convenient to do so without sacrificing clarity

one with an appropriate number of copies of the corresponding substring (e.g. $(s_1s_2)^3 \to s_1s_2s_1s_2s_1s_2$).

Remark: The definition of MDLe given above is based on generative rules. It is also possible to define MDLe as the formal language generated by the context-free grammar [15]:

Definition 3: MDLe is the language generated by the grammar $G := (N, T, P, S)$, where:
$N = \{A, B, C, S\}$ is the set of non-terminal symbols,
S is a starting symbol,
$T = \{\mathcal{A} \cup \{\epsilon\}\}$ are the terminal symbols,
ϵ denotes the null string and
$P \subset N \times (N \cup T)^*$ is a finite relation which consists of the following production rules:

1) $S \to C(A, B)$
2) $C \to C(A, B)$
3) $A \to C(A, B)(A, B)$
4) $A \to a_i,\ a_i \in \mathcal{U}$
5) $B \to \xi_i,\ \xi_i \in \mathcal{B}'$
6) $C \to \epsilon$

It is possible to show that Definitions 3 and 2 are in fact equivalent. Details (as well as a discussion of the language complexity of MDLe) can be found in [16]. For our purposes, Def. 2 will be more convenient especially when it comes to discussing the expressive power of the language.

In previous work [8] it was proposed that the syntactic rules of MDLe define a language over Σ, the alphabet of all atoms (u, ξ, T) with $u \in \mathcal{U}$ and $\xi \in \mathcal{B}$. Our previous discussion suggests that the last statement must be modified: valid MDLe strings are drawn not from Σ^*, but from a richer superset. At execution time however, every valid MDLe string does generate what we might call a "secondary" string $s' \in \Sigma^*$. The string s' is simply the concatenation of atoms that appear in s, in order of their execution. Of course, every MDLe string s can produce many such secondary strings, depending on the order and identity of the interrupts that were triggered during execution.

Definition 4: An MDLe string s is **degenerate** if there is an interrupt ξ associated with an encapsulated substring (s_1, ξ) of s *and* also appears in s_1. The interrupt ξ is said to be **repeated** within s.

In particular, if s is degenerate then there is an ambiguity with respect to which atom should be executed when ξ is triggered. We can resolve this ambiguity by always requiring that the highest-level transition take precedence. For example, if ξ is associated both with an atom and with a behavior b containing that atom (but not with any other proper substring of s that contains b), it is the behavior b that will be terminated when ξ returns `0`. Formally, we convert a degenerate string s to a non-degenerate one as follows:

- identify all valid substrings of s that contain repeated interrupts $(s_1, \xi_1), (s_2, \xi_2), ..., (s_k, \xi_k)$ and are such that no s_i is a substring of s_j for $1 \le i, j \le k$.
- replace all occurrences of ξ_i in s_i with the null interrupt.

We will assume that this conversion is always applied, and thus limit our discussion to non-degenerate strings.

B. The expressive power of the MDLe syntax

By construction, MDLe has a "sequential" syntax. This agrees with our intuition regarding the temporal order of some motion control tasks, but there are certainly alternative ways to express control programs. In particular, one can consider a kinetic state machine whose evolution is controlled not by MDLe strings (as explained in the beginning of Sec. II) but by a *finite state machine* (hereby abbreviated as "FSM") whose states are identified with control quarks, while state-to-state transitions occur in response to interrupts. An example is shown in Fig. 2. A kinetic state machine executes a FSM-

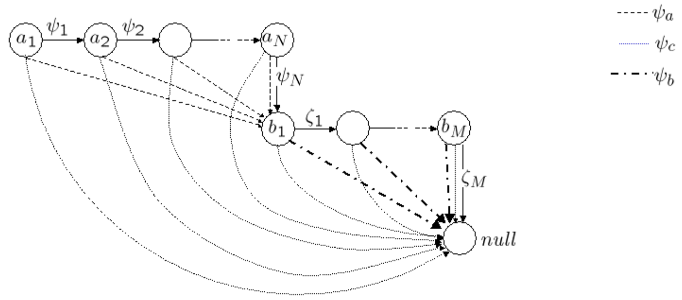


Fig. 2. Finite state machine representation of a two-behavior MDLe plan: $((a, \psi_a)(b, \psi_b), \psi_c)\sigma_{null}$, where $a = ((a_1, \psi_1) \cdots (a_N, \psi_N))$, $b = ((b_1, \zeta_1) \cdots (b_M, \zeta_M))$

like program by running the control law specified by the current state of the FSM until a transition to a new state (corresp. new control law) occurs. The FSM representation of motion control programs may seem more expressive than MDLe strings - after all we have not imposed any syntactical restrictions on transitions. This gives rise to the question of which representation (strings or FSMs) is "richer". To answer, we first require a notion of equivalence between the two representations.

Definition 5: A FSM F with states $c_1, ..., c_n$ is **equivalent** to an MDLe string s with control quarks $a_1, ..., a_n$ if both produce the same trajectories on the same kinetic state machine, starting from the same initial conditions.

Formally, a correspondence between an MDLe string s and an equivalent FSM can be made as follows: Let $\mathbb{R}^n_e = \{e_1, e_2, ..., e_n\}$ denote the set of standard unit vectors in $\mathbb{R}^n$. Identify states of the FSM (resp. control quarks in a string) with vectors in $\mathbb{R}^N_e$ and the FSM (resp. MDLe string) with monomials of the form:

$$\Pi(t) \triangleq \sum_{i=1}^{M} E_i \bar{\xi}_i(t) \qquad (2)$$

where the **transition matrices** $E_i \in \{0, 1\}^{N \times N}$ have columns taken from $\mathbb{R}^n_e$ and $\bar{\xi}(t)$ denotes the complement of ξ. If the control quark (resp. FSM state) running at time t is identified by $e(t) \in \mathbb{R}^N_e$, and ξ_i is trigerred at time t the new atom will be $E_i e(t)$. The evolution $e(t) \to E_i e(t^+)$ could be compared with that of a Markov chain in which transitions always occur with probability 1. A careful examination of the generative rules that define valid MDLe strings (Def. 2) reveals that transition matrices are restricted to only three types (up to renumbering of the atoms):

1) **Atom-level** If ξ_i is an atom-level interrupt associated with the k^{th} atom, then the corresponding E_i is a matrix with all of its diagonal entries except (k,k) set to 1, its $(k+1,k)$ entry set to 1 and all remaining entries being 0.
2) **Behavior-level** If ξ_i is attached to an encapsulated string, E_i will be of the form

$$\begin{array}{r|cccccccccc|}
 & 1 & 0 & \cdots & \cdots & & & & \cdots & 0 & 0 \\
 & 0 & 1 & 0 & & & & & & & 0 \\
 & \vdots & 0 & \ddots & & & & & & & \\
 & & \vdots & & 1 & & & & & & \\
 & & & & & 0 & & & & & \\
E_i = & & & & & \vdots & \ddots & & & & \\
 & & & & & 0 & & 0 & & & \\
(k+1)^{st} \rightarrow & & & & & 1 & \cdots & 1 & 1 & & \\
row & \vdots & \vdots & & & & & & & \ddots & \\
 & 0 & 0 & \cdots & & & & & \cdots & 0 & 1 \\
\multicolumn{5}{c}{} & \multicolumn{4}{c}{\underbrace{\hphantom{0 \quad \cdots \quad 1 \quad 1}}_{(k-l)\dots(k)^{th} columns}} & \multicolumn{2}{c}{}
\end{array}$$

sending states $k-l, k-l+1, ..., k$ to $k+1$ (recall that atoms were numbered sequentially, therefore there will be no gaps in the "partial row" of 1s).
3) **Looping** If ξ_i is the last interrupt in a loop (transitioning from the k^{th} to the $(k-l)^{th}$ atom, then E_i has all of its diagonal entries except (k,k) set to 1, its $(k-l,k)$ entry set to 1, all remaining entries being 0. Then the term $E_i\xi_i$ corresponds to an infinite loop. Finite loops can be handled by associating with Eq. 2 an auxiliary $c(t)$ whose value is initially n (the corresponding exponent in the MDLe string) and decreases by 1 every time ξ (the interrupt triggering a new iteration of the loop) goes from 1 to 0. Then, the term in Eq. 2 corresponding to the loop transition is not $E_i\bar{\xi}_i$ but rather $E_i\bar{\xi}_i\mathbb{I}(c(t)>0)+E_i'\bar{\xi}_i\mathbb{I}(c(t)=0)$, where $\mathbb{I}(b)$ is equal to 1 if the statement b is true and 0 otherwise, and E_i' is a simple atom-level transition. This additional level of detail is not necessary if we agree to always expand out every loop in an MDLe string by replacing the corresponding substring with an appropriate number of copies of itself.

Of course, the same interrupt ξ may appear in several - say n - substrings (atoms, behaviors, etc). If we count ξ only once (instead of each time it appears in a string) the coefficient matrices for each appearance of ξ must be combined accordingly

$$E_i = \sum_{j=1}^{n} E_{ij} - (n-1)I_N, \tag{3}$$

where j indexes the n MDLe sub-strings that are associated with ξ_i and I_N is the $n\times n$ identity matrix. In the following it will be convenient to treat repeated occurrences of an interrupt as distinct, each paired with its own transition matrix E_i, although the discussion is largely unchanged if one decides to use the convention of Eq. 3. Because we always convert strings to their non-degenerate form and because of our assumption that only one interrupt function may change value at any time, we see that the transition (temporally) from any atom to the next will be unambiguous. Said differently,

Observation 1: If an MDLe string s is non-degenerate, all of its associated transition matrices have columns which are standard basis vectors in some $\mathbb{R}^N$.

The monomial representation of an MDLe string by means of Eq. 2 provides a convenient tool for computing the execution trace of that string. Let $e(t)$ be the unit vector whose nonzero row matches the index of the atom being executed at time t. Given the interrupt functions $\xi_i(t)$ on $[0,T]$ and assuming that no two interrupts are triggered at the same instant[3], we can write

$$\begin{aligned} e(t) &= \Pi(t_m)...\Pi(t_2)\Pi(t_1)e(0) \\ &= E_{i(t_m)}...E_{i(t_2)}E_{i(t_1)}e(0), \quad t \geq t_m \end{aligned} \tag{4}$$

where $i(t_k), k=1,...,m$ are the indices of the interrupts that were triggered at $t_1 < t_2 < ... < t_m$ and $e(0)$ is the unit vector corresponding to the index of the first atom (typically $e(0)=[1,0,...,0]^T$). At the same time, the state evolution of the underlying KSM can be expressed as

$$x(t) = \phi(t_m, t, ...\phi_{i(t_2)}(t_1, t_2, \phi_{i(t_1)}(t_0, t_1, x_0)))) \tag{5}$$

where $i(t_k)$ are again the indices of the interrupts that were triggered at $t_1 < t_2 < ... < t_m$ and $\phi_i(t_j, t_k, x_0)$ is the flow of the KSM (Eq. 1) from t_j to t_k, with initial condition x_0 and control u determined by the atom indicated by the nonzero entry of $e(t_j^+)$.

Given any FSM (whose states and transitions are identified with control laws and interrupts) we can ask whether it has an MDLe equivalent. Passing to the monomial representation of the FSM (Eq. 2) helps answer the question and obtain an algorithm for testing for the existence of an equivalent valid MDLe string.

Theorem 1: Given the FSM representation of a motion control program

$$\Pi(t) \triangleq \sum_{i=1}^{M} E_i\bar{\xi}_i(t); \quad E_i \ n\times n$$

there exists a non-degenerate MDLe string equivalent to Π if and only if there exists a renumbering of states in the FSM (renumbering of rows and columns of all E_i) such that the following hold:

- (R1) For $k=1,...,n-1$ there is a unique index $i(k)$ such that $E_{i(k)}$ is the identity matrix with its (k,k) entry set to 0 and its $(k+1,k)$ entry set to 1 (all "atom"-level transitions are present).
- (R2) For all $i=1,...,m$: If the k^{th} column of E_i is the unit vector e_j, then $j \geq k$ and all columns $k, k+1, ..., j$ of E_i must also be e_j.
- (R3) If the k_1^{th} column of E_{i_1} is the unit vector e_{p_1} and there exists among $E_1, ..., E_m$ another E_{i_2}, $i_2 \neq i_1$ with

[3]If one insists on allowing simultaneously occurring interrupts, we will give priority to the highest-level interrupt, i.e. if an atom-level and a behavior-level interrupt both occur at time t, the behavior level interrupt is evaluated first, eliminating the need to evaluate the atom-level interrupt

e_{p_2} in column k_2 where $k_2 \leq k_1$ and $p_2 < p_1$, then E_{i_1} must have e_{p_1} in columns $j, ..., k_2, ..., p_1$ where $j < k_1$.

- (R4) If E_i contains e_j in its k^{th} column with $j > k$ then: i) there must not be any E_l whose $j-p, ..., j-1, j$ columns are equal for $p > 0$ and ii)if there exists an E_l whose columns $q, q+1, ..., r$ are equal and $q \leq j \leq r$, then $j < q$.

Proof:
Clearly, if for any renumbering of the FSM states at least one of R1-R4 are violated, there is no equivalent MDLe string because

- In a valid MDLe string there is always at least one transition from each atom a_i to the next a_{i+1} (again following the left-to-right numbering convention used previously) that occurs when the atom-level interrupt of a_i is triggered.
- If R2 does not hold, then MDLe's rule for encapsulation is violated because there is an interrupt (behavior-level or higher) that causes a transition from atoms $a_k, a_{k+1}, ..., a_l$, $k < l < j-1$ to a_j, effectively "skipping" over $a_{l+1}, ..., a_{j-1}$. Such a composition clearly cannot be expressed either as concatenation or encapsulation.
- If R3 does not hold, then part of a behavior b (or higher-level substring) is encapsulated into a new string but part of b is not, violating MDLe's rule for encapsulation.
- If R4 does not hold, then there is a loop that either does not transition to the first atom of an encapsulated string (i), or transitions outside a string in which the loop should be encapsulated.

For the converse, note that if there is a numbering of atoms such that R1-R4 are satisfied then we can immediately form an MDLe string by first writing down all atoms from left to right and then group them in behaviors, partial plans, etc according to each E_i. $\square$

We now turn to the problem of finding FSM representations of MDLe programs. Given an MDLe string, we can construct an equivalent FSM representation if we allow the FSM to have a sufficiently many states:

Algorithm 1 (FSM equivalent of an MDLe string):

1) Given a non-degenerate MDLe string s, expand any loops that may exist in s and enumerate its control quarks $a_1, ..., a_{N-1}$ sequentially in order of appearance (from left to right) in s, including repeated quarks[4]. To the sequence of control quarks we add the null symbol, labeled a_N.
2) To each a_i, associate a state of an N-state FSM in a one-to-one fashion, including a "termination" state.
3) Enumerate the interrupts $\xi_1, ..., \xi_M$[5] in order of appearance (left to right) in s, including duplications.
4) Associate each ξ_i to a transition from a set of states of the FSM (depending on whether ξ_i is associated with an atom, behavior, etc) to a new state.

[4]This involves running the string s through a parser which spawns an instantiation of each quark and assigns to it a unique identifier.

[5]We are again using the shorthand notation $\sigma = (u, \zeta \texttt{ AND } (t < T))$ for atoms.

By passing to the representation (2) one can check that the execution traces of an MDLe string and its equivalent FSM produced using the above algorithm are identical. The proof is simple and will be ommited.

It should be clear that if we insist that repeated appearances of an atom are identified with the same FSM state, then there are MDLe strings that have no FSM equivalent. For example, $((u_1, \xi_1)(u_2, \xi_2)(u_1, \xi_3)$ (see Fig.3) cannot be expressed using a 3-state FSM unless one is willing to augment the the FSM with an additional variable that will store information on the execution history of the string. The transition functions ξ_i will also have to be altered (their domain must include the additional variable). Conversely, there are FSM that cannot be translated to MDLe strings (one can easily construct instances of Eq. 2 whose transition matrices are not linear combinations of the three types discussed above). Perhaps the simplest example is a FSM that implements branching (see Fig.4)

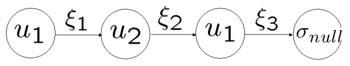


Fig. 3. An MDLe string that has no FSM equivalent.

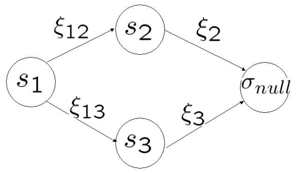


Fig. 4. A FSM that has no MDLe equivalent.

where states s_1, s_2 and s_3 are identified with MDLe atoms or encapsulated substrings. The FSM of Fig. 4 cannot be represented in MDLe because it corresponds to

$$\Pi(t) = \begin{bmatrix} 0&0&0&0\\1&1&0&0\\0&0&1&0\\0&0&0&1 \end{bmatrix}\xi_{12} + \begin{bmatrix} 0&0&0&0\\0&1&0&0\\1&0&1&0\\0&0&0&1 \end{bmatrix}\xi_{13} + \begin{bmatrix} 1&0&0&0\\0&0&0&0\\0&0&1&0\\0&1&0&1 \end{bmatrix}\xi_2 + \begin{bmatrix} 1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&1&1 \end{bmatrix}\xi_3 \quad (6)$$

and one can easily check that there is no renumbering of states for which (R1)-(R4) of Theorem 1 are satisfied. With MDLe, we have no choice but to designate either ξ_{12} or ξ_{13} as a behavior-level interrupt; in either case we are forced to include additional transitions (from s_2 to s_3 for example or vise versa, depending on which of those atoms is in the same behavior as s_1). If we are allowed to augment the state of the KSM by an additional variable z (which could be used to store 1 if ξ_{12} is triggered before ξ_{13} when s_1 is running and 0 otherwise) then the time evolution of the FSM in Fig. 4 is equivalent to that of: $(s_1, (\xi_{12} \texttt{ AND } \xi_{13}))(s_2, (\xi_2 \texttt{ AND } z))(s_3, (\xi_2 \texttt{ AND NOT } z))\sigma_{null}$.

The previous example suggests that one can find an MDLe equivalent for any FSM-like control program if we allow a sufficient (but finite) number of auxiliary variables. The question of equivalence of FSMs and MDLe strings will not be explored further here, as it is beyond the scope of this paper.

C. *Extensions - MDLe for Formations of Multi-modal systems*

In this paper we are mainly concerned with the control of single kinetic state machines. Recently there has been some work that explores extensions of the language that can enable the control of formations of systems. Briefly, [17] consider a collection of KSMs (specifically mobile robots) whose state vectors are concatenated to form an extended state vector x. One can then define a new set of so-called "group atoms" whose control quark U is a function of the extended state vector and whose interrupt ξ depends on sensor information from all robots in the collection.

There are two hurdles to overcome in this setting. First, each group atom must eventually be broken down to atoms for each robot. Second, in order to implement a group atom, robots have to communicate their state and sensor information to one another. For details on group kinetic state machines and group atoms as well as the process of generating atoms from group atom we refer the reader to [17].

III. Software

Of course, we are interested in programs that will run on physical hardware. In this section we describe a relatively complete and self-contained software environment that enables the execution of MDLe programs on a general purpose computer. We proceed to describe the features of this software and explain its operation.

There are two important characteristics in our vision for a control-oriented software organization. One has to do with the "traditional" separation of the user from the low-level details of the hardware. As we have already argued, it is not sufficient to encapsulate device drivers, because for different KSMs the control law expressions, dimension of the state, kinematic configuration, etc, may change. Robustness with respect to such changes cannot be achieved unless we also encapsulate control laws and this is exactly what an MDLe atom does. The second characteristic is the separation of run time components that handle computation, control and communication. This is necessary for code upgradeability, maintainability and because we would like to avoid maintaining large "monolithic" source code that would have to be recompiled every time we add/change hardware.

Any MDLe plan will always be executed one atom at a time (with some atoms omitted, depending on the return values of certain interrupt functions). One could then – in principle – use the syntactic rules of MDLe to compose plans as long as every atom has first been translated into code. To simplify matters we will temporarily focus on the software infrastructure required to execute an single-atom program (u, ξ, T), before discussing more complicated plans.

An atom's feedback control law is implemented in a single run-time process, named the **Modular Engine** (ME) for reasons that will become clear in the development. The ME uses a system timer to enforce a periodic control cycle[6], during which the atom (the executable code implementing its feedback loop) will be evaluated. During a control cycle, the ME process needs to execute several pieces of code, including:

- device drivers for interfacing with the hardware,
- routines that process sensor data (e.g implementing the observation equation $y = h(x)$ of a kinetic state machine),
- the control law and interrupt function of the atom.

A. *A modular software architecture*

Because our controller runs on a (single processor) digital computer, the software components outlined above will necessarily have to share the CPU, and it is common practice to include sensing, computation and actuation sequentially in a single executable program. The pseudocode corresponding to the contents of one control cycle might take the following form, common to many control programs:

```
retrieve current sensor data
evaluate $y(t))$
evaluate $\xi(y(t))$
if ($\xi(y(t)) == 1$ and $t - t_0 < T$) {
   compute $u = u(y(t), t)$
   send $u$ to the actuators
} else
end
```

where t_0 is the time at which the atom begins to run.

In practice, one often translates the pseudocode shown above to compilable source code - all that is required is for the user to insert functions that perform the computations of $u(y, t)$, $\xi(y)$ and provide calls to the device drivers that handle hardware I/O. The source is then compiled to produce an executable which will implement the specified atom. Despite its simplicity, this "monolithic" approach has two serious drawbacks. The first one has to do with the (re-)usability and maintainability of the code: over the lifetime of a robot – or even during the execution of a program – the collection of available sensors or signal processing routines may change (for example, due to hardware upgrades or malfunctions). To incorporate such changes to the control software one would have to modify and re-compile the entire source code. The second drawback is apparent when we consider the fact that the expressions for computing u and ξ must change (at non-deterministic times) during the execution of an MDLe string. While it is possible to create an off-line parser that accepts an MDLe string and outputs C++ code that implements the string, this would still leave the user having to re-compile the source every time a new MDLe program is written. Clearly this is not acceptable, in addition to leading to large and inefficient code.

Instead, we propose a software architecture in which various components (such as I/O, data processing and control evaluation functions) are maintained and compiled separately into run-time libraries, to be loaded as needed (by the ME process) during execution time. This would be an especially desirable situation from the point of view of code maintainability,

[6]In the absence of a real-time operating system we use the term "control cycle" to refer to a pseudo-periodic execution of an atom.

allowing the ME to respond to changes in signal processing methods, sensor or actuator availability and would lead to more efficient executables (e.g. no calls will be made to a frame grabber if the MDLe program currently running does not require image data). To achieve this flexible run-time environment, we need at least a standardized interface between components as will as a way for to coordinate their execution during run time. Towards that end, we define a special C++ class called **module** that provides a template for our generic software component.

An instantiation of the module class can be thought of as a data structure with pointers to functions that one would like to execute within the feedback loop (for example one that will evaluate $u(y,t)$, or retrieve odometry data), as well as a set of variables (termed **attributes**) that the module uses to receive or publish data (e.g. actuator commands to be sent to the motors or an image just retrieved from an on-board camera). Each module is compiled into a run-time library ($.so$ file).

The ME uses modules as the basic "blocks" from which to build a controller that will be modified during run time as an atom expires and another takes its place. Modules are loaded by the ME process and instantiated in separate threads that run concurrently, taking advantage of the operating system's POSIX scheduler. When a new module is loaded, ME obtains a list of contents in that module, including its name, attributes (with each attribute's designation as an input or output) and functions that the module can run. The ME can then make calls to the module's methods and access its attributes. That way, the ME can load modules whose functions perform hardware I/O, process sensor data and evaluate the observation $y = h(x)$, interrupt $\xi(y)$ and control $u(y,t)$.

B. Composing a Feedback controller

The ME controls the execution of modules by splitting the control cycle into two parts:

- the **turn** segment, during which all I/O, data processing and computation take place. During the turn the feedback law is evaluated and the results are sent to the kinetic state machine (robot).
- the **turn break** segment, during which data sharing and book-keeping takes place (e.g. a thread associated with a sensor publishes its data so that it can be used to compute the control at the next cycle).

We must make sure that the module responsible for sensing publishes its data to the module that will compute the next control sample, also that the controller will not keep producing outputs until it has received new data. This is done by requiring that each module contain two special methods:

- the "turn method" - this holds the code that the module will evaluate once during each control cycle, and
- the "turn break" method - code that is executed during the turn break segment.

At the beginning of every control cycle the ME initiates a *turn*, and passes CPU control to the threads associated with all modules modules. Each module executes its turn method and then signals the scheduling process that it has done so. The turn ends when all registered modules have completed their tasks. When a turn ends, the ME gives all modules the opportunity to share data by calling each module's turn-break method. The turn-break method is responsible for broadcasting data to the other processes. Under this scheme data is only transferred between processes during the turn breaks. At all other times modules run independently. During a turn, all turn methods run concurrently, sharing the CPU. In a turn break piece modules run sequentially. Turn-break methods are reserved for data sharing and may not use any function calls that block to avoid taking sole control of CPU.

We have in mind that there are always at least two modules running, one handling I/O and another implementing the feedback laws contained in atoms. Of course, additional modules can accommodate new sensor suites, post-processing, etc, keeping in mind any limitations on CPU power and the minimum loop closure rate that must be maintained. To conserve CPU power and memory, the module class includes "start" and "stop" methods, so that the execution of any module may be suspended during run time and "load" and "unload" methods to remove a module from computer memory. Such calls are made by the ME process during a turn break, which causes the module to cease executing (or be unloaded) at the next cycle. That way, modules that are irrelevant to the atom being executed can be stopped until they are needed again.

C. User-supplied source code

Robot Module: The interface between the kinetic state machine and the hardware is by necessity hardware-specific. This interface is coded in a *robot module* and encapsulates all device drivers necessary to interface with a robot's actuators and built-in sensors. In the case of a mobile robot, one may be given drivers that return the robot's position (from odometry or ranging sensors), sonar data, images taken from on-board cameras, and others that transmit inputs to the robot's wheels. The robot module serves to separate MDLe (and the user) from robot-specific code. Its attributes include readable variables that store the latest sensor data obtained (typically during the last control cycle) and writeable variables for specifying outgoing control signals and for changing the internal configuration of the robot (e.g. enabling or disabling sensors/actuators, if the hardware offers such capabilities).

Additional modules may be written to incorporate custom sensors or actuators that may be added to a robot (e.g. a vision system or a manipulator). All that is required from the developer's point of view is that the MDLe module be provided with pointers to the module(s) whose attributes must be accessed at run-time.

Atom Specification: The code to implement an atom (u, ξ, T) will take the form:

```
while ($\xi(y(t_k)) == 1$ and $t_k - t_0 < T$){
    retrieve current sensor data $y(t_k)$)
    compute $u_{k+1} = u(y(t_k), t_{k+1})$
    send $u_{k+1}$ to the actuators
}
```

where t_0 is the time at which the atom begins to run. Of course, our feedback loop is closed via a digital computer, so any control law $u = u(y(t), t)$ is necessarily be implemented in discrete time, with observation, computation and actuation taking place sequentially. The pseudocode listing given above represents the basic feedback loop that runs the kinetic state machine. The user is only responsible for providing the function calls necessary to compute $u(y,t)$ and $\xi(y(t)$. For example, a function $pd(k_p, k_d, x, \dot{x})$ that evaluates a PD controller $u = k_p x + k_p \dot{x}$ might be used to form the atom $a_{pd} = (pd, null, 10)$ that would apply that PD controller to the kinetic state machine for 10 seconds. To execute a_{pd}, a module is created with its turn method holding a pointer to the function pd. The source code for pd will be compiled within the module's run time library file and will be available to ME when that module is loaded. The situation is similar for the interrupt function of an atom, i.e. it is specified by a function call to an appropriate boolean function whose source code will be included in the run time library. Interrupts may be formed from a single call to a function, (e.g. $a_{pd} = (pd, inter1, 10)$), or as complex expression composed from interrupt functions and logical operators (e.g. $a_{pd} = (pd, (inter1\ OR\ inter2)\ AND\ inter3, 10)$).

D. Executing MDLe programs

Thus far we have assumed that the MDLe string running is a single atom. Of course, in general the atom will be part of a larger string; when the atom's interrupt function returns a logical `0`, the feedback law and interrupt functions evaluated during the control cycle must be replaced. This is accomplished by an MDLe interpreter module that is loaded by the ME and spawned into its own computational thread.

The MDLe module is responsible for "translating" MDLe strings down to individual atoms, and eventually to the (machine-dependent) executable code that will implement those atoms. The module's most important attribute, *main*, holds the MDLe program that is to be executed. This attribute can be set during run time by the user sending a request via the Modular Engine (user interaction with MDLe will be discussed in Sec. III-H).

As the MDLe module runs, it parses the plan stored in *main* and locates the first atom in that plan. Pointers to the control and interrupt functions of the atom are retrieved and placed in the module's turn method. In the following turn cycle, the ME calls the MDLe's turn method which follows the pointers and executes the feedback loop and interrupt evaluation. It is important to note that during the turn cycle, modules responsible for sensing are busy executing their own turn methods, therefore the sensor data used to evaluate the feedback loop must come from the previous turn cycle.

In the next turn break, MDLe checks the results of its last interrupt evaluation(s). If any of the interrupts has been triggered, then MDLe advances down the program being executed (skipping over the atom, behavior, etc. whose interrupt was just triggered) and replaces the pointers in its turn method with those corresponding to the next atom and interrupts to be evaluated. In the same turn break, the control computed during the previous turn cycle is made available to other modules, including those responsible for hardware I/O (see Robot module below). That control will be sent to the actuators in the following turn cycle, when the appropriate module has had a chance to run its turn method.

The programmer is responsible for providing a library of all low-level functions (control and interrupt quarks) that the MDLe program will require in order to run its atoms. 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. Typically, all available control laws and interrupt conditions are compiled into a run-time library together with the MDLe module.

E. Accommodating long computation

Normally, modules run exactly once per turn. This means that the turn methods should be kept short if we are to keep control cycle short. Besides modules that are critical to the closure of the feedback loop (e.g. code responsible for sensing and actuation), it may happen that a control program relies on sensor data that is intermittent or takes a long time to compute. An example could be an image processing algorithm, or communication over wireless network. An example might be a routine that performs time consuming signal processing or communication.

For this reason, our run-time architecture has been designed to accommodate such tasks in a special type of *unregistered module* (UM) that need not or cannot be evaluated once per cycle. UMs differ from registered modules in that their execution is not scheduled by the Modular Engine; instead they run in the background independently of the scheduler and do not affect the timing of critical tasks. This additional flexibility comes at a cost, however as UMs must eventually coordinate with RMs (the MDLe module for example) in order to publish their data. On the other hand, UMs are not under the control of the scheduler and do not know when a turn break will occur. This difficulty is circumvented by implementing a "callback" system by which a UM notifies the ME that it has data to exchange and then blocks its own execution. At the next turn break, the ME runs the module's callback function, publishes any data produced by the UM and unblocks the UM's process. A similar procedure allows UMs to receive data generated by other modules.

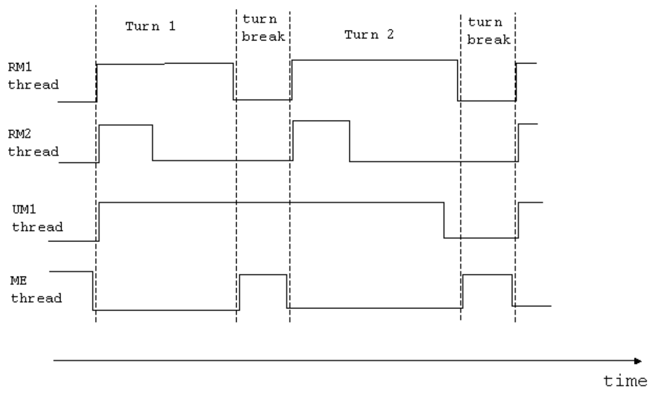


Fig. 5. Timing diagram in multi-threaded mode

Figure 5 illustrates the scheduling scheme in the multi-threaded mode. During the first turn multiple RMs run concurrently and the ME thread sits idle. After the longest RM has completed its task the first turn break begins. During the turn break the ME runs the callback functions of the RMs and any UMs that have requested it at which point the next turn begins. All callback functions run sequentially. The UMs run independent of the turns unless they are to share data with the other modules. In the diagram, the UM 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 (There will of course be additional delays because of operating system overhead for handling the thread switching).

F. A note on performance

The control cycle will not be strictly periodic (owing to the lack of real-time support in Linux), therefore turn lengths are highly dependent on the particular RMs running at the time. 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 loop closure rates can be guaranteed for critical atoms, despite the pseudo-real time performance of Linux. Migration of the MDLe Engine to RT-Linux, currently in progress, will enable real-time performance however as in most real-time operating systems there will be a tradeoff between the rate at which an atom can run and the amount of computation it can perform per turn. Each Registered Module runs as a separate

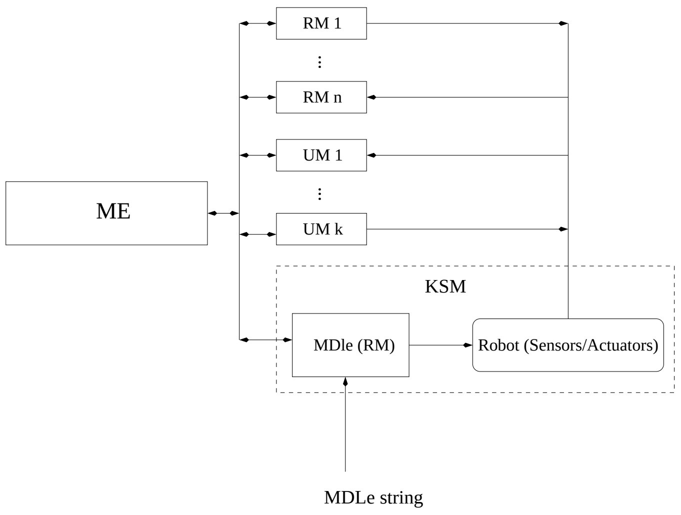


Fig. 6. Block diagram of the MDLe engine: A unique MDLe module together with the physical robot form the kinetic state machine (see Sec. II). Each sensor (actuator) is accessed by only one module which may share its data with other modules. Incoming MDLe programs are parsed and executed in the MDLe module. The Modular Engine synchronizes the operation of all modules.

thread and the length of a control cycle is determined by the slowest registered module Turn-break methods do nothing more than write to variables in memory, therefore turn breaks take a negligible amount of time compared with the run time of the registered modules. We have found that the multi-threaded approach adds complexity to the implementation but at the same time improves performance (compared for example to running each module as a separate process) and takes advantage of Linux's built-in POSIX scheduler.

G. Module Examples

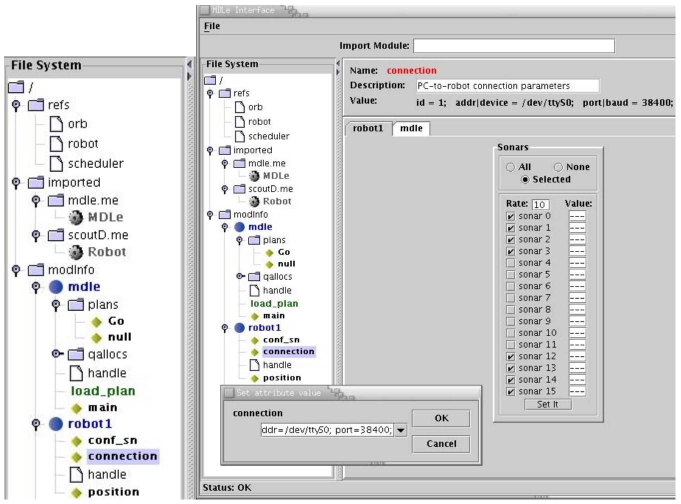
The MDLe and Robot modules described above form the basic infrastructure of any application. The following are some additional modules which are currently operational as part of our robotic testbed.

- **Sound localization module** This module implements a virtual sensor based on the data coming from a stereo input to the sound card on the robot. One of our robots is outfitted with a pair of microphones mounted on a spherical, acoustically hard head. The output of the microphones is run through a set of amplifiers and filters before being fed into the left and right channels of an on-board sound card. By correlating the left and right inputs, the module computes the inter-aural time delay (ITD) (i.e. length of time between when the sound first impinged on one ear and then the other). Assuming the sound source is away from the microphones, its direction of arrival (azimuth) can be approximated by solving
$$v_{sound} ITD = r\theta + r\sin(\theta) \tag{7}$$
for θ [18]. Here v_{sound} is the velocity of sound in air, r is the radius of the head, and θ is the heading angle with 0 radians being straight ahead and $\pi/2$ radians being to the left. Once the module determines the arrival angle it writes this data to the heading module described below.
- **Heading module** This module acts as a "virtual sensor" and provides a desired heading and range to a target. This data can be accessed by any other module and thus gives a simple way to implement target tracking in a sensor-independent way.
- **State link module** The state link module establishes a wireless communication link between with other nearby robots using TCP/IP sockets. This link is used to send data (including state and sensor information as well as MDLe strings) between the connected robots.

H. User Interaction

The ME maintains a list of existing modules and their attributes together with any MDLe plans that have been loaded, organized in a directory-like structure, termed the "file system structure" (FSS). Besides storing module names, attributes and available methods, FSS allows the user to view such data (organized by module name) as well as instantiate new modules. For example, if the code necessary to interface to the actuators of a robot is placed the module RobotA, loading RobotA into the ME creates a the directory entry /lib/RobotA. Selecting that entry instantiates the module and creates a subdirectory /usr/RobotA. In that subdirectory, entries are created for the module's attributes (e.g. a readable binary variable indicating whether the actuator system is functioning properly) and functions that the module can run (e.g. send a control sample to the motors). Attributes can be accessed by external objects via a CORBA interface and provide a way to communicate with, or alter the execution of a module.

The FSS can be navigated using a simple terminal interface that resembles a UNIX shell or by means of a Java GUI that displays the FSS graphically and sends commands to the ME (see Fig.7). Commands to the ME can be sent one at a time



Name: connection
Description: PC-to-robot connection parameters
Value: id = 1; addr|device = /dev/ttyS0; port|baud = 38400;

Rate: 10

connection: ddr=/dev/ttyS0; port=38400;

Status: OK

Fig. 7. Screenshot of the GUI to the MDLe Engine. The close-up of the FSS display (left) shows the currently allocated modules, MDLe and $scoutD$ (referring to the robot make). For the MDLe module, two plans are available (Go and Null) as well as a command $load\_plan$ for importing new ones. An attribute, $main$, contains the MDLe plan currently running (if any). The instantiation of the scoutD robot is named "pinky" and has three attributes, $conf\_sn$ corresponding to its sonar configuration (currently being altered in the main window), $connection$ with the computer controlling the robot and the robot's cartesian $position$.

over the shell or graphical interface. Alternatively, a set of commands (including MDLe programs) may be placed in a text file (much like a shell script) and sent to the ME. The existing interface gives the user access to the full state of the MDLe Engine. This includes the currently running modules and their attributes, available modules and MDLe plans that can be loaded, status information on the MDLe plan being executed, as well as commands for loading and terminating plans and modules. Additionally, a CORBA interface built into the ME enables the programmer to control the handling (loading, unloading, starting and stopping) of modules as desired.

I. MDLe and other languages

The work in [19], [20], describes a time-triggered programming language called Giotto, aimed principaly at embedded control applications. Like MDLe, Giotto is a development model aimed at bridging the gap between control system design and software implementation of that design. Giotto is well suited for hard real-time applications that feature periodic behaviors such as the control of hybrid systems. Some of the features of MDLe have "parallels" in Giotto, such as the use of interrupt mechanisms triggered by sensor outputs. There are other similarities between the two languages. Specifically, a Giotto *mode* combined with a *mode switch* is analogous to an MDLe atom. Giotto modes are made up of sets of tasks which are akin to control quarks. Unlike MDLe, Giotto does not provide for any higher-level abstractions such as the behaviors and plans of MDLe. This difference becomes important when software robustness and reusability are required (see our remarks in the Introduction). MDLe is intended as a suitable abstraction for hybrid system control, while Giotto provides an interface between embedded system and software. In fact it would be possible in principle to implement MDLe "over" Giotto.

Other efforts which are complementary to those described here include the CHARON [21] and Ptolemy projects [22], [23]. These software tools are oriented towards modular design of interacting hybrid systems with emphasis on simulation, prototyping and modeling.

IV. EXPERIMENTS

This section describes a series of experiments on the control of mobile robots via MDLe programs. For each experiment we briefly discuss the motion control task to be performed, give the MDLe programs that implemented the task and show the results of the programs' execution.

A. Hardware and Available Atoms

The experiments detailed below were carried out on a set of Nomadic Technologies Super Scout II and iRobot ATRV-mini robots. Both types of robots have differentially driven wheels and are outfitted with an array of sonar and touch sensors. An internal odometer keeps track of the robot's position and orientation with respect to some initial frame. One of the robots is additionally equipped with a pair of microphones used to localize sounds. The robots are connected to one another and to the Internet through a wireless Ethernet network. At the lowest level, each robot is controlled by means of left and right wheel velocity commands that are sent to dedicated on-board hardware. For convenience, our kinetic state machine model for each robot uses rotational and forward velocity commands (u_θ, u_f respectively) as inputs which are then converted to wheel velocities in the robot module described in Section III.

Based on the available sensors we implemented the following interrupt functions:

- `(bumper)`: returns 0 when the robot's bumper tape detects contact, 1 otherwise.
- `(atIsection b)`, where b is a 4-bit binary number: returns 0 when the sonar sensors detect obstacles (or absence thereof) in 4 principle directions with respect to the current orientation of the robot. Each digit in b selects whether the corresponding direction should be obstacle-free or not in the order (MSB to LSB): front,left,back,right. Used mainly to detect arrival at intersections.
- `(sync robotname)`: returns 0 if the state link module has established a communication link with robot *robotname*, 1 otherwise.
- `(wait` τ`)`: returns 0 if τ seconds have passed after an atom has begun to run, 1 otherwise.

For the experiments described below, a set of MDLe strings were designed from a small alphabet of atoms. The atoms' syntax is: *(Atom (interrupt condition) (control law))*. Under our previous definitions an atom (U, ξ, T) in the formal language is programmed as: *(Atom (ξ OR $t \geq T$) (U))*

- `(Atom (wait` ∞`) rotate (`α`) ):` $u_f = 0; u_\theta = k(\alpha - \theta)$. Causes the robot to make its orientation α with respect to its current coordinate system.
- `(Atom (bumper OR atIsection(`b`)) go(`v`,`ω`)):` $u_f = v; u_\theta = \omega$. Causes the robot to move with forward speed v cm/sec and turn rate ω rad/sec until it comes into contact with an obstacle or it arrives at an intersection specified in b.
- `(Atom (wait T) goAvoid(location)):` Causes the robot to move towards a point (r, ψ) specified in polar coordinates with respect to some initial position in the absence of nearby obstacles. *location* is specified in polar coordinates (by the programmer or by the heading module described in section III-G). If there are objects close to the robot in its desired path then the controls are modified to steer the robot along a safe path to the edge of the obstacle nearest the desired path.
- `(Atom (`$r_i(t)$ `==` $r_j(t)$`) (align` r_i r_j`)):` $u_f = 0; u_\theta = k(r_i(t) - r_j(t))$. Causes the robot to rotate until sonars i and j return equal ranges. Used to align the robot at a given orientation with respect to walls and other obstacles.
- `(Atom (bumper) Follow(robotName)):` $u_f = v; u_\theta = k(\theta_R - \theta)$ where v is a constant and θ_R is the orientation of the robot *robotname* as read from the state link module (the two robots must have established a communication link). Causes the robot to follow robot *robotname* using a "local pursuit" algorithm from [24] (this will be described in more detail in Sec. IV-D).
- `(Atom (bumper) path(file)):` $u_f = u_f(t); u_\theta = u_\theta(t)$ where $u_f(t)$ and $u_\theta(t)$ are read from *file*: Causes the robot to follow a predefined path.
- `(Atom (sync robotName) stop):` $u_f = u_\theta = 0$. Causes the robot to remain stationary until the state link module has established a communication link with robot *robotname*

Of course, each of our two robot types (Super scout or ATRV) comes with its own special set of low-level commands. This means that every MDLe atom (more specifically its control and interrupt quarks) had to be written in multiple versions, one for each type of robot. This one-time operation allows the MDLe programs given below to run on either type of robot with the same results.

B. A sound following robot

The sound localization module (described in section III-G) was used as a "virtual sensor" that estimates the direction of arrival of sounds in the laboratory. That estimate was passed to the module that controls the robot's heading. Combining this with the `goAvoid` atom yields a simple program for following sound sources:

$\Gamma_{soundFollow}$ = { `SoundFollowingRobot (bumper)`
`(Atom (wait` ∞`) goAvoid(heading module))`
}

Executing the above plan causes the robot to continually move towards the perceived direction of arrival of a sound while avoiding obstacles, until the bumper is pressed.

C. A Robot Navigation Task

Language-based descriptions of control tasks can be particularly effective in navigation problems. In a strategy motivated by human experience, one may forgo the requirement for a map (in some cases this may even be unavoidable) and instead use landmarks to move in the environment. In such a setting, language is most useful in allowing us to *replace (locally) details of a map by feedback programs*. In [25], two of the authors investigated exactly such an approach in which local maps of the relevant or interesting areas, termed landmarks, were stored in a directed graph structure and linked pairwise with MDLe plans (see figure 8). To illustrate the ideas we developed a simple indoor

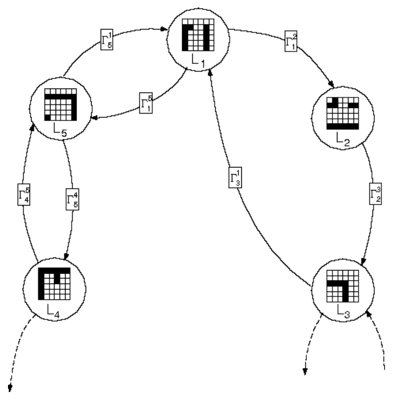
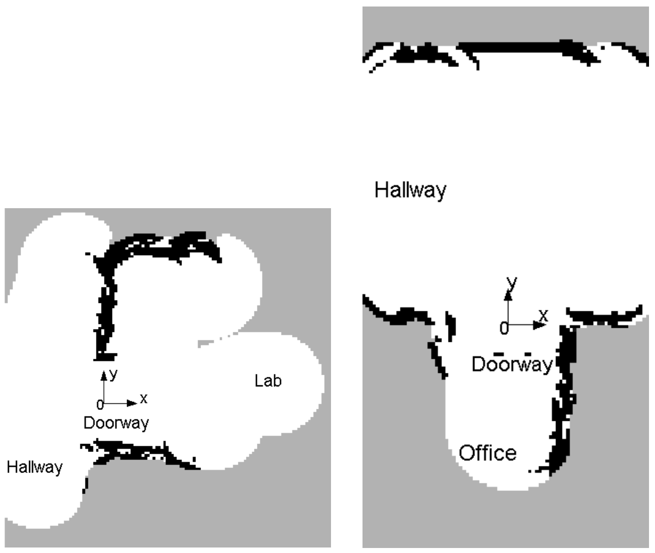


Fig. 8. A partial map structure

navigation experiment using a mobile robot. The goal of the experiment was to reduce a relatively large geographical area (part of an office building) into a graph which the robot could repeatably and safely navigate. We defined three landmarks, one around each of two doors to a lab and one around an office entrance. We used evidence grids [26] to describe local maps around the landmarks. Evidence grids represent terrain by an array of cells together with the probability that each cell is occupied (0 if a cell is empty, 1 if occupied, and 0.5 if the cell's occupancy is unknown). The coordinate systems associated to each landmark were chosen to be orthonormal with their origin at the center of each of the doorways. The evidence grid maps for two of the three landmarks are shown in figure 9-a, 9-b. The control inputs to steer the robot between landmarks were encoded in MDLe plans. Two of these plans, namely one to steer the robot from the rear of the lab (Lab 2) to the front of the lab (Lab 1), Γ^{lab1}_{lab2}, and one to steer the robot from the front of the lab to the office, Γ^{office}_{lab1}, are shown below.



(a) Landmark 1: Front lab door (b) Landmark 3: Office door

Fig. 9. Evidence grids surrounding each of two landmarks. Dimensions of each grid cell are 1 in. $\times 1$ in. and each map is 150×150 cells. Gray levels indicate the probability of a cell being occupied (0 for white and 1 for black).

```
$\Gamma^{lab1}_{lab2}$ = { Lab2ToLab1Plan (bumper)
  (Atom (atIsection 0100) (goAvoid 90 40 20))
  (Atom (atIsection 0010) (go 0 0.36))
  (Atom (wait ∞) align 7 9)
  (Atom (atIsection 1000) (goAvoid 0 40 20))
  (Atom (atIsection 0100) (go 0 0.36))
  (Atom (wait ∞) align 3 5)
  (Atom (wait 7) (goAvoid 270 40 20))
  (Atom (atIsection 1000) (goAvoid 270 40 20))
}
```

```
$\Gamma^{office}_{lab}$ = { Lab1ToOfficePlan (bumper)
  (Atom (atIsection 1001) (goAvoid 90 40 20))
  (Atom (atIsection 0011) (go 0 0.36))
  (Atom (wait ∞) align 11 13)
  (Atom (atIsection 0100) (goAvoid 180 40 20))
  (Atom (wait 10) (rotate 90));
}
```

Similar plans were created to fully connect the graph. These plans are not unique but they are fairly simple, and also quite similar to the set of directions one might give to someone unfamiliar with the floor layout. For example Γ^{office}_{lab1} can be read as: "walk down the hallway until you come to a corner, turn left, walk to the first open door on your right, turn right". When asked to go from landmark i to landmark j the robot checks for the existence of a connecting path in the graph (in this case an edge connecting the two landmarks in the specified direction), retrieves Γ^{j}_{i} from the graph, and executes it by interpreting each of its atoms into control signals that actuate the wheels. In figure 10 we show the graph structure, two of the connecting edges, and a blueprint (not to scale) of the environment along with a typical path produced by the execution of Γ^{lab1}_{lab2} followed by Γ^{office}_{lab1}. On the path produced by Γ^{lab1}_{lab2} an obstacle in the hallway (not indicated on the blueprint) caused the robot to temporarily move away from the wall.

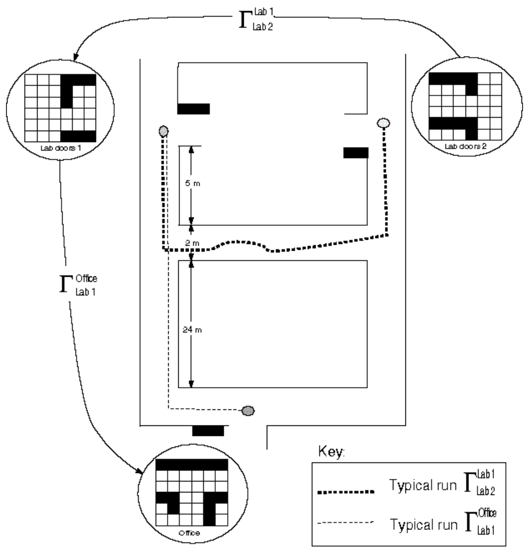
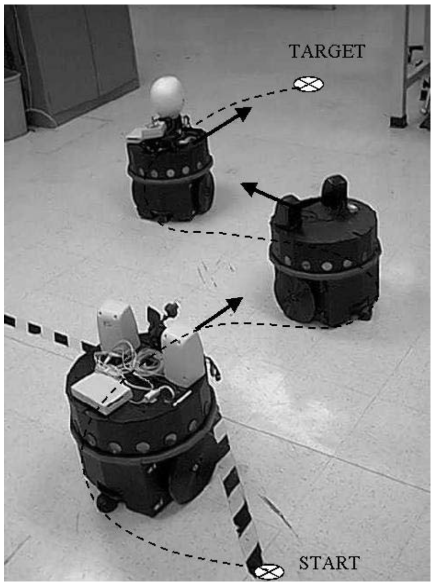

$\Gamma^{Lab 1}_{Lab 2}$
Lab doors 1
Lab doors 2
5 m
2 m
24 m
$\Gamma^{Office}_{Lab 1}$
Office
Key:
Typical run Γ^{Lab1}_{Lab2}
Typical run Γ^{Office}_{Lab1}

Fig. 10. A three-vertex graph containing the office and lab landmarks and the environment floor plan

D. Multi-robot motion control: learning minimum length paths

Finally, we describe an experiment involving cooperative path optimization again with control implemented using MDLe. It is known that there are some tasks which can be accomplished by groups cooperating agents, vehicles or other systems, but not by individuals. One of these tasks involves the optimization of trajectories on uneven terrain using only local sensing [24], [27]. More precisely, we consider a group of vehicles that must travel repeatedly between a pair of distant locations without access to a map of the terrain. The lack of a map prevents any vehicle from planning a path to the target. However, if an initial possibly circuitous path to the target is available (obtained from a combination of prior knowledge, sensor measurements and/or random exploration), that initial path can be optimized by a simple cooperative strategy that involves only short-range interactions between vehicles and local measurements of each vehicle's surroundings. That strategy, known as "local pursuit" [28], [24] requires vehicles to move in a "column" much like a line of marching ants. The leader of the column follows the initial, suboptimal path to the target with each of the other vehicles following the one ahead of it. A vehicle follows another by pointing its velocity vector along the shortest path (geodesic) connecting the two (see [24], [27] for a geometric view of the pursuit problem). It can be proved that the sequence of iterated paths taken by the vehicles converges to a path whose length is (locally) minimum.

An experiment was designed to illustrate the local pursuit algorithm with three indoor vehicles (robots 1 through 3) which were required to find the shortest path between the origin and a goal location $(3.75m, 0.75m)$. For the purposes of this experiment, we chose an initial path which allowed us to ignore the nonholonomic constraints of the robots. Thus, we took $\mathbb{R}^2$ as the configuration space for each robot, with the standard Euclidean metric on the plane, so that the shortest



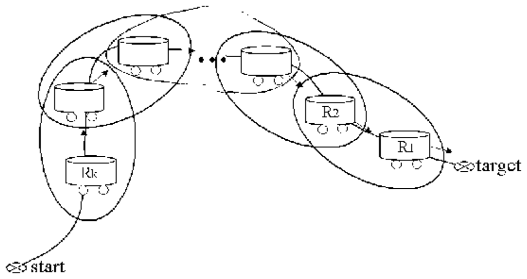


Fig. 11. Group of vehicles (a) with Decentralized communication (b)

path between any two points was simply a straight line. Robot 1 was given a predefined (suboptimal) path taking it from the origin to the goal location. Robots 2 and 3 followed robots 1 and 2 respectively, each implementing the local pursuit algorithm by maintaining a constant forward velocity while adjusting its turn rate so as to keep its leader directly ahead. Once at the target location the roles of the robots were reversed: robot 3 used its odometry data to retrace its path to the origin while the other robots followed. The sequence was repeated four times.

The MDLe plans for the first robot and second robots are shown below.

```
Γrobot1 = { FirstRobotPlan (bumper)
 ExecPlan 4 {
 (Atom (sync robot2) stop)
 (Atom (bumper) path(lastPath))
 (Atom (wait 2) go(0.5 0))
 (Atom (wait 10) rotate(180))
  (Atom (sync robot2) stop)
 (Atom (wait 2) go(0.5 0))
 (Atom (bumper) Follow(robot2))
 (Atom (wait 10) rotate(180)) }
}
```

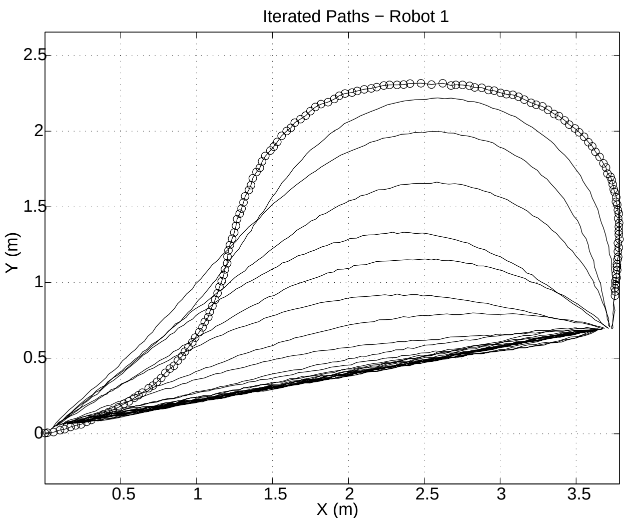
```
Γrobot2 = { SecondRobotPlan (bumper)
 ExecPlan 4 {
 (Atom ((sync robot1) AND (sync robot3)) stop)
 (Atom (wait 1) go(0.5 0))
 (Atom (bumper) Follow(robot1)
 (Atom (wait 1) go(0.5 0))
 (Atom (wait 10) rotate(180))
  (Atom ((sync robot3) AND (sync robot1)) stop)
 (Atom (wait 1) go(0.5 0))
 (Atom (bumper) Follow(robot3))
 (Atom (wait 1) go(0.5 0))
 (Atom (wait 10) rotate(180)) }
}
```

In Γ_{robot1}, `lastPath` refers to a file initially containing the first known path to the goal (in the form of open loop heading and forward velocity setpoints). That file is subsequently overwritten by the robot each time it traverses from the start to the goal or vice-versa, allowing it to retrace its steps. The MDLe plan for the robot 3 is similar to that for the robot 1, except that atoms are ordered roughly in reverse (it first follows robot 2, then retraces its own path back to the start location, taking on the role of the leader). Figure 12 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.

V. CONCLUSIONS

We discussed the use of motion description languages as tools that provide useful abstractions of multi-modal systems. We described the formal language MDLe and explored some of its properties and expressive power. We view MDLe as part of the foundation of a software and computational infrastructure that can support reusable control software, with the latter term taking on special meaning when applied to control systems as opposed to desktop computers. MDLe programs are composed from a dictionary of control primitives called “atoms”. An atom combines a control law and an interrupts condition which determines the atom’s “lifetime”. Interrupts and atoms can be composed hierarchically to form specifications of complex control tasks. The MDLe syntax: i) generates a rich set of control programs in a way that agrees with our intuition regarding the sequential nature of multi-modal control and ii) uses encapsulation as a means of managing the complexity of a control specification (a familiar example is that of giving someone directions to a distant location).

We described the basic features of an MDLe compiler and user interface which allow one to implement “device independent” motion control programs on physical hybrid systems. The software tools presented are known collectively as the *MDLe engine* and enable the development of hierarchically structured programs which accommodate switching logic, differential equations, and interaction with hardware and with the environment. The MDLe engine translates MDLe strings into C/C++ and compiles them to hardware-specific executable code. Hardware-specific routines for actuation and



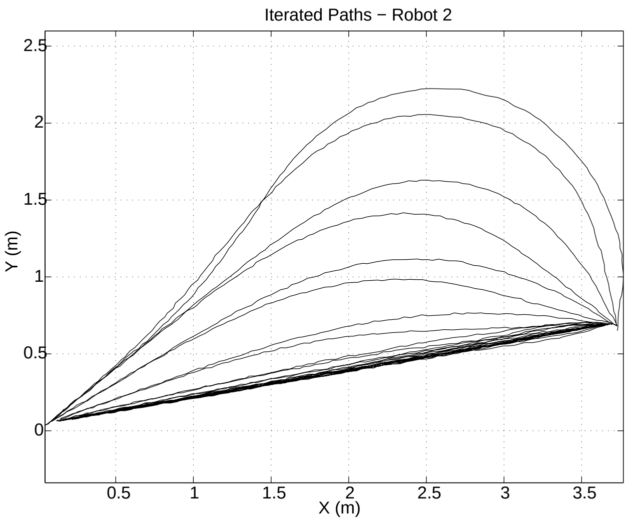


Fig. 12. Iterated paths created by "following" (from [24]): (a) first and (b) second robots.

sensing are linked at compile time so that MDLe programs are in principle portable.

The performance of an MDLe program depends on the hardware capabilities of the machine which is attempting to execute that program. Our implementation provides for atoms with varying amounts of computational requirements by allowing each atom to spawn multiple threads that will perform any necessary pre-processing (for example, a controller may servo on a few bits of information extracted from an incoming video stream). 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 burden of that atom.

We presented examples of MDLe programs and the results of their execution on autonomous mobile robots. We believe that the flexibility and device independence of MDLe make it a useful tool for practical hybrid control. We plan to make the MDLe Engine available on the Web. Future work is directed towards the development of a real-time version of MDLe under RTLinux as well as a set of graphical programming tools. By freeing the user from always having to design controllers at the level of sensors and actuators, a control-oriented language like MDLe offers the opportunity to make contact with long-standing work in formal languages and logic, and to explore their intersection with differential equation-based control. Research directions which in our opinion are important to pursue, include the specification and refinement of high-level descriptions of control tasks, planning (how to find the right sequence of atoms to accomplish a task) and language selection (striking a balance between the number of atoms in the language and their specialization).

References

[1] M. Egerstedt and R. Brockett, "Feedback can reduce the specification complexity of motor programs," *IEEE Trans. Robotics and Automation*, to appear.

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

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

[4] ——, "Formal languages for motion description and map making," in *Robotics*. American Mathematical Society, 1990, pp. 181–93.

[5] ——, "Hybrid models for motion control systems," in *Perspectives in Control*, H. Trentelman and J. Willems, Eds. Boston: Birkhäuser, 1993, pp. 29–54.

[6] V. Manikonda, J. Hendler, and P. S. Krishnaprasad, "Formalizing behavior-based planning for nonholonomic robots," in *Proceedings 1995 International Joint Conference on Artificial Intelligence*, vol. 1, August 1995, pp. 142–9.

[7] 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*, vol. 2, May 1995, pp. 2021–8.

[8] ——, "Languages, behaviors, hybrid architectures and motion control," in *Mathematical Control Theory*, J. W. J. Baillieul, Ed. Springer, 1998, pp. 199–226.

[9] R. A. Brooks, "Intelligence without reason," MIT, Tech. Rep. A.I. Memo No. 1293, 1991.

[10] ——, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.

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

[12] D. Hristu-Varsakelis and R. W. Brockett, "Experimenting with hybrid control," *IEEE Control Systems Magazine*, vol. 22, no. 1, pp. 82–95, Feb. 2002.

[13] R. M. Murray, D. C. Deno, K. S. J. Pister, and S. S. Sastry, "Control primitives for robot systems," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 22, pp. 183–193, 1 1992.

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

[15] A. Meduna, *Automata and Languages*. London, U.K.: Springer, 2000.

[16] D. Hristu-Varsakelis and M. Egerstedt, "The complexity of the motion description language MDLe," in *submitted to the IEEE Conference on Decision and Control*, Dec. 2003.

[17] F. Zhang, M. Goldgeier, and P. S. Krishnaprasad, "Control of small formations using shape coordinates," in *Proceddings of the 2003 IEEE Int'l Conference on Robotics and Automation*, 2003, to appear.

[18] G. Kuhn, "Model for the interaural time differences in the azimuthal plane," *Journal of the Acoustical Society of America*, vol. 62, no. 1, pp. 1567–167, July 1977.

[19] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: a time-triggered language for embedded programming," in *Proceddings of the First International Workshop on Embedded Software*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2001.

[20] ——, "Embedded control systems development with giotto," in *Workshop on Languages, Compilers, and Tools for Embedded Systems*, vol. 36, 2001, pp. 64–72.

[21] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, and G. Pappas, “Hierarchical hybrid modeling of embedded systems.” in *Proceedings of the Workshop on Embedded Software*, 2001.

[22] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “A framework for simulating and prototyping heterogeneous systems.” in *International Journal of Computer Simulation*, vol. 4, April 1994, pp. 155–182.

[23] E. A. Lee, “Overview of the ptolemy project.” University of California at Berkeley, Department of Electrical Engineering and Computer Sciences, Tech. Rep. UCB/ERL M01/11, March 2001.

[24] D. Hristu-Varsakelis, “Robot formations: Learning minimum-length paths on uneven terrain,” in *Proc. of the 8th IEEE Mediterranean Conference on Control and Automation*, July 2000.

[25] D. Hristu-Varsakelis and S. Andersson, “Directed graphs and motion description languages for robot navigation and control,” in *Proc. of the IEEE Conf. on Robotics and Automation*, 2002, pp. 2689–2694.

[26] M. Martin and H. Moravec, “Robot evidence grids,” The Robotics Institute, Carnegie Mellon University, Tech. Rep. CMU-RI-TR-96-06, 1996.

[27] D. Hristu-Varsakelis, “Optimization by local pursuit,” *subm. to IEEE Transactions on Robotics and Automation*.

[28] A. M. Bruckstein, “Why the ant trails look so straight and nice,” *The Mathematical Intelligencer*, vol. 15, no. 2, pp. 59–62, 1993.