On The Structural Complexity of the Motion Description Language MDLe

D. Hristu-Varsakelis* , M. Egerstedt† , P. S. Krishnaprasad‡
{hristu@eng.umd.edu, magnus@ece.gatech.edu, krishna@isr.umd.edu}

***Abstract*— As modern control theory attempts to elucidate the complexity of systems that combine differential equations and event-driven logic, it must overcome challenges having to do with limited expressive power as well as practical difficulties associated with translating control algorithms into robust and reusable software. The Motion Description Language (MDL) and its "extended" counterpart MDLe, have been at the center of an ongoing effort to make progress on both of these fronts. The goal of this paper is to define MDLe as a formal language, thereby connecting with the vast literature on the subject, and to stimulate experimental work. We discuss the expressive power of MDLe and provide some examples of MDLe programs.**

I. Introduction

The development of increasingly complex engineering systems, where modern control theory has to coexist with event-driven logic, has brought to the fore the need for models that are expressive yet conceptually and computationally tractable. For example, knowing that a robot is controllable (by checking the properties of the appropriate differential equation-based model) tells us little about whether the robot can be controlled from one location to another in any reasonably complex environment. (There are holes or obstacles along the way, etc.) At the same time, questions concerning the complexity of a control program [9], [8] cannot easily be answered unless one adopts "suitable abstractions" of multi-modal systems that treat differential equation-based models as low-level details, to be accessed only when necessary. These abstractions are equally important when it comes to translating algorithms into control software whose portability approaches that of its desktop counterpart, while encapsulating the details of the control system (e.g. number of wheels, link lengths, etc) in software objects that are transparent to the programmer.

One attempt to address these challenges began over a decade ago with the "Motion Description Language" (MDL) developed in [2], [3], [4]. The idea was to provide a formal basis for robot programming using "behaviors" (structured collections of control primitives) while incorporating kinematic and dynamic models of robots in the form of differential equations. The work in [18], [19], [20] (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 [1], [2], [3], [5], [6], [14], [20], [22] and references therein. For additional recent work on abstractions, see [11], [23], [24], [7], [17].)

For this linguistic approach to control to be meaningful, one must be able to answer system-theoretic questions at the level of control primitives and their compositions, as opposed to analog signals through sensors and actuators. Work along these directions is already underway ([9], [10]), but the connections between MDLe (and other similar constructs) and the theory of formal languages and theoretical computer science has remained largely unexplored. This paper, following [13], [20], provides a précis of the discussion in [13], gives a definition of MDLe as a formal language, (thereby making precise some earlier statements), and discusses its expressive power.

II. The Motion Description Language MDLe

In this section we distill the early descriptions of MDLe (see [3], [18], [19], [20]) into a formal language definition that can be used for exploring the expressive power of the syntax. We begin with a brief discussion of MDLe's syntax. What we have in mind is that there is an underlying physical system (equipped with a set of limited range 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 [20]), which stands as an abstraction between the simplest elements of a control language (yet 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 an *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, for some given $T > 0$. 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 the p-dimensional sensory data, and $T \in \mathbb{R}^+ \cup \{\infty\}$ denotes the 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 "high" (logical `1`) or until T units of time elapse, whichever occurs first.

The first and third authors were 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. The second author was supported by NSF through the programs EHS NSF-01-161 (grant # 0207411) and ECS NSF-CAREER award (grant # 0237971).

∗ Department of Mechanical Engineering and Institute for Systems Research, University of Maryland, College Park, MD 20742.

† Corresponding author. Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332.

‡ Department of Electrical and Computer Engineering and Institute for Systems Research, University of Maryland, College Park, MD 20742.

In order to manage the complexity of hybrid control tasks and allow one to write reusable programs, a language suitable for control should support hierarchical levels of encoding. The idea is to let programs be put together from simpler programs, all the way down to hardware-specific functions. Towards that end, 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 "high" (logical `1`), or T_b units of time have elapsed, or σ_2 has terminated. Behaviors themselves can be composed to form higher-level strings, called *plans*, e.g. $((b, (u_3, \xi_3, T_3)), \xi_p, T_p)$.

III. MDLe as a Formal Language

Previous work characterized MDLe loosely as a language over the alphabet of atoms [19]. This is not entirely correct since it ignores the grammatic rules of the language. For example, strings that include nested structures are not concatenations of atoms. It is true that every execution/evaluation trace of an MDLe program (with nested behaviors) is indeed a string over the alphabet of atoms, however that string is determined by the interrupt triggering sequence which is not known a priori. This leaves open the question of generative structures of MDLe programs and leads naturally to thinking about MDLe as a language over an alphabet that refines atoms, as in the definition below, following [13].

Formally, we let $\mathcal{U}$ be a *finite* subset of $\{u : \mathbb{R}^p \times \mathbb{R}^+ \to \mathbb{R}^m\}$, i.e. $\mathcal{U}$ is the finite set of possible control laws (including the trivial $u_{null} = 0$). Furthermore, $\mathcal{B}$ is a finite subset of $\{\xi : \mathbb{R}^p \to \{0, 1\}\}$, i.e. it is a finite set of boolean functions of p variables (including the null interrupt $\xi_{null} : \mathbb{R}^p \to 1$). However, it is convenient to "absorb" an atom's timer into the atom's 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 convention we will say that an atom is made up of a *control quark* selected from $\mathcal{U}$ and an *interrupt quark* from $\mathcal{B}'$ which is a finite subset of $\{\xi : \mathbb{R}^p \times \mathbb{R}^+ \to \{0, 1\}\}$. Now, it is clear that the sets of control and interrupt quarks together with the special symbols "(", ")" and "," define a finite alphabet over which the MDLe strings are formed.

Definition 3.1: MDLe is the formal language generated by the context free grammar $G := (N, T, S, P)$, where:
N is the set of non-terminal symbols E, where E is a valid MDLe string;
T is the set of terminal symbols, i.e. the set of symbols that form the strings of the language. In other words, $T = \mathcal{U} \cup \mathcal{B}' \cup \{(), \}$;
S is the set of start symbols that represent the language that is being defined, i.e. S is the set of valid MDLe strings; and $P \subset N \times (N \cup T)^*$ is a finite relation which consists of the following production rules:

(P1) $E \to \epsilon$
(P2) $E \to (u, \xi),\ u \in \mathcal{U},\ \xi \in \mathcal{B}'$
(P3) $E \to EE$
(P4) $E \to (E, \xi),\ \xi \in \mathcal{B}'$

An immediate consequence of Definition 3.1 is that MDLe is a *context free language* (see for example [16], [21]). Consequently, one can write a compiler for MDLe and in fact we have done so (see Sec. V and [13]). However, it also follows that MDLe does not define a regular language:

Theorem 3.1: MDLe is not a regular language
Proof: Assume that MDLe is regular. Then, by the pumping lemma (e.g. [21]), there is a positive integer k such that for all strings s in the language with $|s| \geq k$, we can write $s = uvw$ with $v \neq \epsilon$, $|uv| \leq k$, and $uv^m w$ is in the language for all $m \geq 0$. In particular uw should be in the language (setting $m = 0$). Now consider the string

$$s = ((\ldots(((u, \xi), \psi_1), \psi_2), \ldots, \psi_{N-1}), \psi_N),$$

formed by the repeated application of production rule *P4*, and where N is much greater than k. This means that the first k symbols in s are in fact "(". This furthermore implies that any prefix uv of s of length less than or equal to k is just a string containing "(" repeatedly. Since $v \neq \epsilon$, v has to contain a positive number of left parentheses, and by pumping on v, i.e. by forming $uv^m w$, we get more (if $m > 1$) or fewer (if $m = 0$) left parentheses than right parentheses. Such an expression can never be formed from the production rules, hence MDLe is in fact not a regular language. ∎

Definition 3.1 is useful for discussing certain aspects of the language (including its complexity) but is not always convenient when it comes to writing control programs. In practice, it is easier to compose MDLe strings bottom-up, i.e. starting with atoms and composing behaviors, etc, as opposed to using the top-down productions of the grammar G. In particular,

Definition 3.2: MDLe is the collection of strings generated from the following rules:

(R1) **Encapsulation:** If s is a valid MDLe string then (s, ξ) is a valid MDLe string as well for any $\xi \in \mathcal{B}'$. We refer to s as an **encapsulated** substring, and to ξ as the interrupt **associated** with s;
(R2) **Concatenation:** If $s_1, \ldots, s_n$ are valid MDLe strings, then so is $s_1 \cdots s_n$; and
(R3) **Looping:** If s is a valid MDLe string, then so is s^n, for any $n \in \mathbb{Z}_+$.

The last rule (looping) is added for notational convenience and does not alter the complexity or expressive power of the language; one could add productions to Def. 3.1 in order to include exponents or insist that repeated substrings are explicitly written out without the use of exponents. The set of strings constructed using $R1, \ldots, R3$ is in fact the same set which the production rules $P1, \ldots, P4$ generate.

Fact 3.1: Definitions 3.2 and 3.1 define the same language (after all loops have been expanded).
Proof: Let L be the set of strings generated by the grammar G and L' be the strings generated by the rules $R1, R2, R3$. Now, take any $s' \in L'$ and "expand" any loops by replacing any substring s_1^n by n copies of itself in s'. We will construct s' using the production rules of the grammar G. (We will denote this string by s.) Now, s' was produced from simpler

strings (or quarks) by applications of $R1$ and $R2$ in a given order. If the last operation in that order was a concatenation (by $R2$) of substrings $s_1, ..., s_n$, then use $P3$ on s, $n-1$ times. If the last operation was encapsulation (s_1, ξ) (using $R1$) then apply $P4$ to s. Now, apply the same procedure to each of the substrings that were concatenated or encapsulated to form s', each time modifying s via applications of $P3$ or $P4$. This algorithm will terminate when we reach the first in the series of operations to construct s', at which point we apply $P2$ repeatedly in order to eliminate occurrences of non-terminal symbols. Having done so, we have $s = s'$, implying that $L' \subset L$.

For the converse, take any $s \in L$, produced by applications of $P1$ through $P4$. Beginning with the last production and proceeding towards the first, we can then construct s' using the rules $R1$ and $R2$ to build atoms which are then concatenated by $R2$ or encapsulated by $R1$. This procedure results in a string s' which is identical to s, implying that $L \subset L'$. ■

IV. Hybrid Automata and MDLe

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 but by a *hybrid automaton*. This is a widespread modeling tool for characterizing heterogeneous models (e.g. [12]), such as multi-modal control procedures. In this paper, we focus our attention on a particular class of hybrid automata whose discrete states are identified with individual control quarks, while state-to-state transitions occur in response to interrupts. An example is shown in Fig. 1.

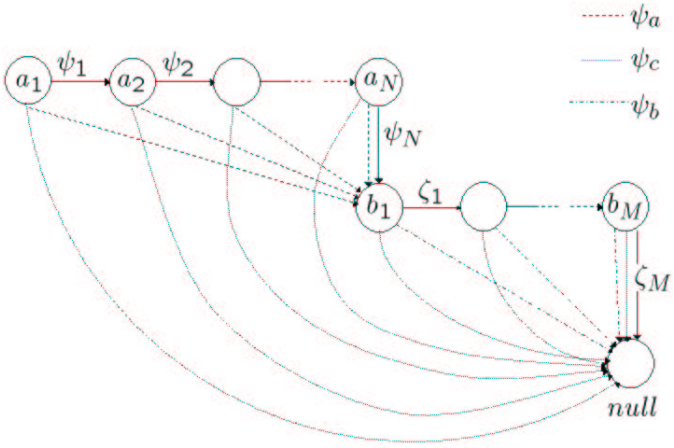

Fig. 1. Hybrid automaton representation of a two-behavior MDLe plan: $((a, \psi_a)(b, \psi_b), \psi_c)\sigma_{null}$, where $a = ((a_1, \psi_1) \cdots (a_N, \psi_N))$, $b = ((b_1, \zeta_1) \cdots (b_M, \zeta_M))$, and $\sigma_{null} = (u_{null}, \xi_{null})$

A kinetic state machine executes a program defined on such a hybrid automaton by running the control law specified by the current discrete state until a transition to a new state (corresp. new control law) occurs. This representation of motion control programs may seem more expressive than MDLe strings - after all we have not imposed any syntactical restrictions on the transitions. However, as we will see, this is not necessarily the case. To see this, we first require a notion of equivalence between the two representations, and we say that a hybrid automaton is *equivalent* to an MDLe string if they both produce the same trajectories on the same kinetic state machine, starting from the same initial conditions.

Now, we furthermore say that a MDLe string s is *degenerate* if an interrupt ξ is associated with an encapsulated substring of $s' = (s_1, \xi, T)$ of s and with s_1 or a substring of s_1. The interrupt ξ is then said to be *repeated* within s'. If s is degenerate then there is an ambiguity with respect to which atom should be executed when ξ is triggered. We will 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 `1`. We can always convert a degenerate string s to a non-degenerate one [13]. This is done by identifying occurrences of a repeated interrupt and replacing all but one (that which corresponds to the highest-level substring), with the null interrupt.

Formally, a correspondence between an MDLe string s and an equivalent hybrid automaton can be made as follows: Assume that we may use n control quarks, and 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 hybrid automaton (resp. control quarks in a MDLe string) with vectors in $\mathbb{R}^n_e$, for which the following monomials come in handy:

$$\Pi(t) \triangleq \sum_{i=1}^{M} E_i \xi_i(t). \tag{2}$$

Here M is the number of distinct interrupts in a given MDLe string, the *transition matrices* $E_i \in \{0,1\}^{n \times n}$ have columns taken from $\mathbb{R}^n_e$, and $\xi_i(t) \in \{0,1\}$ denotes the i^{th} interrupt in the string. If the control quark running at time t is identified by $e(t) \in \mathbb{R}^n_e$, and ξ_i is triggered at time t, then the new control quark will be $E_i e(t)$. The evolution $e(t) \rightarrow E_i e(t) = e(t^+)$ 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 (rules $R1, R2, R3$) 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

$$E_i = \begin{array}{c} \\ \\ \\ \\ \\ (k+1)^{st} \rightarrow \\ row \\ \\ \\ \end{array} \begin{bmatrix} 1 & 0 & \cdots & \cdots & & & & \cdots & 0 & 0 \\ 0 & 1 & 0 & & & & & & & 0 \\ \vdots & 0 & \ddots & & & & & & & \\ & \vdots & & 1 & & & & & & \\ & & & & 0 & & & & & \\ & & & & \vdots & \ddots & & & & \\ & & & & 0 & & 0 & & & \\ & & & & 1 & \cdots & 1 & 1 & & \\ \vdots & \vdots & & & & & & & \ddots & \\ 0 & 0 & \cdots & & & & & \cdots & 0 & 1 \end{bmatrix}$$

$$\underbrace{\qquad\qquad}_{(k-l)...(k)^{th} columns}$$

sending states $k-l, k-l+1, ..., k$ to $k+1$ (recall that atoms were numbered sequentially, which will ensure that 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, and all remaining entries being 0.

Because we always convert strings to their non-degenerate form, and under the additional assumption that only one interrupt function may change value at any given time[1], the transition (temporally) from any atom to the next will be unambiguous. Hence, 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 $[0,T]$, we can write

$$\begin{aligned} e(t) &= \Pi(t_m)...\Pi(t_2)\Pi(t_1)e(0) \\ &= E_{i(t_m)}...E_{i(t_2)}E_{i(t_1)}e(0), \quad t \geq t_m \end{aligned} \tag{3}$$

where $i(t_k),\ k = 1, ..., m$ are the indices of the interrupts that were triggered at $t_1 < t_2 < ... < t_m$. At the same time, the state evolution of the underlying KSM can be expressed as

$$x(t) = \phi(t_m, t, ...\phi_{i(t_2)}(t_1, t_2, \phi_{i(t_1)}(t_0, t_1, x_0)))) \tag{4}$$

where $\phi_i(t_j, t_k, x_0),\ t_j < t_k$ is the flow of the 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 $e(t_j^+)$.

Now, given a hybrid automaton (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 (Eq. 2) leads to the following result whose proof can be found in [13]:

[1]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

Theorem 4.1: Given the hybrid automaton representation of a motion control program

$$\Pi(t) \stackrel{\triangle}{=} \sum_{i=1}^{M} E_i \xi_i(t); \quad E_i :\ n \times n$$

there exists a non-degenerate MDLe string equivalent to Π if and only if there exists a renumbering of the discrete states in the hybrid automaton (renumbering of rows and columns of all E_i's) such that:

(S1) For $k = 1, ..., n-1$ there is a unique index $i(k)$ such that $E_{i(k)}$ is the identity matrix modified in such a way that its (k,k) entry is set to 0 and its $(k+1,k)$ entry is set to 1 (all "atom"-level transitions are present).

(S2) 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.

(S3) 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 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 $j < k_1$.

(S4) 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 $j < q$.

It should be clear that since we insist that repeated appearances of an atom are identified with a distinct discrete state in the hybrid automaton, there exist MDLe strings that have no hybrid automata equivalent. For example, $(u_1, \xi_1)(u_2, \xi_2)(u_1, \xi_3)$ (see Fig.2-(b)) cannot be expressed using a 3-state hybrid automaton unless one is willing to augment the automaton 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 hybrid automata 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 hybrid automaton that implements branching (see Fig.2-(a)), where states s_1, s_2 and s_3 are identified with MDLe atoms or encapsulated substrings.

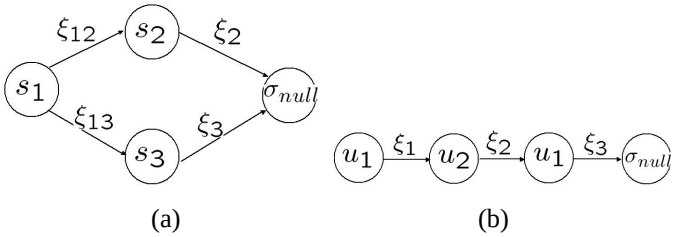

Fig. 2. (a) A hybrid automaton that has no MDLe equivalent. (b) An MDLe string that has no hybrid automata equivalent.

The hybrid automaton of Fig. 2-(a) cannot be represented in MDLe because under the rules of the language 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). In fact, one can easily check that there is no renumbering of states for which $S1, \ldots S4$ of Theorem 4.1 are satisfied.

V. The MDLe Software Engine

Recent work [15] at the University of Maryland has produced an implementation of MDLe, including an MDLe compiler, user interface, and related software tools. This software platform (dubbed the "MDLe Engine") includes facilities for parsing an MDLe plan and interpreting it down to the library routines that implement the quarks. The user is responsible for supplying the C/C++ code that implements any control and interrupt quarks used. All hardware-dependent code resides within the quarks so that an MDLe plan can accomplish the same task when executed on kinetic state machines whose dynamics and transducer-suites differ to a certain degree.

The MDLe Engine handles all CPU scheduling of atoms as well as the sharing of CPU time by a control quark and the corresponding interrupt(s) that must be concurrently evaluated. Furthermore, atoms with high computational needs can spawn additional threads to facilitate the processing of sensor data. A typical example is that of a control law that servos on quantities extracted from an incoming sound or video stream. This software runs on the Linux operating system and has been exercised extensively over the last three years. In the following, we give an example of the kinds of experiments that are facilitated by the current MDLe implementation.

As an example, we consider the motion control task performed by a mobile robot that is driven using MDLe strings. The robot used in the experiment is a Nomadic Technologies Super Scout II. It is differentially driven and outfitted with an array of ultrasonic and tactile sensors, as well as an internal odometer which keeps track of the robot's position and orientation with respect to an initial reference frame. The robot accepts two inputs, namely forward and angular velocity, denoted by u_f and u_θ.

An example of useful interrupt functions is given below:

- `(bumper)`: returns 1 when the robot's bumper tape detects contact, 0 otherwise.
- `(atIsection` b`)`, where b is a 4-bit binary number: returns 1 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.
- `(wait` τ`)`: returns 1 if τ seconds have passed after an atom has begun to run, 0 otherwise.

The plans for the experiments described in this section were built from the following set of atoms; (Their syntax is: `(Atom (interrupt condition) (control law))`.)

- `(Atom (wait` ∞`) rotate (`α`))`: $u_f = 0,\ u_\theta = k(\alpha - \theta)$. Causes the robot to make its orientation equal to α with respect to its current coordinate system.
- `(Atom (bumper AND 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 in `location`. If there are objects close to the robot along 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.

To illustrate the use of these atoms, three landmarks (specified through b in the previous paragraph) were created in order to allow the robot to safely navigate between all three locations in a repeatable manner. The control inputs that steer the robot between these landmarks was encoded as a MDLe string. Two of these MDLe programs, namely one to steer the robot from the rear of the lab (Lab 2) to the front of the lab (Lab 1), Γ_{lab2}^{lab1}, and one to steer the robot from the front of the lab to the office, Γ_{lab1}^{office}, are shown below.

Γ_{lab2}^{lab1} = {

```
 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))
}
```

Γ_{lab}^{office} = {

```
 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));
}
```

VI. Conclusions

Motion description languages are gaining attention as tools for understanding multi-modal control systems. The root of such languages may be traced to the early days of machine tool languages. Connections between differential equation-based control, planning, and logic can be placed

on solid ground by formalizing a necessary lowest level of abstraction between these two modalities, and we propose to use MDLe for that purpose. In particular, we show that MDLe is a context free but not a regular language, suggesting the proper setting for interesting computational questions, as well as enables access to the vast literature on languages and computational complexity. In particular, knowing that MDLe is a context-free language is equivalent to knowing that it can be defined by a push-down automaton, i.e. a finite automaton augmented with an infinite capacity stack. This in turn implies that the complexities associated with various decision properties (such as emptiness, membership, etc.) are well-studied and understood.

A comparison is furthermore made between MDLe and a particular type of hybrid automaton for understanding the expressiveness of the language. In this connection, the MDLe software engine is presented. This package provides an environment in which multi-modal control procedures can be implemented as MDLe strings directly.

VII. Acknowledgments

The authors would like to thank S. Andersson, P. Sodre, and F. Zhang for their contributions to the collaboration [13], and for sharing freely in the discussions leading up to the ideas in the present paper.

VIII. References

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

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

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

[4] R. W. Brockett. Hybrid models for motion control systems. In H. Trentelman and J.C. Willems, editors, *Perspectives in Control*, pages 29–54. Birkhäuser, Boston, 1993.

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

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

[7] P. E. Caines and Y-J Wei. Hierarchical hybrid systems: A lattice theoretic formulation. *IEEE Trans. on Automatic Control*, 43(4):1–8, Apr. 1998.

[8] M. Egerstedt. Some complexity aspects of the control of mobile robots. In *Proc. of the American Control Conf.*, May 2002.

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

[10] M. Egerstedt and D. Hristu-Varsakelis. Observability and policy optimization for mobile robots. In *Proc. of the 41st IEEE Conf. on Decision and Control*, pages 3596–3601, Dec. 2002.

[11] M. Egerstedt and C.F. Martin. Conflict resolution for autonomous vehicles: A case study in hierarchical control design. *Int'l Journal of Hybrid Systems*, to appear, 2003.

[12] T. A. Henzinger. The theory of hybrid automata. In *Proc. of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292,. IEEE Computer Society Press, 1996.

[13] D. Hristu-Varsakelis, S. Andersson, F. Zhang, P. Sodre, and P.S. Krishnaprasad. A motion description language for hybrid system programming. *submitted to IEEE Trans. Robotics and Automation*.

[14] D. Hristu-Varsakelis and R. W. Brockett. Experimenting with hybrid control. *IEEE Control Systems Magazine*, 22(1):82–95, Feb. 2002.

[15] D. Hristu-Varsakelis, P.S. Krishnaprasad, S. Andersson, F. Zhang, P. Sodre, and L. D'Anna. The MDLe engine: a software tool for hybrid motion control. Technical Report TR2000-54, Institute for Systems Research, Oct. 2000.

[16] R. Motwani J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, Reading, MA, 2000.

[17] E. Lemch and P. E. Caines. Hierarchical hybrid systems; partition deformations and applications to the acrobot system. In T. A. Henzinger and S. Sastry, editors, *Int'l Workshop on Hybrid Systems: Computation and Control*, LNCIS 1386, pages 237–252, NYC, 1998. Springer.

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

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

[20] 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.

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

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

[23] G. Pappas and S. Simic. Consistent hierarchies of nonlinear abstractions. In *Proc. of 39th IEEE Conf. on Decision and Control*, Dec. 2000.

[24] G.J. Pappas, G. Lafferriere, and S. Sastry. Hierarchically consistent control systems. *IEEE Trans. on Automatic Control*, 45(6):1144–1160, June 2000.