NPS-MV-11-002

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

A Taxonomy of Turing Machines and Mission Execution Automata (MEA) with Lisp/Prolog Implementation

by

Robert McGhee, Don Brutzman, and Duane Davis

25 July 2011

This report was prepared by:

R. B. McGhee
Emeritus Professor

D. P. Brutzman
Associate Professor

D. T. Davis
Assistant Professor

REPORT DOCUMENTATION PAGE		*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**		
1. REPORT DATE *(DD-MM-YYYY)* 25-07-2011	**2. REPORT TYPE** NPS Technical Report	**3. DATES COVERED** *(From - To)* 01 Jan–30 Jun 2011
4. TITLE AND SUBTITLE A Taxonomy of Turing Machines and Mission Execution Automata (MEA) with Lisp/Prolog Implementation		**5a. CONTRACT NUMBER** N/A
		5b. GRANT NUMBER N/A
		5c. PROGRAM ELEMENT NUMBER N/A
6. AUTHOR(S) R. B. McGhee, D. P. Brutzman and D. T. Davis		**5d. PROJECT NUMBER** N/A
		5e. TASK NUMBER N/A
		5f. WORK UNIT NUMBER N/A
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Navy Modeling and Simulation Office (NMSO) 1333 Isaac Hull Ave. Stop 5012 Washington Navy Yard, D.C. 20376-5012		**10. SPONSOR ACRONYM** NMSO
		11. SPONSOR REPORT NUMBER N/A
12. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release: distribution is unlimited.		
13. SUPPLEMENTARY NOTES:The views expressed in this report are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.		

14. ABSTRACT In ongoing research relating to computer control of unmanned untethered submersible vehicles (UUVs), the authors have been guided by our knowledge of the way task abstraction and mission execution are accomplished in manned submarines. This has led us to propose and investigate the “Rational Behavior Model” (RBM), a tri-level software architecture in which the top “strategic” level encompasses the functioning of a submarine commander carrying out formal written mission orders. Below this level, a “tactical” level of software decomposes high level commands into real-time “execution” level commands to the sensors and actuators of the UUV.

At sea experiments with two UUVs have demonstrated RBM utility, but we have been frustrated by the difficulty of expressing strategic level orders understandable to non-programmer mission specialists. We have concluded that this can be achieved by defining a new mathematical abstraction, a “Mission Execution Automaton” (MEA), as a generalization of a “Turing Machine” (TM). Specifically, a TM consists of a Finite State Machine (FSM) and a potentially infinite memory in the form of an “incremental tape recorder”. The MEA generalization recognizes the tape recorder as an “external agent” and allows for a human being or a sensor-based robot agent. This takes a TM “out of its box” and provides situational awareness engendering an ability to carry out real-time missions in the physical world.

We use the Prolog “logic programming” language to realize an MEA and use this realization to theoretically and experimentally demonstrate the MEA abstraction’s power. Examples, with all Prolog code, include a representative manned submarine mission and a second mission that realizes a universal Turing Machine to prove Turing completeness. Because the MEA is based on formal mathematical logic, we can demonstrate “proof of correctness” for any mission consisting of a finite number of phases with defined and finite phase transition and mission completion conditions. We believe this to be fundamentally important for both military and civilian applications.

15. SUBJECT TERMS
AUV Control, Rational Behavior Model, Turing Machine, Prolog

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NBR OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Don Brutzman
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UL	35	**19b. TELEPHONE NUMBER** *(include area code)* 831-656-2149

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

ABSTRACT

The authors have been involved in research relating to computer control of unmanned untethered submersible vehicles (UUVs) for some time. In this work, we have been guided in our efforts by our knowledge of the way task abstraction and mission execution are accomplished in manned submarines. This has lead us to propose and investigate a tri-level software architecture called the “Rational Behavior Model” (RBM) in which the top “strategic” level of code encompasses the functioning of a human submarine commander in carrying out formal written mission orders. Below this level, a “tactical” level of software decomposes high level commands from the strategic level into real-time “execution” level commands to the sensors and actuators of the UUV.

While we have been successful in demonstrating the utility of RBM in at-sea experiments with two UUVs, we have been frustrated by the difficulty of finding a means of expressing mission orders at the strategic level in a way that can be understood by mission specialists who are not programmers. We have come to the conclusion that this goal can best be achieved by defining a new mathematical abstraction which we call a Mission Execution Automaton (MEA). An MEA is a generalization of the previously defined notion of a Turing Machine (TM), which in turn serves as a general model for computation. Specifically, a Turing Machine consists of a Finite State Machine (FSM), provided with a potentially infinite memory in the form of an “incremental tape recorder.” The MEA generalization recognizes the tape recorder as an “external agent” of the FSM, and allows for the possibility that such an agent could alternatively be a human being or a sensor-based robot. This generalization takes a TM “out of its box”, and provides it with situational awareness, thereby engendering an ability to carry out real-time missions in the physical world.

In this report, we show how to realize MEAs using the Prolog logic-programming language. With this realization, we have demonstrated the power of the MEA abstraction by both theoretical and experimental means. The report contains two detailed examples along with all Prolog code used for mission specification and execution. One of these missions is a representative manned submarine mission. The second realizes a universal

Turing Machine, thereby proving the Turing completeness of MEAs. Because our MEA model is based on formal mathematical logic, we are able to demonstrate “proof of correctness” using Prolog code for any mission consisting of a finite number of phases with defined and finite phase transition and mission completion conditions. We believe this to be of fundamental importance for military missions, and perhaps as well for some classes of civilian applications of UUVs.

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A *Turing Machine* (TM) is a *Finite State Machine* (FSM) with an associated one-dimensional infinite storage medium called a *tape* [1]. In the original formulation of such machines [2], the FSM is associated with a *tape head* capable of moving over the tape in either direction and reading from it or writing to it. It is known that no digital computing machine can be more powerful than a TM [1, 2]. In this report, we first implement a *specific* TM in Common Lisp [3], and then show how this machine can be generalized into a *universal* TM [1, 2] by factoring out the *state table* of the specific machine. It is then shown how a universal TM can be used to return a value for any *computable function* by specializing its state table to that function [1, 2].

In following sections of this report, the problem chosen to illustrate a TM is the *zero-crossing detector* problem associated with a random sequence of contiguous zeros and ones. Specifically, if such a sequence, of any length, is written on the input tape of a *zero-crossing detector* Turing Machine, then, beginning on the left side of the sequence, such a TM can keep a running count of the difference of the number of ones encountered on the tape to that point, and the number of zeros encountered. When this difference is equal to zero, the machine stops and signals *success*. If this never happens, when the machine reaches the end of the original sequence, it stops and indicates *failure*. In usual TM parlance, if success is achieved, it is said that the machine “accepts the tape”. Otherwise, the tape is “rejected”.

In what follows, a zero-crossing detector is designed using a 12-row *primitive* state table, in which every row of the state table implements exactly one “primitive” action by the TM. After that, this state table is encoded in Common Lisp in *executable* form, and tested with four different tape sequences. Next, standard *state-reduction* techniques [4] are used to reduce the state table to just 4 rows. In order to facilitate testing of the reduced-row state table, the code of the zero-crossing detector is *factored* into separate functions needed by *every* TM from those *specific* to the 12-row state table, thereby achieving a *Universal Turing Machine* (UTM). The latter machine is then employed to realize the 4-row zero-crossing detector by reading in its state table before beginning execution. It is shown that, as it must, this machine returns the same value for the test functions as those returned by the 12-row machine, thus satisfying a necessary condition for the correctness of both the UTM code as well as the reduced state table and its code.

Having illustrated and coded the usual idea of a Turing Machine, a *proper super class* of such machines called a *mission execution automaton* (MEA) is defined. An MEA is generalization of a TM in that it can have any kind of *external agent*, not limited to a mechanical tape recorder. To facilitate this generalization, because of its power in expressing abstract definitions in executable form, *Allegro Prolog* [5, 6] with Lisp extensions is adopted as the initial programming language for MEAs. The first MEA developed here is intended for use with a human external agent, so it is called a *human interactive* MEA. A simple submarine reconnaissance mission is presented to illustrate the application of this form of MEA. Evidently, if a *sensor-based robot* [7] were to replace a human as the external agent for the given submarine mission, an autonomous *unmanned underwater vehicle* (UUV) would result [8]. A second MEA included in this report shows how an *incremental tape recorder* can replace a human as an external agent to accomplish Turing Machine missions, thereby demonstrating that MEA do in fact subsume Turing Machines.

II. A ZERO-CROSSING DETECTOR TURING MACHINE

A. DESIGN AND CODING

In general, despite their universal computing power, it is not easy to design (or program) Turing Machines. This is one reason that they are usually treated as mathematical entities, and not constructed or even emulated [1, 2], while more "user friendly" digital computers are used for practical computation. However, in what follows, to illustrate the TM concept, and to support the definition and testing of MEAs, emulated Turing Machines are implemented and realized.

In order to derive the state-transition table for a zero-crossing detector, the following general strategy will be adopted. First of all, an additional character, "X", will be introduced to be used for erasing 1's or 0's from the TM tape. In addition, the character "B" will be used to denote "blank" read-write cells on the tape [1, 2]. Using these conventions, the TM begins by reading and remembering the leftmost entry on the tape containing a sequence to be tested. An X is written in place of this character and the tape head then moves left and deposits the original character. The data on the tape is now one character longer than it was initially, with the X separating the first character from the rest of the sequence on the tape. Next, the head moves right, passing over the X, and picks up and erases the next character, if any. If none, there is no possibility of a zero-crossing, and the tape is rejected. If there is a character, the head picks it up, writes an X, and then moves left past all X's (only two at this point). When the first 1 or 0 is encountered, if the tape is carrying the same symbol, it moves one more position left and deposits the symbol. In this case, there are now either two 1's or two 0's to the left of two X's. If the carried symbol is not the same as the symbol under the read head, then a zero-crossing has been detected and the tape is accepted. By repeating this process, the excess of 0's or 1's is accumulated to the left of all the X's and the cycle continues until either a zero-crossing occurs (tape accepted), or all of the original characters have been processed (tape rejected).

Figure 1 below shows a primitive state table encoding this algorithm. In this table, an entry of "d" indicates a "don't care" or *impossible* condition. The three-tuple entries (*actions*) are: next state (next row), symbol to be written, and head movement (left or right) respectively.

Present State	Input = 0	Input = 1	Input = X	Input = B	Description
1	2, X, L	3, X, L	d	d	Start
2	2, 0, L	4, X, L	2, X, L	5, 0, R	Carry 0 left
3	6, X, L	3, 1, L	3, X, L	7, 1, R	Carry 1 left
4	d	8, 1, R	d	11, B, R	Just cancelled a 1
5	9, 0, R	d	10, X, R	d	Excess 0 deposited
6	9, 0, R	d	d	11, B, R	Just cancelled a 0
7	d	8, 1, R	10, X, R	d	Excess 1 deposited
8	d	8, 1, R	10, X, R	d	Cross 1's and X's
9	9, 0, R	d	10, X, R	d	Cross 0's and X's
10	2, X, R	3, X, R	10, X, R	12, B, R	Cross over X's
11	d	d	d	Accept	Halt
12	d	d	d	Reject	Halt

Figure 1. Primitive State-Transition Table for Zero-Crossing Detector

To understand the functioning of the above state table, it is necessary to associate a well defined purpose with each row. This is generally quite difficult, and the comments above actually just hint at this information. As an aid to understanding the algorithm, the reader is invited to develop a more comprehensive natural language statement for the purpose of each row. To assist in such an effort, and to furnish a means for processing an arbitrary binary data tape, the Common Lisp code in the following Figure 2 is provided.

```
;C:/Documents and Settings/mcghee/My Documents/Mission Control/Turing Machine/
;moving-head-12-state-zero-crossing-turing-machine.cl

;Written in Allegro ANSI Common Lisp, Version 8.2, by Prof. Robert B. McGhee
;(robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey, CA.
```

```
;Date of last revision: 30 January 2011.

;Basic problem to be solved below is for a Turing Machine to be furnished with a doubly
;infinite tape containing one sequence of contiguous 1's and 0's, and otherwise blank.
;The machine is started with its moveable tape head located over the leftmost non-blank
;tape cell. It then carries out a computation determined by a given state table designed
;to ensure that the machine stops only after it has processed the entire tape or else has
;encountered an equal number of 1's and 0's on the tape. This latter event is called a
;"zero crossing" event because a running count of the difference between the number of
;1's encountered and the number of 0's encountered crosses zero at this point in the
;processing of the initial tape.

;What follows makes use of a "primitive" state table in which every row of the table
;accomplishes exactly one action (no combined purposes for any given row). "Don't care"
;entries in the state table are simply left out of the "action" function definition.
;State reduction techniques for finite state machines can reduce this machine to four
;states. However, it is much more difficult to provide a word description of the function
;of each row in the reduced row machine.

;In the code below, the Lisp reserved word "nil" is used to represent a "blank" symbol
;either read from or written to the tape. The symbol "X" is used to overwrite either a 1
;or a 0 on the tape. The case "otherwise" denotes action to be taken when tape head reads
;a "blank". This machine functions by picking up the first symbol on the original tape,
;writing an "X" in its place, and then depositing the original symbol to the left of the
;X. It then goes back for another symbol (if any), carries it back over the X, and either
;erases the first symbol or passes over it and deposits the second symbol. Erasure means
;that the first and second symbols were different and therefore a zero crossing has been
;found and processing ends with success. No erasure means that there are now two of the
;first symbol deposited to the left of the two X's indicating an excess of two of the
;first symbol. Process continues until there are only blanks to the left of the row of
;X's, signifying success, or else there are no more of the original characters to the
;right of the row of 1's, signifying failure. Details provided by further comments below.

;The tape is initially positioned so that the recorder read-write head is on the leftmost
;non-blank cell of the tape. After that, the head moves left or right according to entry
;in state table. A moving head (rather than a moving tape) was chosen for this code to
;make it easier for a human to think of the tape head as a little "robot" crawling over a
;one-dimensional tape, and carrying memory and mission logic with it. This makes it
;easier to draw the tape as stationary, but with changing contents during manual
;simulation of tape processing.


(defclass zero-crossing-detector ()
  ((recorder :accessor recorder :initform (make-instance 'incremental-tape-recorder))
   (current-action :accessor current-action)  ;State table 3-tuple.
   (current-state :accessor current-state)))  ;State table row.

(defclass incremental-tape-recorder ()
  ((tape :accessor tape :initform (make-instance 'doubly-infinite-tape))))

(defclass doubly-infinite-tape ()
  ((middle-cell-contents :accessor middle-cell-contents :initform nil)
   (right-side :accessor right-side :initform nil)
   (left-side :accessor left-side :initform nil)))

(defmethod read-tape ((recorder incremental-tape-recorder))
  (middle-cell-contents (tape recorder)))

(defmethod cycle ((machine zero-crossing-detector)) ;Returns nil if current-action is nil
  (let* ((current-state-table-entry (current-action machine))
         (next-state (first current-state-table-entry))
         (write-symbol (second current-state-table-entry))
         (movement (third current-state-table-entry)))
        (when current-state-table-entry
          (write-tape-and-move-head (recorder machine) write-symbol movement)
          (setf (current-state machine) next-state
                (current-action machine) (read-state-table machine)))))

(defmethod cyclen ((machine zero-crossing-detector) N)
  (dotimes (i N) (cycle machine))
  (print-total-state machine)
  (print-current-tape-processing-state))

(defmethod write-tape-and-move-head
          ((recorder incremental-tape-recorder) write-symbol movement)
  (case movement
    ('L (and (push write-symbol (right-side (tape recorder)))
             (setf (middle-cell-contents (tape recorder))
                   (pop (left-side (tape recorder))))))
    ('R (and (push write-symbol (left-side (tape recorder)))
             (setf (middle-cell-contents (tape recorder))
                   (pop (right-side (tape recorder))))))))

(defmethod tape-state ((machine zero-crossing-detector))
  ;Character "H" brackets symbol currently under tape-head
  (append (reverse (left-side (tape (recorder machine)))) '(h)
          (list (middle-cell-contents (tape (recorder machine))))
          '(h) (right-side (tape (recorder machine)))))

(defmethod print-total-state ((machine zero-crossing-detector))
  (pprint (list (current-state machine) (middle-cell-contents (tape (recorder machine)))
                (current-action machine) (tape-state machine))))

(defmethod initialize ((machine zero-crossing-detector) start-state initial-tape-list)
  (setf (current-state machine) start-state
        (middle-cell-contents (tape (recorder machine)))
        (first initial-tape-list) (right-side (tape (recorder machine)))
                 (rest initial-tape-list)
        (left-side (tape (recorder machine))) nil
        (current-action machine) (read-row-1 machine)))

(defconstant sequence-1 '(1))
(defconstant sequence-2 '(0 0 0 1 1))
(defconstant sequence-3 '(0 0 1 1 0))
(defconstant sequence-4 '(1 1 1 0 1 0 0 1 0 0 0))

(defmethod read-state-table ((machine zero-crossing-detector))
  (case (current-state machine)
        (1 (read-row-1 machine)) (2 (read-row-2 machine)) (3 (read-row-3 machine))
        (4 (read-row-4 machine)) (5 (read-row-5 machine)) (6 (read-row-6 machine))
        (7 (read-row-7 machine)) (8 (read-row-8 machine)) (9 (read-row-9 machine))
        (10 (read-row-10 machine)) (11 (read-row-11 machine))
        (12 (read-row-12 machine))))

(defmethod read-row-1 ((machine zero-crossing-detector))
  ;Start. Cancel symbol under tape head and move left while remembering symbol.
  (case (read-tape (recorder machine))
        (0 '(2 X L)) (1 '(3 X L))))

(defmethod read-row-2 ((machine zero-crossing-detector)) ;Carry 0 left over 0's and X's.
  ; If 1 encountered, cancel it.  If blank encountered, write 0
  (case (read-tape (recorder machine))
        (0 '(2 0 L)) (1 '(4 X L)) (X '(2 X L)) (otherwise '(5 0 R))))

(defmethod read-row-3 ((machine zero-crossing-detector))
  ;Carrying left over 1s and Xs--if 0 encountered cancel it, if blank encountered write 1
  (case (read-tape (recorder machine))
        (0 '(6 X L)) (1 '(3 1 L)) (X '(3 X L)) (otherwise '(7 1 R))))

(defmethod read-row-4 ((machine zero-crossing-detector))
  ;Just cancelled a 1 and moved left—unless 1 is under head, done.
  (case (read-tape (recorder machine))
        (1 '(8 1 R)) (otherwise '(11 nil R))))

(defmethod read-row-5 ((machine zero-crossing-detector))
  ;Just deposited excess 0 and moved right
  (case (read-tape (recorder machine))
        (0 '(9 0 R)) (X '(10 X R))))

(defmethod read-row-6 ((machine zero-crossing-detector))
  ;Just cancelled a 0 and moved left—unless 0 is under head, done.
  (case (read-tape (recorder machine))
        (0 '(9 0 R)) (otherwise '(11 nil R))))

(defmethod read-row-7 ((machine zero-crossing-detector))
  ;Just deposited excess 1 and moved right
  (case (read-tape (recorder machine))
        (1 '(8 1 R)) (X '(10 X R))))
```

```
(defmethod read-row-8 ((machine zero-crossing-detector))
  ;Crossing over excess 1's while moving right
  (case (read-tape (recorder machine))
    (1 '(8 1 R)) (X '(10 X R))))

(defmethod read-row-9 ((machine zero-crossing-detector))
  ;Crossing over excess 0's while moving right.
  (case (read-tape (recorder machine))
        (0 '(9 0 R)) (X '(10 X R))))

(defmethod read-row-10 ((machine zero-crossing-detector))
  ;Crossing over X's while moving right—done if no more 1's or 0's.
  (case (read-tape (recorder machine))
    (0 '(2 X L)) (1 '(3 X L)) (X '(10 X R)) (otherwise '(12 nil R))))

(defmethod read-row-11 ((machine zero-crossing-detector)) nil) ;Halt and accept.

(defmethod read-row-12 ((machine zero-crossing-detector)) nil) ;Halt and reject.

(defun start (data-tape-list)
  (setf M1 (make-instance 'zero-crossing-detector))
  (initialize M1 1 data-tape-list)
  (print-total-state M1))

(defun print-current-tape-processing-state ()
  (cond
    ((equal (current-state M1) 11) (princ " ACCEPT"))
    ((equal (current-state M1) 12) (princ " REJECT"))
    (t (princ " INCOMPLETE"))))


(defun test1 () (start sequence-1) (cyclen M1 20))

(defun test2 () (start sequence-2) (cyclen M1 100))

(defun test3 () (start sequence-3) (cyclen M1 100))

(defun test4 () (start sequence-4) (cyclen M1 100))

(defun test5 () (start sequence-4) (cyclen M1 1000))

(defun ct () (cycle M1) (print-total-state M1))

(defun verbose-test1 () (start sequence-1) (dotimes (i 4) (ct))
                        (print-current-tape-processing-state) 'done)

(defun test () (test1) (test2) (test3) (test4) (test5) 'DONE)
```

Figure 2. Lisp Code for a 12-State Zero-Crossing Turing Machine

As can be seen, the above code defines three top-level object classes: a zero-crossing detector, an incremental tape recorder, and a doubly infinite tape. Methods are then defined for the various functions needed by any Turing Machine. After that, a state table for the 12-state zero-crossing detector is encoded along with associated utility functions for creating and testing an instance of such a machine. It should be noted that the *read-state-table* function provides extended comments on the role of each state in the table in carrying out the desired search for a zero-crossing. This should be helpful to readers in further understanding the state table of Figure 1. The *verbose-test1* function provides an exhaustive trace of the evaluation of sequence-1. A similar function could be written for other sequences.

The authors believe that the details of the above code are self explanatory for readers acquainted with Common Lisp, and not understandable at all otherwise. Since this report is intended for reading by computer scientists or others literate in Lisp, our apologies to all others. The authors are not currently expressing the definition of the zero-crossing detector in *executable* form in any languages other than Lisp and Prolog.

At this point we must admit that we do not have a *formal* proof that either the state table or the Lisp code above presents a correct solution to the zero-crossing problem. Instead, all that is available is the verbal argument presented above. On the other hand, it should be recognized that until the development of the algebra of *first-order predicate calculus* beginning in 1879 [9, 10], such a natural-language proof was all that was available in any branch of mathematics. In fact, it is an astonishing fact that a complete and consistent axiomatization of plane geometry was not available until 2000! Until then, from the point of view of contemporary abstract mathematics, all proofs of Euclid's famous theorems were *experimental* and *physical* in nature.

In the absence of a formal proof, all that is available to further verify the table of Figure 1 and the code of Figure 2, is that it must pass the necessary condition of giving a correct answer for any specific example presented to it. Figure 2 provides five such examples. Figure 3 below shows the results of these tests. It is the authors' opinion that these results are all correct.

```
International Allegro CL Free Express Edition
8.2 [Windows] (Jan 25, 2010 15:08)
Copyright (C) 1985-2010, Franz Inc., Oakland, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   Allegro CL 8.2 Express user

CG version 1.134 / IDE version 1.125
Loaded options from C:\Documents and Settings\mcghee\My Documents\allegro-prefs-8-2-express.cl.

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current optimization settings
;; evaluate (EXPLAIN-COMPILER-SETTINGS).

[changing package from "COMMON-LISP-USER" to "COMMON-GRAPHICS-USER"]
CG-USER(1):
; Fast loading
;    C:\Documents and Settings\mcghee\My Documents\Mission Control\Turing Machines\moving-head-
12-state-zero-crossing-turing-machine.fasl

CG-USER(1): (test)

(1 1 (3 X L) (H 1 H))
(12 NIL NIL (1 X NIL H NIL H)) REJECT
(1 0 (2 X L) (H 0 H 0 0 1 1))
(12 NIL NIL (0 X X X X X X X NIL H NIL H)) REJECT
(1 0 (2 X L) (H 0 H 0 1 1 0))
(11 X NIL (NIL H X H X X X X X 0)) ACCEPT
(1 1 (3 X L) (H 1 H 1 1 0 1 0 0 1 0 0 0))
(7 1 (8 1 R) (1 H 1 H X X X X X X X X X X X 0 0 0)) INCOMPLETE
(1 1 (3 X L) (H 1 H 1 1 0 1 0 0 1 0 0 0))
(11 X NIL (NIL H X H X X X X X X X X X X X X X X 0)) ACCEPT
DONE
CG-USER(2):
```

Figure 3. Test Function Results for Zero-Crossing Detection Turing Machine

B. STATE MINIMIZATION

Having some confidence that the code of Figure 2 is correct, it is now possible to apply common state reduction techniques to the primitive state table of Figure 1 in order to obtain a *minimal state* machine [4] that is functionally equivalent and much smaller in size. The results of this step are presented in Figure 4 below:

Present State	Input = 0	Input = 1	Input = X	Input = B
1	2, X, L	3, X, L	1, X, R	Reject
2	2, 0, L	4, X, L	2, X, L	4, 0, R
3	4, X, L	3, 1, L	3, X, L	4, 1, R
4	4, 0, R	4, 1, R	1, X, R	Accept

Figure 4. State Table for Four-State Zero-Crossing Turing Machine

The above table is used to specialize and test the "universal" Turing Machine developed in the next section of this memorandum. It should be noted that it is no longer possible to associate a straightforward function with to each row of this table. That is, in some sense, Figure 1 constitutes "source" code for a specific Turing Machine state table, while Figure 4 shows the corresponding condensed "compiled" code.

C. A UNIVERSAL TURING MACHINE (UTM)

Having reduced the state table to four states, it is now possible to test the reduced machine with the five test functions defined in Figure 2. However, rather than starting from scratch in encoding the reduced machine, it is more useful to first factor out the state table and associated functions from the code of Figure 2 to obtain a *universal* Turing Machine (UTM) [1, 2]. The results of this step are presented below as Figure 5. It should be noted that this code includes a Lisp *load* function call to provide a specific state table to the UTM.

```
;C:/Documents and Settings/mcghee/My Documents/Mission Control/Turing Machine/
;universal-turing-machine.cl

;This code was written in Allegro ANSI Common Lisp, Version 8.2, by Robert B. McGhee
;(robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey,CA.
;Date of last revision: 28 January 2011.

;This code defines a universal turing machine class along with its components and
;methods. Execution requires that it be loaded along with a specific state table and supporting
;supporting function definitions. Examples can be found in the folder
;"~/Turing Machine/State Table Archive".

;The tape is initially positioned so that the recorder read-write head is on the leftmost
;non-blank cell of the tape. After that the tape moves left or right according to entry
;in state table. A moving tape (rather than a moving head) was chosen for this code to
;make it easier for a human to emulate the recorder by using two "stacks" of paper with
;one sheet in between them to achieve the functionality of a doubly-infinite tape.

;The functions "read-state-table", "read-row-1", and "print-current-tape-processing-
;state" must be defined in "state-table" code per examples.


(load "C:/Documents and Settings/mcghee/My Documents/Mission Control/Turing Machines/state-
table.fasl")

(defclass universal-turing-machine ()
  ((recorder :accessor recorder :initform (make-instance 'incremental-tape-recorder))
   (current-action :accessor current-action)  ;State table 3-tuple.
   (current-state :accessor current-state)))  ;State table row.

(defclass incremental-tape-recorder ()
  ((tape :accessor tape :initform (make-instance 'doubly-infinite-tape))))

(defclass doubly-infinite-tape ()
  ((middle-cell-contents :accessor middle-cell-contents :initform nil)
   (right-side :accessor right-side :initform nil)
   (left-side :accessor left-side :initform nil)))

(defmethod read-tape ((recorder incremental-tape-recorder))
  (middle-cell-contents (tape recorder)))

(defmethod cycle ((machine universal-turing-machine))
  ;Returns nil if current-action is nil
  (let* ((current-state-table-entry (current-action machine))
         (next-state (first current-state-table-entry))
         (write-symbol (second current-state-table-entry))
         (movement (third current-state-table-entry)))
        (when current-state-table-entry
          (write-and-move-tape (recorder machine) write-symbol movement)
          (setf (current-state machine) next-state
                (current-action machine) (read-state-table machine)))))
```

```
(defmethod cyclen ((machine universal-turing-machine) N)
  (dotimes (i N) (cycle machine))
  (print-total-state machine)
  (print-current-tape-processing-state))

(defmethod write-and-move-tape
    ((recorder incremental-tape-recorder) write-symbol movement)
  (case movement
    ('R (and (push write-symbol (right-side (tape recorder)))
             (setf (middle-cell-contents (tape recorder))
                   (pop (left-side (tape recorder))))))
    ('L (and (push write-symbol (left-side (tape recorder)))
             (setf (middle-cell-contents (tape recorder))
                   (pop (right-side (tape recorder))))))))

(defmethod tape-state ((machine universal-turing-machine))
;Character "H" brackets symbol currently under tape-head
  (append (reverse (left-side (tape (recorder machine)))) '(h)
          (list (middle-cell-contents (tape (recorder machine))))
          '(h) (right-side (tape (recorder machine)))))

(defmethod print-total-state ((machine universal-turing-machine))
  (pprint (list (current-state machine) (middle-cell-contents (tape (recorder machine)))
                (current-action machine) (tape-state machine))))

(defmethod initialize ((machine universal-turing-machine) start-state initial-tape-list)
  (setf (current-state machine) start-state
        (middle-cell-contents (tape (recorder machine)))
        (first initial-tape-list) (right-side (tape (recorder machine)))
        (rest initial-tape-list)
        (left-side (tape (recorder machine))) nil
(current-action machine) (read-row-1 machine)))
```

Figure 5. Lisp Code for a Universal Turing Machine

Per comments on this figure, it should be recognized that the universal Turing Machine defined here utilizes a *moving tape* rather than a moving tape head. As stated in these comments, this was done to facilitate later factoring out of the tape recorder to allow for the possibility of either a human being or a sensor-based robot fulfilling this function as an external agent for a mission execution automaton.

Having provided the UTM code of Figure 5 above, it remains to convert the state table of Figure 4 to executable form. This is done in Figure 6 below. In comparing this figure to the table of Figure 4, it is important to realize that all “L” movements have been changed to “R”, while all “R” have become “L”. This is because, as stated above, the UTM has been encoded as a moving tape Turing Machine, rather than a moving head machine as was assumed for Figures 1 and 4.

```
;C:/Documents and Settings/mcghee/My Documents/Mission Control/Turing Machine/
;State Table Archive/4-state-zero-crossing.cl

;This code was written in Allegro ANSI Common Lisp, Version 8.2, by Robert B. McGhee
;(robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey, CA.
;Date of last revision: 28 January 2011.

;Basic problem to be solved below is for a Turing Machine to be furnished with a doubly
;infinite tape containing one sequence of contiguous 1's and 0's, and otherwise blank.
;The machine is started with its moveable tape installed so that the tape read-write head
;is over the leftmost non-blank tape cell. It then carries out a computation determined
;by a given state table designed to ensure that the machine stops only after it has
;processed the entire tape or else has encountered an equal number of 1's and 0's on the
;tape.  This latter condition is called a "zero crossing" event because a running count
;of the difference between the number of 1's encountered and the number of 0's
;encountered reaches zero at this point in the processing of the initial tape.

;The code listed below is obtained by applying standard state reduction techniques to the
;"12-state-zero-crossing" state table listed in the above "Turing Machine" directory. It
;is much harder to understand since state reduction means that, unlike the primitive
;state-table of the 12-state machine, every state now performs multiple functions.

;In the code below, the Lisp reserved word "nil" is used to represent a "blank" symbol
;either read from or written to the tape. The symbol "X" is used to overwrite either a 1
;or a 0 on the tape. The case "otherwise" denotes action to be taken when tape head reads
;a "blank". This machine functions by picking up the first symbol on the original tape,
;writing an "X" in its place, and then depositing the original symbol to the left of the
;X. It then goes back for another symbol (if any), carries it back over the X, and either
;erases the first symbol or passes over it and deposits the second symbol. Erasure means
;that the first and second symbols were different and therefore a zero crossing has been
;found and processing ends with success. No erasure means that there are now two of the
;first symbol deposited to the left of the two X's indicating an excess of two of the
;first symbol. Process continues until there are only blanks to the left of the row of
;X's, signifying success, or else there are no more of the original characters to the
;right of the row of 1's, signifying failure.

;To run this code, it must first be saved in "Turing Machine" directory as
;"state-table.cl" and then compiled. After that, "universal-turing-machine.cl" from same
;directory must be compiled and loaded. Test functions below can then be called to
;provide examples of code execution.


(defconstant sequence-1 '(1))
(defconstant sequence-2 '(0 0 0 1 1))
(defconstant sequence-3 '(0 0 1 1 0))
(defconstant sequence-4 '(1 1 1 0 1 0 0 1 0 0 0))

(defclass zero-crossing-detector (universal-turing-machine) ())

(defmethod read-state-table ((machine zero-crossing-detector))
  (case (current-state machine)
        (1 (read-row-1 machine)) (2 (read-row-2 machine)) (3 (read-row-3 machine))
        (4 (read-row-4 machine))))

(defmethod read-row-1 ((machine zero-crossing-detector))
  (case (read-tape (recorder machine))
        (0 '(2 X R)) (1 '(3 X R)) (X '(1 X L))))

(defmethod read-row-2 ((machine zero-crossing-detector))
  (case (read-tape (recorder machine))
        (0 '(2 0 R)) (1 '(4 X R)) (X '(2 X R)) (otherwise '(4 0 L))))

(defmethod read-row-3 ((machine zero-crossing-detector))
  (case (read-tape (recorder machine))
        (0 '(4 X R)) (1 '(3 1 R)) (X '(3 X R)) (otherwise '(4 1 L))))

(defmethod read-row-4 ((machine zero-crossing-detector))
  (case (read-tape (recorder machine))
        (0 '(4 0 L)) (1 '(4 1 L)) (X '(1 X L))))

(defun start (data-tape-list)
  (setf M1 (make-instance 'zero-crossing-detector))
  (initialize M1 1 data-tape-list)
  (print-total-state M1))
```

```
(defun print-current-tape-processing-state ()
  (cond
    ((and (equal (middle-cell-contents (tape (recorder M1))) nil)
          (equal (current-state M1) 4)) (princ " ACCEPT"))
    ((and (equal (middle-cell-contents (tape (recorder M1))) nil)
          (equal (current-state M1) 1)) (princ " REJECT"))
    (t (princ " INCOMPLETE"))))


(defun test1 () (start sequence-1) (cyclen M1 20))

(defun test2 () (start sequence-2) (cyclen M1 100))

(defun test3 () (start sequence-3) (cyclen M1 100))

(defun test4 () (start sequence-4) (cyclen M1 100))

(defun test5 () (start sequence-4) (cyclen M1 1000))

(defun ct () (cycle M1) (print-total-state M1))

(defun verbose-test1 () (start sequence-1) (dotimes (i 4) (ct))
                        (print-current-tape-processing-state) 'done)

(defun test () (test1) (test2) (test3) (test4) (test5) 'DONE)
```

Figure 6. Four-State Zero-Crossing Detector Lisp Code

Per comments above, copying the code of Figure 6 into the *state-table* function stored in the address indicated by the *load* function call of Figure 5, and then compiling, implements the specialization of the UTM to a zero-crossing detector when the UTM code is compiled. After this, calling the function *test* returns the same results as in Figure 3, again satisfying a necessary condition for the correctness of both state table and code.

THIS PAGE INTENTIONALLY LEFT BLANK

III. MISSION EXECUTION AUTOMATA (MEA)

A. A TURING MACHINE GENERALIZATION

A further useful abstraction of Turing Machines beyond a Universal Turing Machine is possible. Specifically, in all standard treatments of UTM, it is assumed that the FSM part of the machine has access only to an incremental tape recorder as its *external agent*. But this is very limiting. If Turing Machines are to come into the real world as actual realized systems, they will need access to other types of information. The generalization of a tape recorder to any type of external agent with a finite set of input and output symbols, and arbitrary motion capabilities, leads to the idea of a *mission execution automaton* (MEA). That is, a MEA is what is left of an *implemented* UTM after the tape recorder external agent is factored out. The function of the state table in specializing a UTM to a specific task is fulfilled by the *mission orders* for an MEA. Because of its powers of abstraction, and the relative ease with which it can be read *declaratively* [11], the Prolog *logic programming language* will be adopted in this report for MEA definition and implementation.

In the example to follow, a human being will be used as the external agent to the MEA. Complex missions to be carried out by human beings are typically specified in terms of a series of phases with predetermined phase transition rules and defined mission end conditions. For example, a simple five phase manned submarine reconnaissance mission might be phrased in structured natural language as follows:

Goal 1. Proceed to Area A and search the area. If the search is successful execute goal 2. If the search is unsuccessful, execute goal 3.

Goal 2. Obtain an environment sample from Area A. If the sample is obtained, execute goal 3. If the sample cannot be obtained, proceed to recovery position to complete the mission.

Goal 3. Proceed to Area B and search the area. Upon search success or failure, execute goal 4.

Goal 4. Proceed to Area C and rendezvous with UUV-2. Upon rendezvous success or failure, proceed to recovery position to complete the mission.

This mission will be used to illustrate both the capabilities and syntax of Prolog, and the design of an MEA capable of carrying out any similar mission when expressed as a series of phases written in Prolog as *mission orders*. Figure 7 below contains Prolog code for a human interactive multiphase MEA.

```
;C:/Documents and Settings/mcghee/My Documents/Mission Control/mission-controller.cl

;This code was written in Allegro ANSI Common Lisp, Version 8.2, by Prof.
;Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey,
;CA. Date of last revision: 4 February 2011.

;Allegro Prolog uses Lisp syntax. Rule head is first expression following "<--" symbol.
;Rule body is rest of expressions. Subsequent definitions of rule use "<-" symbol.

;Note that mission orders must be saved as "mission-orders.cl" in "Mission Control"
;folder, and then compiled before attempting execution by mission-controller. After
;compiling "mission-orders.cl", if "mission-controller.cl" has not been previously
;compiled it may be necessary to open it in a new Allegro Editor window to avoid "name
;conflict error" response from compiler.

(require :prolog) (shadowing-import '(prolog:==)) (use-package :prolog) ;Start Prolog.
(load "C:/Documents and Settings/mcghee/My Documents/Mission Control/mission-orders.fasl")


;Execution status (registers)

(<-- (current_phase 0)) ;Start phase.
(<-- (phase_exit_condition 0)) ;Selector for determining subsequent mission phase.


;Mission execution rule set

(<-- (execute_mission) (initialize_mission) (repeat) (execute_current_phase) (done) !)
(<-- (initialize_mission) (abolish current_phase 1) (asserta ((current_phase 1))))
(<-- (execute_current_phase) (abolish phase_exit_condition 1) (current_phase ?x)
                             (execute_phase ?x) (next_phase ?x) !)
(<-- (done) (current_phase 'mission_complete) (report "Mission succeeded"))
(<- (done) (current_phase 'mission_abort) (report "Mission failed"))


;Human external agent communication functions

(<-- (negative nil))
(<- (negative n))
(<-- (affirmative ?x) (not (negative ?x)))
(<-- (report ?C) (princ ?C) (princ ".") (nl))
(<-- (command ?C) (princ ?C) (princ "!") (nl))
(<-- (ask ?Q ?A) (princ ?Q) (princ "?") (read ?A))


;Test function (illustrates format for calling for mission execution from Lisp)

(defun tm () (?- (execute_mission)))
```

Figure 7. A Universal Human Interactive Multiphase Mission Execution Automaton

In reading the above code, following Lisp conventions, it is important to remember that a semicolon denotes that what follows is a *comment* intended to aid human code reading, and ignored by the Prolog compiler. Keeping this in mind, the two lines of this code constituting the MEA *registers* are Prolog *facts*, defining the *execution status* of a given mission. The facts state that the current mission phase is Phase 0 (Start phase), and that the current phase exit condition is also equal to 0.

Turning next to the *mission execution rule set*, Allegro Prolog [5, 6] syntax places a *rule head* immediately after a left arrow symbol. The *rule body* consists of all function calls listed after the rule head, and before the terminating parenthesis. Thus it can be seen that, in the context of the specified mission execution automaton, a mission is executed if it is initialized and successive phases are executed until done. The looping implied by this statement is achieved by the "repeat" function call. Specifically, *repeat* is a Prolog *system function* that always succeeds, but cannot be entered from the right (during backtracking). More precisely, referring to the general nature of Prolog code execution [6, 11], it can be seen that when the *done* predicate fails, Prolog *backtracks* and tries to find another value for *current_phase* by searching its fact database. This continues until "done" is satisfied by either mission completion or mission abort. Finally, the *execute_mission* predicate definition ends with a "!" symbol called a *cut*. The meaning of this symbol is that it stops backtracking by always succeeding when encountered during *forward* code execution (evaluation of successive predicates from left to right in a given rule body), but always failing on backtrack. In this particular case, use of a cut assures that when the test function *tm* is called, as intended, only *one* attempt to execute a mission will occur. Likewise, the cut associated with *execute_current_phase* assures that this predicate is entered only from the left.

Turning next to the other rules in Figure 7, it can be seen that the second definition of the predicate "done" uses a shorter arrow than the line above it. This is because, by Allegro Prolog convention, a long arrow *redefines* a predicate, replacing all prior definitions, while a short arrow signifies a *secondary* definition. Next, "initializing" a mission involves abolishing the "Start phase" and replacing it with "Phase 1". This asserts a convention of this MEA that all multiphase missions must begin with Phase 1. The predicate *abolish* is another Prolog system function (there are only approximately fifty such functions), that erases all occurrences of the named predicate, providing there is *at least one* such occurrence. It is because of this requirement that all facts are initialized. The syntax of the "abolish" function requires that the *arity* (number of variables in the definition) of a predicate selected to be erased from the Prolog database be specified (in this case the arity of *current_phase* is equal to 1). Finally, the *execute_current_phase* predicate definition introduces the *logic variable*, "?x". Logic variables are initially *unbound*, and values are found by Prolog by searching the fact database. Logic variables are uniquely signified in Allegro Prolog by the first character in the variable name being "?". Once a logic variable acquires a value, the *unification* feature of Prolog [6, 11] assures that all subsequent appearances of this variable in a given rule body will use the same value.

Following the mission execution rule set is another set of predicates called *human external agent communication functions.* It is this set of functions that gives the MEA a potential for *situational* awareness. It should be noted that this capability is obtained by means of communication with an external agent able to respond to a restricted and predefined set of commands, queries, and statements to and from the mission controller.

B. MISSION ORDERS AND RESULTS FOR SUBMARINE RECONNAISSANCE MISSION

The following Figure 8 presents mission orders for the above defined submarine reconnaissance mission:

```
;C:/Documents and Settings/mcghee/My Documents/Mission Control/Mission Orders Archive/
;AVCL-mission.cl"

;This code was written in Allegro ANSI Common Lisp, Version 8.2, by Prof.
;Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey,
;CA. Date of last revision: 4 February 2011.

;This code can be executed only if it is first saved in /My Documents/Mission Control/ as
;"mission_orders.cl" and then compiled. When this has been done, it can be executed by
;loading and compiling "mission_controller.cl", which is also located in
;/My Documents/Mission Control/.

;The "<--" predicate definition symbol should be used only for the first definition of a
;given predicate. After that, subsequent definitions must use "<-" to avoid overwrite.

(require :prolog) (shadowing-import '(prolog:==)) (use-package :prolog) ;Start Prolog.


;Mission specification

(<-- (execute_phase 1) (command "Search Area A") (phase_completed 1))
(<-- (phase_completed 1) (ask "Search successful" ?A) (affirmative ?A)
                         (asserta ((phase_exit_condition 1))))
(<- (phase_completed 1) (asserta ((phase_exit_condition 2))))
(<-- (next_phase 1) (phase_exit_condition 1) (retract ((current_phase 1)))
                    (asserta ((current_phase 2))))
(<- (next_phase 1) (retract ((current_phase 1))) (asserta ((current_phase 3))))
(<- (execute_phase 2) (command "Sample environment") (abolish phase_exit_condition 1)
                      (phase_completed 2))
(<- (phase_completed 2) (ask "Sample obtained" ?A) (affirmative ?A)
                        (asserta ((phase_exit_condition 1))))
(<- (phase_completed 2) (asserta ((phase_exit_condition 2))))
(<- (next_phase 2) (phase_exit_condition 1) (retract ((current_phase 2)))
                   (asserta ((current_phase 3))))
(<- (next_phase 2) (retract ((current_phase 2))) (asserta ((current_phase 5))))


(<- (execute_phase 3) (command "Search Area B") (phase_completed 3))
(<- (phase_completed 3) (ask "Search successful" ?A) (affirmative ?A)
                        (asserta ((phase_exit_condition 1))))
(<- (phase_completed 3) (asserta ((phase_exit_condition 2))))
(<- (next_phase 3) (phase_exit_condition 1) (retract ((current_phase 3)))
                   (asserta ((current_phase 4))))
(<- (next_phase 3) (retract ((current_phase 3))) (asserta ((current_phase 4))))


(<- (execute_phase 4) (command "Rendezvous UUV2") (phase_completed 4))
(<- (phase_completed 4) (ask "Rendezvous successful" ?A) (affirmative ?A)
                        (asserta ((phase_exit_condition 1))))
(<- (phase_completed 4) (asserta ((phase_exit_condition 2))))
(<- (next_phase 4) (phase_exit_condition 1) (retract ((current_phase 4)))
                   (asserta ((current_phase 5))))
(<- (next_phase 4) (retract ((current_phase 4))) (asserta ((current_phase 5))))


(<- (execute_phase 5) (command "Return to base") (phase_completed 5))
(<- (phase_completed 5) (ask "At base" ?A) (affirmative ?A)
                        (asserta ((phase_exit_condition 1))))
(<- (phase_completed 5) (asserta ((phase_exit_condition 2))))
(<- (next_phase 5) (phase_exit_condition 1) (retract ((current_phase 5)))
                   (asserta ((current_phase 'mission_complete))))
(<- (next_phase 5) (retract ((current_phase 5)))
                   (asserta ((current_phase 'mission_abort))))
```

Figure 8. Prolog Mission Orders for Submarine Reconnaissance Mission

The key difference between Figure 8 and the natural language mission definition is that the Prolog orders are simple enough to be read by the person providing the latter, and are yet *executable*. Figure 9 below presents the results of a partial test of the logic of the above mission orders.

```
International Allegro CL Free Express Edition
8.2 [Windows] (Jan 25, 2010 15:08)
Copyright (C) 1985-2010, Franz Inc., Oakland, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   Allegro CL 8.2 Express user

CG version 1.134 / IDE version 1.125
Loaded options from C:\Documents and Settings\mcghee\My Documents\allegro-prefs-8-2-express.cl.

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current optimization
;; settings evaluate (EXPLAIN-COMPILER-SETTINGS).

[changing package from "COMMON-LISP-USER" to "COMMON-GRAPHICS-USER"]
CG-USER(1):
; Fast loading
;    C:\Documents and Settings\mcghee\My Documents\Mission Control\human-interactive-mission-
controller.fasl
```

```
;   Fast loading C:\acl82express\code\PROLOG.fasl
;   Fast loading
;      C:\Documents and Settings\mcghee\My Documents\Mission Control\mission-orders.fasl

CG-USER(1): (tm)
Search Area A!
Search successful?n
Search Area B!
Search successful?n
Rendezvous UUV2!
Rendezvous successful?n
Return to base!
At base?n
Mission failed.
Yes

No.
CG-USER(2): (TM)
Search Area A!
Search successful?n
Search Area B!
Search successful?n
Rendezvous UUV2!
Rendezvous successful?n
Return to base!
At base?y
Mission succeeded.
Yes

No.
CG-USER(3): (TM)
Search Area A!
Search successful?y
Sample environment!
Sample obtained?n
Return to base!
At base?y
Mission succeeded.
Yes

No.
CG-USER(4): (TM)
Search Area A!
Search successful?y
Sample environment!
Sample obtained?n
Return to base!
At base?n
Mission failed.
Yes

No.
CG-USER(5): (TM)
Search Area A!
Search successful?y
Sample environment!
Sample obtained?y
Search Area B!
Search successful?y
Rendezvous UUV2!
Rendezvous successful?y
Return to base!
At base?y
Mission succeeded.
Yes

No.
CG-USER(6):
```

Figure 9. Partial Test Results for Submarine Reconnaissance Mission Execution

The results of Figure 9 constitute a kind of *Turing test* [12] that the MEA together with its mission orders behaves in exactly the same way as a human mission commander would in carrying out the natural language mission specification. It is the authors' opinion that these results are in agreement with the natural language definition of this mission. However, Figure 9 does not constitute an *exhaustive* test. Fortunately, since mission orders for an MEA define a FSM, exhaustive testing is possible, though tedious. The authors have completed such a test, and still believe that the specified mission has been correctly encoded. Note, however, that a dialogue can now be initiated between the person who coded this mission and the person who provided the natural language definition as to whether or not the desired mission logic has been captured. Once the correctness of the Prolog form of the mission orders has been agreed upon, it then becomes possible to replace the human external agent by a sensor-based robot. Evidently, this requires rewriting the external agent communication functions to suit the robot agent. Since this is vehicle specific, it is not done here. However, an example of such coding, using an earlier idea of an MEA, can be found in [13].

C. REDUCED STATE UNIVERSAL HUMAN INTERACTIVE MEA

Further consideration of the MEA defined in Figure 7 shows that some simplification is possible. Specifically, except for purposes of explanation, there is no reason to define the predicate *next_phase* as being separate from the *execute_phase* predicate. Instead, the functionality of the former can be absorbed into the latter. Moreover, when this is done, there is no longer a need for the *phase_exit_condition* fact. Since the "facts" of an MEA definition are analogous to the "state variables" of an FSM, this amounts to *state reduction* for an MEA. The results of this simplification are shown in Figure 10 below:

```
;C:/Documents and Settings/mcghee/My Documents/Mission Control/mission-controller.cl

;This code was written in Allegro ANSI Common Lisp, Version 8.2, by Prof.
;Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey,
;CA. Date of last revision: 13 March 2011.

;Allegro Prolog uses Lisp syntax. Rule head is first expression following "<--" symbol.
;Rule body is rest of expressions. Subsequent definitions of rule use "<-" symbol.

;Note that mission orders must be saved as "mission-orders.cl" in "Mission Control"
;folder, and then compiled before attempting execution by mission-controller. After
;compiling "mission-orders.cl", if "mission-controller.cl" has not been previously
```

```
;compiled it may be necessary to open it in a new Allegro Editor window to avoid "name
;conflict error" response from compiler.

(require :prolog) (shadowing-import '(prolog:==)) (use-package :prolog) ;Start Prolog.
(load "C:/Documents and Settings/mcghee/My Documents/Mission Control/mission-orders.fasl")

;Facts
(<-- (current_phase 0)) ;Start phase.


;Mission execution rule set

(<-- (execute_mission) (initialize_mission) (repeat) (execute_current_phase) (done) !)
(<-- (initialize_mission) (abolish current_phase 1) (asserta ((current_phase 1))))
(<-- (execute_current_phase) (current_phase ?x) (execute_phase ?x) !)
(<-- (done) (current_phase 'mission_complete))
(<- (done) (current_phase 'mission_abort))


;Human external agent communication functions

(<-- (negative nil))
(<- (negative n))
(<-- (affirmative ?x) (not (negative ?x)))
(<-- (report ?C) (princ ?C) (princ ".") (nl))
(<-- (command ?C) (princ ?C) (princ "!") (nl))
(<-- (ask ?Q ?A) (princ ?Q) (princ "?") (read ?A))


;Test function (illustrates format for calling for mission execution from Lisp)

(defun tm () (?- (execute_mission)))
```

Figure 10. Reduced State Universal Human Interactive Multiphase Mission Execution Automaton

Using the simplified MEA, simplified mission orders result. Specifically, the mission orders of Figure 8 become:

```
;C:/Documents and Settings/mcghee/My Documents/Mission Control/Mission Orders Archive/
;AVCL-mission.cl"

;This code was written in Allegro ANSI Common Lisp, Version 8.2, by Prof.
;Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey,
;CA. Date of last revision: 13 March 2011.

;This code can be executed only if it is first saved in /My Documents/Mission Control/ as
;"mission_orders.cl" and then compiled. When this has been done it is executed by loading
;and compiling "mission_controller.cl", (located in /My Documents/Mission Control/).

;The "<--" predicate definition symbol should be used only for the first definition of a
;given predicate. After that, subsequent definitions must use "<-" to avoid overwrite.

(require :prolog) (shadowing-import '(prolog:==)) (use-package :prolog) ;Start Prolog.


;Utility functions

(<-- (change_phase ?old ?new) (retract ((current_phase ?old))) (asserta ((current_phase ?new))))


;Mission specification

(<-- (execute_phase 1) (command "Search Area A") (phase_completed 1))
(<-- (phase_completed 1) (ask "Search successful" ?A) (affirmative ?A) (change_phase 1 2))
(<- (phase_completed 1) (change_phase 1 3))
(<- (execute_phase 2) (command "Sample environment") (phase_completed 2))
(<- (phase_completed 2) (ask "Sample obtained" ?A) (affirmative ?A) (change_phase 2 3))
(<- (phase_completed 2) (change_phase 2 5))

(<- (execute_phase 3) (command "Search Area B") (phase_completed 3))
(<- (phase_completed 3) (ask "Search successful" ?A) (change_phase 3 4))

(<- (execute_phase 4) (command "Rendezvous UUV2") (phase_completed 4))
(<- (phase_completed 4) (ask "Rendezvous successful" ?A) (change_phase 4 5))

(<- (execute_phase 5) (command "Return to base") (phase_completed 5))
(<- (phase_completed 5) (ask "At base" ?A) (affirmative ?A)
    (change_phase 5 'mission_complete) (report "Mission succeeded"))
(<- (phase_completed 5) (change_phase 5 'mission_abort) (report "Mission failed"))
```

Figure 11. Simplified Mission Orders Resulting from Using Reduced State MEA

The authors believe that the above simplified orders are considerably easier to read than those of Figure 8. Exhaustive testing of the code of Figure 10 and 11 yields results identical to those of obtained for the code of Figures 7 and 8, thereby satisfying a necessary condition for the validity of the reduced state MEA and its associated orders. Since MEA constitute a new mathematical formalism, there is as yet no theory of simplification comparable to that used in this memorandum for the simplification of Turing Machine state tables. Thus, all that can be said about these results is that they show that the definition of a universal MEA is not unique, and that the authors prefer the simplified MEA to the original definition. Further research may reveal even simpler MEA formulations.

D. REALIZATION OF UNIVERSAL TURING MACHINE AS AN MEA

In order to formally establish that MEA subsume Turing Machines, and are therefore "Turing complete", it is sufficient to demonstrate that a UTM can be realized as an MEA. The first step in such a proof is to "factor out" a *tape recorder agent* from the UTM of Figure 5. Lisp code realizing such an external agent is presented in Figure 12 below:

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Turing Machines/incremental-tape-
recorder.cl

;This code was written in Allegro ANSI Common Lisp, Version 8.2, by Prof. Robert B. McGhee
;(robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey,CA.
;Date of last revision: 13 May 2011.

;The tape is initially positioned so that the recorder read-write head is on the leftmost
;non-blank cell of the tape. After that the tape moves left or right according to entry in
;state table. A moving tape (rather than a moving head) was chosen for this code to make
;it easier for a human to emulate the recorder by using two "stacks" of paper with one sheet
;in between them to achieve the functionality of a doubly-infinite tape.
```

```
(defconstant sequence-1 '(1))
(defconstant sequence-2 '(0 0 0 1 1))
(defconstant sequence-3 '(0 0 1 1 0))
(defconstant sequence-4 '(1 1 1 0 1 0 0 1 0 0 0))
(defvar recorder-1)

(defclass incremental-tape-recorder ()
  ((tape :accessor tape :initform (make-instance 'doubly-infinite-tape))
   (input-register :accessor input-register)
   (cycle-count :accessor cycle-count :initform 0)
   (time-out-count :accessor time-out-count)
   (time-out-flag :accessor time-out-flag :initform nil)))

(defclass doubly-infinite-tape ()
  ((middle-cell-contents :accessor middle-cell-contents :initform nil)
   (right-side :accessor right-side :initform nil)
   (left-side :accessor left-side :initform nil)))

(defmethod read-tape ((recorder incremental-tape-recorder))
  (middle-cell-contents (tape recorder)))

(defmethod update-tape ((recorder incremental-tape-recorder))
  (let* ((symbol (first (input-register recorder))) (move (second (input-register recorder))))
        (write-and-move-tape recorder symbol move)))

(defmethod write-and-move-tape ((recorder incremental-tape-recorder) write-symbol movement)
  (setf (cycle-count recorder) (1+ (cycle-count recorder)))
  (setf (time-out-flag recorder) (if (>= (cycle-count recorder) (time-out-count recorder)) T))
  (case movement
    ('R (and (push write-symbol (right-side (tape recorder)))
             (setf (middle-cell-contents (tape recorder))
                   (pop (left-side (tape recorder))))))
    ('L (and (push write-symbol (left-side (tape recorder)))
             (setf (middle-cell-contents (tape recorder))
                   (pop (right-side (tape recorder))))))))

(defmethod tape-state ((tape doubly-infinite-tape))
  ;Character "H" brackets current input symbol.
  (append (reverse (left-side tape)) '(h) (list (middle-cell-contents tape))
          '(h) (right-side tape)))

(defmethod initialize-tape ((tape doubly-infinite-tape) initial-sequence)
  (setf (middle-cell-contents tape) (first initial-sequence)
        (right-side tape) (rest initial-sequence) (left-side tape) nil))

(defun set-up-recorder-and-tape (data-sequence max-cycles)
  (setf recorder-1 (make-instance 'incremental-tape-recorder))
  (initialize-tape (tape recorder-1) data-sequence)
  (setf (time-out-count recorder-1) max-cycles)
  (pprint (tape-state (tape recorder-1))))

(defun t1 () (set-up-recorder-and-tape sequence-1 20))

(defun t2 () (set-up-recorder-and-tape sequence-2 100))
```

Figure 12. Lisp Code for Incremental Tape Recorder External Agent

With the above code available, it becomes possible to modify the human interactive MEA code of Figure 5 to replace the human external agent with a tape recorder external agent. Figure 13 below incorporates such changes:

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Turing Machines/turing-mission-
controller.cl

;This code was written in Allegro ANSI Common Lisp, Version 8.2, by Prof.
```

```
;Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey,
;CA. Date of last revision: 16 May 2011.

;Allegro Prolog uses Lisp syntax. Rule head is first expression following "<--" symbol. Rule
;body is rest of expressions. Subsequent definitions of rule use "<-" symbol.

;Note that mission orders must be saved as "mission-orders.cl" in "Mission Control" folder,
;and then compiled before attempting execution by mission-controller. After compiling
;"mission-orders.cl", if "turing-mission-controller.cl" has not been previously compiled, it
;may be necessary to open it in a new Allegro Editor window to avoid "name conflict error"
;response from compiler.

(require :prolog) (shadowing-import '(prolog:==)) (use-package :prolog) ;Start Prolog.
(load "C:/Documents and Settings/mcghee/My Documents/Tech Reports/Turing Machines/mission-
orders.fasl")
(load "C:/Documents and Settings/mcghee/My Documents/Tech Reports/Turing Machines/incremental-
tape-recorder.fasl")


;Facts (execution status)

(<-- (current_phase 0)) ;Start phase.

;Mission execution rule set

(<-- (execute_mission) (initialize_mission) (repeat) (execute_current_phase) (done) !)
(<-- (initialize_mission) (abolish current_phase 1) (asserta ((current_phase 1))))
(<-- (execute_current_phase) (current_phase ?x) (execute_phase ?x) !)
(<-- (done) (current_phase 'accept))
(<- (done) (current_phase 'reject))
(<- (done) (current_phase 'incomplete))


;Tape recorder external agent communication functions

(<-- (get_input ?x) (is ?x (read-tape recorder-1)))
(<-- (get_time_out_flag ?x) (is ?x (time-out-flag recorder-1)))
(<-- (set_output ?x) (lisp ?x (setf (input-register recorder-1) ?x)))
(<-- (write_and_move) (lisp (update-tape recorder-1)))
(<-- (update_tape ?x) (set_output ?x) (write_and_move))

;Utility functions

(<-- (change_phase ?old ?new) (retract ((current_phase ?old)))
                              (asserta ((current_phase ?new))))
(<-- (test_time_out) (get_time_out_flag ?x) (== ?x T))
(<-- (execute_phase ?x) (test_time_out) (change_phase ?x 'incomplete) (report " Incomplete"))
(<- (execute_phase ?x) (get_input ?y) (execute_action ?x ?y))
(<-- (report ?C) (princ ?C) (princ ".") (nl))


;Test functions (illustrate query format)

(defun tm1 () (set-up-recorder-and-tape sequence-1 20) (?- (execute_mission)))

(defun tm2 () (set-up-recorder-and-tape sequence-2 100) (?- (execute_mission)))

(defun tm3 () (set-up-recorder-and-tape sequence-3 100) (?- (execute_mission)))

(defun tm4 () (set-up-recorder-and-tape sequence-4 100) (?- (execute_mission)))

(defun tm5 () (set-up-recorder-and-tape sequence-4 1000) (?- (execute_mission)))

(defun tm6 () (?- (set_output (1 R))))

(defun tm7 () (input-register recorder-1))

(defun tm8 () (?- (set_output (X R))))

(defun tm9 () (?- (update_tape (X R))) (pprint (tape-state (tape recorder-1))))

(defun tm10 () (?- (write_and_move)) (pprint (tape-state (tape recorder-1))))
```

Figure 13. UTM Realized as MEA with Tape Recorder External Agent

As can be seen by comparing Figures 10 and 13, the human external agent code and the tape recorder external agent code differ only in revision of the agent communication functions and in the addition of a third definition of the *done* predicate. This additional definition is needed to ensure that Turing Machine mission execution terminates in finite time for all missions.

To demonstrate the correctness of the code of Figure 13, the state table of Figure 4 can be rewritten as MEA mission orders. One way to do this is provided as Figure 14 below:

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Turing Machines/
;zero-crossing-mission.cl"

;This code was written in Allegro ANSI Common Lisp, Version 8.2, by Prof.
;Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School in Monterey,
;CA. Date of last revision: 15 May 2011.

;This code can be executed only if it is first saved in /My Documents/Tech Reports/
;Turing Machines/ as "mission_orders.cl" and then compiled. When this has been done, it can
;be executed by loading and compiling "turing-mission_controller.cl", which is also located
;in this directory.

;The "<--" predicate definition symbol should be used only for the first definition of a
;given predicate. After that, subsequent definitions must use "<-" to avoid overwrite.

(require :prolog) (shadowing-import '(prolog:==)) (use-package :prolog) ;Start Prolog.


;Mission specification

(<-- (execute_action 1 0) (change_phase 1 2) (update_tape (X R)))
(<- (execute_action 1 1) (change_phase 1 3) (update_tape (X R)))
(<- (execute_action 1 X) (change_phase 1 1) (update_tape (X L)))
(<- (execute_action 1 nil) (change_phase 1 'reject) (report " Reject"))

(<- (execute_action 2 0) (change_phase 2 2) (update_tape (0 R)))
(<- (execute_action 2 1) (change_phase 2 4) (update_tape (X R)))
(<- (execute_action 2 X) (change_phase 2 2) (update_tape (X R)))
(<- (execute_action 2 nil) (change_phase 2 4) (update_tape (0 L)))

(<- (execute_action 3 0) (change_phase 3 4) (update_tape (X R)))
(<- (execute_action 3 1) (change_phase 3 3) (update_tape (1 R)))
(<- (execute_action 3 X) (change_phase 3 3) (update_tape (X R)))
(<- (execute_action 3 nil) (change_phase 3 4) (update_tape (1 L)))

(<- (execute_action 4 0) (change_phase 4 4) (update_tape (0 L)))
(<- (execute_action 4 1) (change_phase 4 4) (update_tape (1 L)))
(<- (execute_action 4 X) (change_phase 4 1) (update_tape (X L)))
(<- (execute_action 4 nil) (change_phase 4 'accept) (report " Accept"))
```

Figure 14. Zero-Crossing Detector State Table Rewritten as MEA Mission Orders

With the definitions provided by Figures 12 and 14, running the MEA code of Figure 13 produces the results presented in Figure 15. The results are in agreement with those of Figure 3, thereby demonstrating the correctness of the associated code. Moreover, the authors believe that the MEA realization of Turing Machine missions would be much easier for a non-programmers to understand because no ability to read Lisp is required (providing that tape recorder functionality is taken as a given).

```
CG-USER(1): (tm1)

(H 1 H) Reject.
Yes

No.
CG-USER(2): (tm2)

(H 0 H 0 0 1 1) Reject.
Yes

No.
CG-USER(3): (tm3)

(H 0 H 0 1 1 0) Accept.
Yes

No.
CG-USER(4): (tm4)

(H 1 H 1 1 0 1 0 0 1 0 0 0) Incomplete.
Yes

No.
CG-USER(5): (tm5)

(H 1 H 1 1 0 1 0 0 1 0 0 0) Accept.
Yes

No.
CG-USER(6):
```

Figure 15. Results Obtained by MEA for Zero-Crossing Detector Turing Machine Mission

THIS PAGE INTENTIONALLY LEFT BLANK

IV. HOW TO RUN LISP/PROLOG CODE EXAMPLES

The reader is invited to copy and execute the code presented in this report. In order to accomplish this, a free trial copy of Allegro Common Lisp 8.2, including an integrated development environment (IDE), can be downloaded from www.franz.com. When this system has been installed, the code of interest can be copied from this report and pasted into an Allegro Editor pane. It should then be saved in an appropriate directory, and compiled (by clicking on the “dumptruck” icon). Entering commands to the debug window, as shown in Figures 3 and 9, should produce the indicated results. Of course the *load* function calls in your code should be modified to match your file structure before compilation.

THIS PAGE INTENTIONALLY LEFT BLANK

V. SUMMARY AND CONCLUSIONS

This report provides executable definitions of two types of Turing machines (TM), and two examples of universal mission execution automata (MEA). Since MEA subsume Turing Machines, it follows that *all* Turing Machines can be encoded as MEA by simply providing an incremental tape recorder as an external agent, and expressing the state table in the form of mission orders. However, this does *not* mean that MEA are *computationally* more powerful. They are, rather, a kind of *real-time* TM, which have been taken "out of their box" and into the real world through their human or robotic external agents. This makes them *useful* in a practical sense, rather than remaining a purely theoretical concept. Finally, while every TM is an infinite machine, MEA may be either *finite* or *infinite* (an infinite tape recorder is not required for many missions). This relates to practicality and expense in real machines.

The definition of an MEA can be applied recursively to refine individual mission phases until reaching *primitive subphases*, needing no further decomposition before execution. The authors believe this to be important since exhaustive debugging of even a five phase mission proved to be tedious and error prone.

The MEAs defined in this report are in no sense unique. Further work could determine that there are additional features that could be added to the MEAs presented here to make them more user friendly. Were that to be done, the enhanced MEA would simply be another type of universal MEA. Obviously, for dealing with a robot external agent, new communication functions will be needed. In addition, further *facts* (execution status registers) may be needed for robot MEAs. When more experience has been obtained with robot external agents, it may turn out that a common *mission execution rule set* will serve for humans, robots, and tape recorders. Such a rule set could then be said to be "universal" without qualification as to the nature of the MEA external agent. Belief in this possibility is enhanced by the fact that the rule sets for human and tape recorder agents provided in this report differ only trivially.

In summary, MEA implemented in Prolog provide a means of stating mission orders in an executable form that may easier for mission specialists to read than other executable forms. Specifically, at the present time, XML together with Java provides the only practical alternative to Prolog known to the authors for expressing executable mission specifications directly from a natural language mission definition [14]. An argument can be made that it is substantially easier for a non-programmer to learn to read Prolog than it would be to learn to read XML and associated Java code. More research is needed to determine the possibility of other solutions to this problem, and to investigate the relative utility of each. Finally, readers should be aware that the combination of commercial *industrial strength* Lisp and Prolog on Windows and similar platforms, as used in this report, is new technology, provided at this time by only a single source [5], and only since about 2003. The stabilization of Prolog in the form of an ISO standard was completed only in 2000. Since then, there has been a modest proliferation of Prolog implementations [15]. Some of these may turn out to be more suited to imbedded systems than the Prolog/Lisp MEA implementation used in this report. Much remains to be learned about what can be accomplished using these tools in a variety of realms of application, including specifically UUV mission specification and execution. We look forward to dialogue with others interested in this important topic [16].

LIST OF REFERENCES

1. Minsky, M.L., *Computation: Finite and Infinite Machines*, Prentice Hall, 1967.
2. Petzold, C., *The Annotated Turing*, Wiley Publishing, 2008.
3. Graham, P., *ANSI Common Lisp*, Prentice Hall, 1996.
4. Kohavi, Z., and Niraj, K.J., *Switching and Finite Automata Theory*, McGraw Hill, 2010.
5. Franz, Inc., Allegro Prolog Online Documentation, 2011. Available at www.franz.com/support/documentation/current/doc/prolog.html
6. Norvig, P., *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann Publishers, 1992.
7. McGhee, R.B., "Future Prospects for Sensor-Based Robots," in *Computer Vision and Sensor-Based Robots*, pp. 323-333, ed. by G. G. Dodd and L. Rossal, Plenum Publishing Corp., 1979.
8. Brutzman, D., et al, "The Phoenix Autonomous Underwater Vehicle", *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, Ch. 13, pp. 323-360, ed. by Kortenkamp, D., et al, MIT Press, Cambridge, MA 02142, 1998.
9. Frege, G., *Begriffsschrift*. 1879. Translated in van Heijenoort, J., *From Frege to Gödel: A Source Book on Mathematical Logic*, 1879-1931. Harvard University Press, 1967.
10. Hofstadter, D.R., *Godel, Escher, Bach: An Eternal Golden Braid*, Basic Books, New York, 1999, pp. 204 - 230.
11. Rowe, N.C., *Artificial Intelligence Through Prolog*, Prentice Hall, Englewood Cliffs, NJ 07632, 1988.
12. Russell, S.J., and Norvig, P., *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.
13. Marco, D.B., Healey, A.J., and McGhee, R.B., "Autonomous Underwater Vehicles: Hybrid Control of Mission and Motion", *Autonomous Robots 3*, pp. 169-186, 1996.
14. Davis, D.T., and Brutzman, D.P., "The Autonomous Unmanned Vehicle Workbench: Mission Planning, Mission Rehearsal, and Mission Replay Tool for Physics-Based X3D Visualization", *Proc. Of 14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August, 2005.
15. Wikipedia contributors, "Comparison of Prolog Implementations," Wikipedia, The Free Encyclopedia, April 2011. Available at www.wikipedia.org/wiki/Comparison_of_Prolog_implementations

16. McGhee, R. B., Brutzman, D. P., and Davis, D. T., “A Universal Multiphase Mission Execution Automaton (MEA) with Prolog Implementation for Unmanned Untethered Vehicles”, *Proc. Of 17th International Symposium on Unmanned Untethered Submersible Technology*, Portsmouth, NH, August, 2011. Available at https://savage.nps.edu/AuvWorkbench/website/documentation/papers/papers.html

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943