Proceedings of the 17th International Symposium on Unmanned Untethered Submersible Technology, Portsmouth, NH, August, 2011.

A UNIVERSAL MULTIPHASE MISSION EXECUTION AUTOMATON (MEA) WITH PROLOG IMPLEMENTATION FOR UNMANNED UNTETHERED VEHICLES

R. B. McGhee, D. P. Brutzman, and D. T. Davis
Naval Postgraduate School, MOVES Institute
Monterey, CA 93943
robertbmcghee@gmail.com, brutzman@nps.edu and dtdavi1@nps.edu

Abstract—The authors have been involved for a considerable time in research relating to computer control of robotic vehicles and mechanisms. For the past two decades, our work in this area has been focused primarily on unmanned untethered submersibles (UUVs), especially those intended for eventual military use. This being the case, we have been guided in our efforts by our knowledge of the way task abstraction and mission execution are accomplished in manned submarines. This led us some time ago 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 strategic level mission orders 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 paper, we show how to realize an MEA 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 paper contains one detailed example along with all Prolog code used for mission specification and execution. While we have explored and discuss other realizations of MEAs, we find none to be as well suited as the Prolog implementation to verification of executable mission orders by mission specialists. Because our MEA model is based on formal mathematical logic, we are able to demonstrate "proof of correctness" of the code for the selected mission. We believe this to be of fundamental importance for military missions, and perhaps as well for some classes of civilian missions.

1. Introduction

The authors have been engaged for some time in research relating to computer control of complex robotic vehicles, including *walking machines* and *autonomous underwater vehicles* [1, 2, 3, 4]. Since 1996, we and a number of our colleagues at the Naval Postgraduate School (NPS) have used the *Rational Behavior Model* (RBM) tri-level software architecture to organize our activities in this area, especially in relation to control of autonomous submersibles [3, 4, 5]. In the vocabulary of the RBM formalism, the *execution* level of vehicle control software is concerned with carrying out the *hard real-time tasks* typically associated with physical interaction of a vehicle with its surrounding medium. In a manned submarine, these tasks are usually carried out by *crew members*, and include responsibilities such as controlling diving planes, rudders, engine rpm, etc. Again in RBM terminology, above the execution level lie the *soft real-time* tasks of the *tactical* level. At the tactical level, execution level functions are organized into *behaviors*, which are sometimes associated with *autopilots*. Such functions include maintaining course and depth, sonar obstacle avoidance, maneuvering while surfaced, etc. However, the tactical level also includes more complicated behaviors such as sonar mapping, transit to navigational waypoints, construction of obstacle maps, response to emergency situations, etc. In a manned submarine, the Officer of the Deck (OOD) is responsible for coordinating the actions of the *watch officers* and crew members to ensure that such behaviors are correctly implemented and carried out [4, 5].

The highest level of software in the RBM architecture is called the *strategic* level [3, 4, 5]. This level corresponds to the functioning of the *commander* of a manned submarine. Unlike the lower two levels, in RBM the strategic level operates entirely in a discrete event, non-numeric mode, considering alternative actions and making decisions without a sense of continuous time or space. That is, it is entirely in the domain of *mathematical logic*. This being the case, the strategic level so far has been implemented at NPS mainly by using *logic programming* in the form of the *Prolog* language [6, 7]. This language was employed successfully in encoding the strategic level for both test tank evaluation [3], and for open ocean missions conducted with the *Phoenix* unmanned autonomous submersible [4]. Although this vehicle has been retired from service, we have continued our research on RBM implementation by using *computer simulation*, including both real-time control software and a detailed representation of the nonlinear hydrodynamics of the Phoenix [8]. However, in this work, we have experienced to date a disappointingly slow rate of progress in developing improved strategic level software. This has been due in part to the lack of a strong mathematical model for the functioning this level of vehicle control. This paper addresses this need by defining and implementing a new type of mathematical *machine* [9] that we call a *mission execution automaton* (MEA). While the domain of application that motivates our work is onboard mission control for unmanned untethered vehicles (UUVs), it will be seen in this paper that the MEA we define subsumes *Turing machines* [9, 10], and thus constitutes a more general concept.

A Turing machine (TM) consists of a *finite state machine* (FSM) augmented by an *external agent* in the form of a *potentially infinite* memory realized as the *tape* of an "incremental tape recorder" [10]. It is known that no digital machine can be more computationally powerful than a *universal* Turing machine, in which the logical behavior of a specific FSM is encoded on the tape of the machine in the form of a *state table* [9, 10]. Nevertheless, partly because their programming is so difficult [10], Turing machines have been almost exclusively relegated to the status of a *mathematical* concept, with practical computing being accomplished by *digital computers*. The main idea developed in this paper relates to a generalization of TMs to MEAs by allowing the external agent to be not only a tape recorder, but alternatively, either a human being or a *sensor-based robot* [11].

The authors have chosen to use Prolog to both define and implement a *universal multiphase human interactive* MEA. We have done so because of our belief in the strong expressive power of Prolog predicate definitions when read declaratively [6], while at the same time representing executable code. In what follows, it is assumed that the reader is familiar with ANSI Common Lisp [12], at least to the extent of being able to read the relatively simple code presented in the figures of this paper. A brief explanation of the syntax and semantics of the *Allegro* dialect of Prolog [7, 13], implemented in Common Lisp, is included in what follows.

2. A Universal Human Interactive Multiphase Mission Execution Automaton (MEA)

Complex missions to be carried out by human agents 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 specialized natural language as in Figure 1 below.

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*. To avoid errors in execution, it is assumed that the syntax and semantics of specialized natural language mission orders are understood in the same way by both the person issuing the orders and the person receiving them. Orders written to achieve this objective are said to be syntactically *well formed* and semantically *unambiguous*. Figure 2 contains Prolog code for a *universal* human interactive multiphase MEA. This machine is believed to be "universal" in the sense that it is suited to cycling mission states for any set of well formed and unambiguous orders for a multiphase mission to be carried out by a human being.

3. Reading Prolog Code

In reading the code of Figure 2, per Lisp convention, it is important to recognize 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 third line of this code is a Prolog *fact* [6, 7]. This fact

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.

Figure 1: Example Manned Submarine Mission Orders Expressed in Structured Natural Language Using Standardized Vocabulary

```
;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 2: A Universal Human Interactive Multiphase Mission Execution Automaton (MEA)

states that the current mission phase is Phase 0 (Start phase). Turning next to the *mission execution rule set*, Allegro Prolog [7, 13] 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, 7], it can be seen that when the *done* predicate fails, Prolog *backtracks* and tries to find another way of executing the current phase, which leads to searching the fact data base for a new value for *current_phase*. Providing that the previous call to *execute_ phase* has updated this fact appropriately, this action continues until *done* is satisfied by either mission completion or mission abort. Finally, the *execute_mission* and *execute_current_phase* predicate definitions end 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, the cut assures that when the test function *tm* is called, as intended, only *one* attempt to execute a mission will occur. Likewise, *execute_current_phase* can be entered only from the left, thereby ensuring that the latest value for *current_phase* will be used in executing this function call.

Turning next to the other rules in Figure 2, 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 [13]. Next, “initializing” a mission involves abolishing the “Start phase” and replacing it with “Phase 1”. This asserts a convention of this MEA definition that execution of any multiphase mission 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. 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 from top to bottom. Logic variables are uniquely signified in Allegro Prolog by the first character in the variable name being a “?” character. Once a logic variable acquires a value, the *unification* feature of Prolog [6, 7] 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 because a human being is able to respond to a restricted and predefined set of commands, queries, and statements to and from the MEA. More generally, a sensor-based robot could fulfill this function [3,4].

```
;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 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.


;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 3: Prolog Mission Orders for a Human Interactive Submarine Reconnaissance Mission

4. Mission Specification

Figure 3 contains additional Prolog code for *mission orders* defining and implementing the above described reconnaissance mission. Because of the interactive nature of these mission orders, the external agent communication functions defined in the "mission-controller.cl" code provide for issuing commands, making statements, and asking questions via a computer screen, and also for receiving responses from the keyboard. These predicates make use of the Common Lisp system functions *princ* and *read* [12]. In addition, the Prolog system functions *not* and *nl* (new line) are used in these definitions. Of course, in the case of UUV applications, the intent of such interactive execution is to validate mission coding by a human expert before *embedding* the mission controller in a real physical vehicle. Thus this form of code constitutes a kind of *Turing test* [14] for the mission controller and a specific set of mission orders. It is the authors' belief that no mission controller or mission orders should be deployed in an actual physical robot until such a test has been successfully completed. A "tongue in cheek" way of expressing this is that a human should provide *artificial AI* to the mission controller MEA for the first stage of mission debugging. Once this stage has been completed, subsequent testing must verify the functioning of this system by using *real AI* from the robot vehicle.

The code of Figure 3 can be viewed as providing *executable mission specifications*. That is, this code can be either read declaratively as specifications by a human, or compiled to *executable code* by the Prolog compiler. This is the main advantage of the RBM software architecture and the MEA realization of the strategic level. That is, when a Prolog implementation of the strategic level of RBM is used, no *recoding* of mission orders into computer readable form is needed.

Examining the code of Figure 3, it can be seen that five mission phases are defined. Beyond this observation, the authors feel that this code is self explanatory, so no further discussion is provided here. Rather, the results of an interactive debugging session are presented below as Figure 4. It is important for the reader to note that, per comments at the top of Figure 2, obtaining results of this sort requires that code for "mission-orders.cl" be appropriately compiled and loaded before calling "execute_mission" via the given test function "tm."

It is the authors' opinion that the above results are in agreement with the natural language definition of this mission. However, Figure 4 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. We have ourselves found this kind of dialogue to be very useful during the writing of the present paper.

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 [3].

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

CG-USER(1): (?- execute_mission))
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(2): (?- execute_mission))
Search Area A!
Search successful?y
Sample environment!
Sample obtained?n
Return to base!
At base?y
Mission succeeded.
Yes

No.
CG-USER(3): (?- execute_mission))
Search Area A!
Search successful?n
Search Area B!
Search successful?n
Rendezvous UUV2!
Rendezvous successful?y
Return to base!
At base?n
Mission failed.
Yes

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

Figure 4: Partial Test Results for Submarine Reconnaissance Mission Execution (user input in bold)

5. Mission Execution as Theorem Proving

From the perspective of *first order predicate logic*, the MEA presented above constitutes a *formal system*. Viewed in this way, the code of Figure 2 can be called the *axioms* of the system. A specific set of mission orders, as in Figure 3, constitutes a *theorem* to be proved by purely algebraic means, without reference to the semantics of either the axioms or the mission [15]. Specifically, if Prolog successfully compiles a set of mission orders and associated mission controller, then the Prolog syntax is correct. Once this has happened, if *execute_mission* returns "yes", then the mission orders have been proved "true". This is the case regardless of whether the outcome of the mission is *mission_complete* or *mission_abort*. If Prolog fails to return "yes" then the mission orders are not "true", and Prolog returns "no". Such an outcome could come about because no value could be found for one or more unbound logic variables, because an infinite loop has resulted from unintended errors in phase completion conditions, because of Prolog syntax or semantic errors detected only at run time, or for many other reasons.

In understanding the above discussion from the perspective of mathematical logic, it is important to realize that Prolog implements only a subset of first order predicate logic [6, 7, 15]. In particular, Prolog uses "proof by example" as its sole means for theorem proving. This means that it can prove only *existentially quantified* theorems [15], and even then only in the world defined by the *facts* presented to it. Thus when Prolog says "no", it means "I couldn't find a binding of logic variables that satisfies your query within the rules and facts you provided to me". While this sounds limiting, it may be exactly the kind of behavior desired of an autonomous robot, since too much freedom in mission execution could potentially lead to disastrous unforeseen consequences. As a final remark, from a theorem proving point of view, the actual execution of a specific mission resulting from Prolog calls to the tactical level during theorem proving is a *side effect* [12].

6. Turing Machine Realization as MEA Mission Orders

As defined thus far, there are no limits on MEA mission orders other than that the format implied by the Prolog definitions of the MEA in Figure 2 be respected. That is, mission orders must take the form of a series of phases with predetermined mission end conditions, and specified state transitions conditioned on the outcome of phase execution. In Turing machine terminology [9], such orders define a *state table* in which state transitions are determined by the *current state* (current mission phase) and the *input* from the doubly infinite *tape* of the machine. From the perspective of MEA mission orders, a "Turing machine mission" involves a specific type of external agent, called an "incremental tape recorder", capable only of reading from or writing to the tape using a finite predefined set of symbols, and moving the tape right or left one step as determined by the state table. Clearly, tape recorder functionality could be achieved either by mechanical means, by computer simulation [10], or by a human external agent.

7. How to Run Lisp/Prolog Code Examples

The reader is invited to copy and execute the code presented in this paper. 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 and pasted into an Allegro Editor pane. It should then be saved in an appropriate directory, and compiled (by clicking on the "dumptruck" icon). When this has been done, entering commands to the *debug* window, as shown in Figure 4, should produce the indicated results. Of course the *load* function calls in your code should be modified to match your file structure before compilation.

8. Alternatives to Prolog for MEA Realization

It is an important to note that, although Prolog provides an attractive mechanism for MEA realization, it is not the only option. Generally speaking any data manipulation system that is *Turing complete*, including context sensitive grammars, lambda calculus, and all commonly utilized computer programming languages, is suitable for MEA implementation [9]. In virtually all cases, however, the MEA will be indecipherable by anyone unfamiliar with that particular system. Other declarative programming languages, especially those based on predicate logic, might allow for fairly user-friendly implementations, but the authors have yet to encounter a system capable of realizing both the MEA and the mission orders that provides the intuitive readability of Prolog.

If the MEA and the mission orders are implemented with different systems, it is possible to achieve a level of readability approaching that of our Prolog implementation. If, for instance, the MEA is implemented with a programming language along the lines of Java or C++, the mission orders FSM can be implemented using a more user-friendly mechanism. One such system utilizes a Java program to execute FSM missions authored in an Extensible Markup Language (XML) vocabulary [16].

Use of XML provides a format that is specifically designed to be easily read and interpreted by both computers and humans, making it an attractive choice for the definition of autonomous vehicle missions. Figure 5 depicts one possible XML encoding of the multi-phase UUV mission from the previous natural language and Prolog examples. In the authors' opinion, this version is equivalent to the previous versions and is as intuitive as well.

```
<?xml version="1.0" encoding="UTF-8"?>
<UUVMission>
    <GoalSet>
        <Goal area="A" id="goal1">
            <Search nextOnSucceed="goal2" nextOnFail="goal3"/>
        </Goal>
        <Goal area="A" id="goal2">
            <SampleEnvironment nextOnSucceed="goal3" nextOnFail="recover"/>
        </Goal>
        <Goal area="B" id="goal3">
            <Search nextOnSucceed="goal4" nextOnFail="goal4"/>
        </Goal>
        <Goal area="C" id="goal4">
            <Rendezvous nextOnSucceed="recover" nextOnFail="recover"/>
        </Goal>
        <Goal area="recoveryPosition" id="recover">
            <Transit nextOnSucceed="missionComplete" nextOnFail="missionAbort"/>
        </Goal>
    </GoalSet>
</UUVMission>
```

Figure 5: XML Mission Orders for a Human Interactive Submarine Reconnaissance Mission

Because XML is not a programming language, it is not suitable for full MEA implementation by itself. Notwithstanding the availability of programming language functionality that makes XML easier to process than many other potential mission-specification formats, full MEA implementation will be significantly more complex than a Prolog version. A more thorough discussion of XML/Java implementation details and requirements, including a discussion of the full XML vocabulary of the implementation, can be found in [8] and [16].

9. MEA Testing and Integration

As discussed in Section 4, the MEA mission-control paradigm has been tested primarily by querying a human external agent rather than a sensor-based robot. It is the authors' intention to continue testing and development in simulation and ultimately in real vehicles. Specifically, the MEA will be implemented to direct vehicles operating in the Autonomous Unmanned Vehicle Workbench [8] virtual environment. This system has also been shown to be suitable for use with actual vehicles of various types [16].

It goes without saying that strategic level mission specification for an MEA does not contain all of the information required to define all aspects of a mission. For example, the locations and characteristics of the operating areas, the launch and recovery positions, and the specific objectives and requirements of the individual goals must be specified before the mission can commence. It is an interesting observation, however, that this information is completely irrelevant to the MEA—if the information is available to the lower control layers, they will be able to respond appropriately to MEA queries. An implementation as depicted in Figure 6 is therefore appropriate for MEA control of arbitrary real or simulated vehicles.

In this implementation, a phase controller is instantiated for each phase of the mission (i.e., each state of the MEA FSM). The phase controller is implemented as a Java object that provides tactical-level control for the completion of a single phase. All numerical data related to the completion of a single phase is maintained by or available to the phase controller for that phase. Because the requirements of each mission phase are different, a separate phase controller object is instantiated for each phase. Thus, the mission of Figure 6 will require five phase controllers for the defined phases. As the mission is executed, the MEA ensures that only the controller for the current phase is active at any specific time.

The phase controller is responsible for all activity and path planning in support of phase execution, sensor fusion and interpretation, and onboard system monitoring. Most importantly, the phase controller monitors the progress of the current phase, provides direction to the lower control levels, and responds appropriately to strategic-level MEA queries. In order to meet its control requirements, each phase controller must have access to parameters such as operating area, timing constraints, and any other phase-specific requirements. Additionally, vehicle state information, sensor data, and onboard systems status must be obtained from other tactical-level modules or from the execution level controller.

Since the communication mechanism of the MEA consists solely of queries, control of the tactical level must be realized as a side effect of these queries. Queries are actually implemented as function calls from the Prolog MEA to the Java phase control object. Because the Prolog associated with a specific phase only makes calls (i.e., queries) to the phase controller for its phase, activation and deactivation of the individual controllers is an implicit byproduct of the MEA-level state transitions. Each MEA-query will initiate a single cycle of the tactical-level control loop. When queried by the strategic-level MEA, a phase

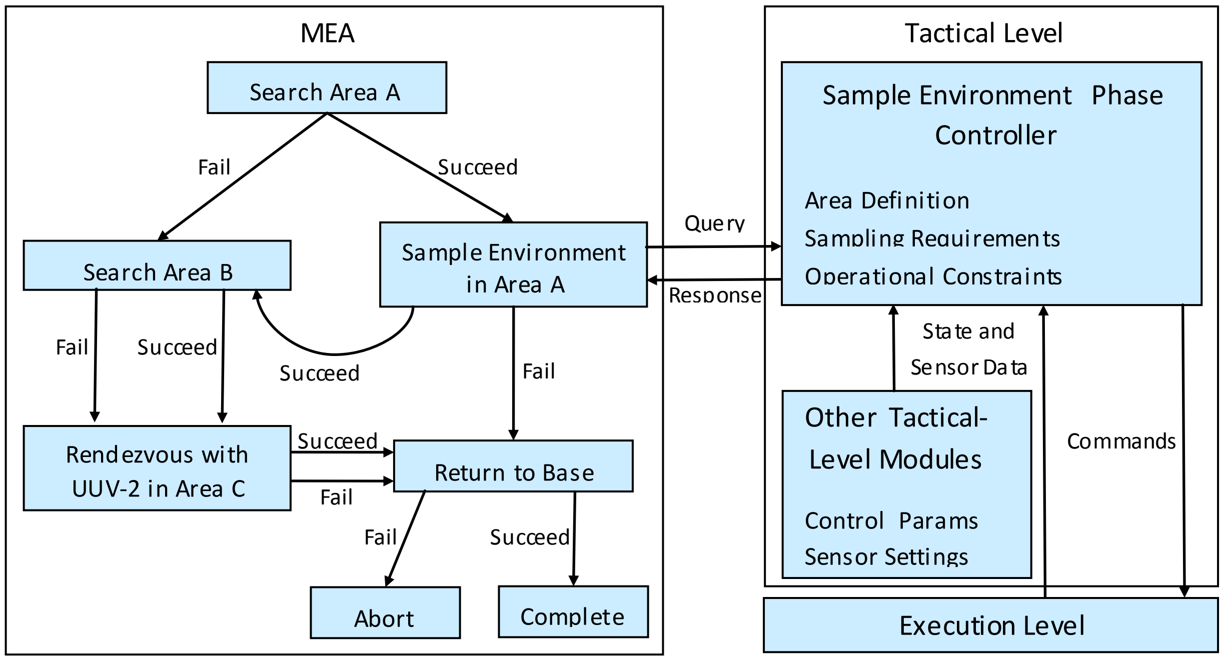

Figure 6: MEA Implementation for a Sensor-Based Robot

controller obtains and analyzes all required data from the execution level and other tactical-level modules, performs any required planning and assesses phase progress, provides direction to the execution level as required, and responds to the query. In this way, the phase controller implements a tactical-level sense-decide-act loop in support of a single mission phase, and the MEA controls transitions between mission phases.

10. Summary and Conclusions

Thoughtful analysis of the above results shows that any *specific* multiphase mission could be specified as an FSM without making use of the MEA and mission orders abstraction. However that is not the intent of our work. Rather, we are looking for a *fully general* means capable of *animating* any syntactically correct set of mission orders. That is, through the use of the RBM and MEA formalisms, we desire to replicate in a UUV the level of "end-user programmability" attained in a manned submarine by means of formal written mission orders.

We have not dealt with problems of embedding an MEA in a real vehicle. This is vehicle specific, and requires that a finite set of queries and responses be defined for communication between the strategic and tactical software levels, and that real-time execution issues be resolved. However, we have done some work of this sort [3, 4, 8], and intend to do more. Moreover, in [10], a fully coded example of the use of a tape recorder external agent to achieve a universal Turing machine as an MEA is provided. This example proves the Turing completeness of MEA.

It is noteworthy that all of the code presented in this paper uses only *eight* Prolog system functions. Moreover, most of these functions have common English names relating to their behavior. We know of no other computer programming language with so few primitive functions. This is one reason that we are optimistic about the practicality of mission specialists being able to read mission orders written in Prolog after only a short period of training. This possibility is enhanced by the fact that Prolog execution closely resembles human reasoning with a *one track mind* dedication to a specific task.

Nothing in either Prolog or our definition of an MEA requires a binary response to queries from the mission controller. For example, a response from an external agent of "ny" could stand for the meaning "not yet" by simply asserting the Prolog fact: (not_yet ny). To understand this, note that this is analogous to the definition of "affirmative" and "negative" in the code presented above. Evidently, allowing for more than two answers to queries permits general n-way branching on exit from mission phases. In addition, more than just two halt states are possible for an MEA. An example of a mission exhibiting both of these features can be found in [10].

In this paper, we have introduced and explained what we believe to be a new formalism for specification and execution of arbitrary multiphase missions by unmanned vehicles. However, we do not consider what we have done to be a contribution to *artificial intelligence*, since the performance we envisage for MEA and their associated vehicles is far too limited and regimented to be compared to that of human beings. On the other hand, we do hope that we have made a contribution to *machine intelligence* by defining and implementing a general solution to the problem of achieving enhanced unmanned vehicle mission specification and execution, without constraints on the length or branching factors of mission orders.

In summary, MEA implemented in Prolog provide a means of stating mission orders in an executable form that may be easier for mission specialists to read than when written in other languages. Moreover, MEA mission orders are subject to mathematical *proof of correctness* by means of exhaustive pre-mission testing involving dialogue between the originator of the natural language orders and the person responsible for their Prolog implementation. This being the case, we believe that the use of a *multilingual* implementation of UUV control software as in RBM facilitates *transparency* and *accountability* in planning, coding, and after action evaluation of autonomous mobile robot missions. These are key issues in gaining acceptance of such robots as *trusted highly-autonomous decision-making systems*, one of four "grand challenge" science and technology problems selected by the US Air Force as being central to national defense for the next twenty years [17].

Finally, at the present time, XML together with Java (or another high-level programming language) provides the only practical alternative to Prolog known to the authors for expressing executable mission specifications more or less directly from natural language mission definition. More research is needed to determine the possibility of other solutions to this problem, and to investigate the relative utility of each. With respect to the languages used in this paper, readers should be aware that the combination of commercial *industrial strength* Lisp and Prolog on Windows and similar platforms is *new technology*, provided at this time only by one source [13], 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 proliferation of Prolog implementations [18]. Some of these may turn out to be more suited to imbedded systems than the Prolog/Lisp implementation used in this paper. 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 topic.

7. References

[1] McGhee, R.B., "Vehicular Legged Locomotion," in *Advances in Automation and Robotics*, Vol. 1,pp. 259-284, ed. by G. N. Saridis, Jai Press, Inc., 1985.

[2] Song, S.M., and Waldron, K. J., Machines That Walk: The Adaptive Suspension Vehicle, MIT Press, Cambridge, MA, 1989.

[3] 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.

[4] 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.

[5] Byrnes, R.B., et al, "The Rational Behavior Software Architecture for Intelligent Ships", *Naval Engineers Journal*, pp. 43-55, March, 1996.

[6] Rowe, N.C., *Artificial Intelligence Through Prolog*, Prentice Hall, Englewood Cliffs, NJ 07632, 1988.

[7] Norvig, P., *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann Publishers, 1992.

[8] 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.

[9] Minsky, M.L., *Computation: Finite and Infinite Machines*, Prentice Hall, 1967.

[10] McGhee, R.B., Brutzman, D.P., and Davis, D.T., *A Taxonomy of Turing Machines and Mission Execution Automata with Lisp/Prolog Implementation*, Technical Report NPS-MV-11-002, Naval Postgraduate School, Monterey, CA 93943, July, 2011. Available at https://savage.nps.edu/AuvWorkbench/website/documentation/reports/reports.html

[11] 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.

[12] Graham, P., *ANSI Common Lisp*, Prentice Hall, 1996.

[13] Franz, Inc., Allegro Prolog Online Documentation, 2011. Available at www.franz.com/support/documentation/current/doc/prolog.html

[14] Russell, S.J., and Norvig, P., *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.

[15] Hofstadter, D.R., *Godel, Escher, Bach: An Eternal Golden Braid*, Basic Books, New York, 1999, pp. 204 - 230.

[16] Davis, D.T., Brutzman, D.P., and Becker, W.J., "Facilitation of Autonomous Vehicle Coordination through an XML-Based Vehicle-Independent Control Architecture", *Proc. Of the 16th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August, 2009.

[17] *Technology Horizons*, Vol. 1, AF/ST-TR-10-01, United States Air Force Chief Scientist, 15 May 2010, pg. 100. Available at www.flightglobal.com/assets/getasset.aspx?ItemID=35525

[18] Wikipedia contributors, "Comparison of Prolog implementations," *Wikipedia, The Free Encyclopedia*, April 2011. Available at http://en.wikipedia.org/w/index.php?title=Comparison_of_Prolog_implementations&oldid=425200212