NPS-MV-12-001
17 Apr 12

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

RECURSIVE GOAL REFINEMENT AND ITERATIVE TASK ABSTRACTION FOR TOP-LEVEL CONTROL OF AUTONOMOUS MOBILE ROBOTS BY MISSION EXECUTION AUTOMATA—A UUV EXAMPLE

by

Robert McGhee, Don Brutzman, Duane Davis

March 2012

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From-To)*
	Technical Report	1 Aug 2011 – 1 Mar 2012

4. TITLE AND SUBTITLE

Recursive Goal Refinement and Iterative Task Abstraction for Top-Level Control of Autonomous Mobile robots by Mission Execution Automata—a UUV Example

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)

Robert McGhee, Don Brutzman, Duane 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

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-MV-12-001

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Naval Modeling and Simulation Office
1333 Isaac Hull Ave. Stop 5012
Washington Navy Yard, D.C. 20376-5012

10. SPONSOR/MONITOR'S ACRONYM(S)
NMSO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
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 necessarily reflect the official policy or position of the Department of Defense of the U.S. Government

14. ABSTRACT

A *Mission Execution Automaton* (MEA) is a generalization of a Turing Machine (TM) that allows either a human being or a sensor-based mobile robot to serve as an *external agent* to the finite state machine (FSM) part of a TM. This approach is in contrast to the standard definition of a TM that allows only an incremental tape recorder to fulfill this function. Thus, MEA constitute a *proper superclass* of TM, and have a potential for wider application. In this report, the possibility of using MEA for top level control of autonomous mobile robots is explored by means of a Lisp/Prolog computer simulation.

A *universal* MEA consists of a fixed *mission execution engine* and a set of *mission orders* to specialize the resulting automaton to a specific mission. That is, such a mission execution engine, together with its orders, define the finite state part of a *mission specific* MEA. This report presents a Prolog implementation of a mission execution engine and includes a set of orders, also written in Prolog, for an example "area search and sample" mission for an autonomous unmanned undersea vehicle (UUV). A Lisp simulation of a particular UUV and its environment permits testing of a complete mission in the given circumstances.

The software architecture used in this simulation is based on the *Rational Behavior Model* (RBM) in which the top *strategic* level is realized by means of an MEA. Below this is a *tactical* level, that coordinates the functioning of controllers for physical devices at an *execution* level. In this report, the tactical level is simulated by a combination of Lisp and Prolog, while the execution level is entirely in Lisp.

Functions implemented at the tactical level are called *behaviors*, and are defined in this report by Lisp function definitions in a hierarchical arrangement. This process is often called *behavior subsumption*. Thus, creation of more complex behaviors is accomplished by a bottom up process of *task abstraction*. On the other hand, executable goals are reached by a top down process of *recursive refinement* of strategic level goals. This process terminates when goal refinement reaches far enough down to attain a *basis condition* in which only previously defined tactical level behaviors are called.

The report presents a methodology that allows for a gradual migration from execution of mission orders by a human tactical officer to execution by an entirely robotic tactical officer. Proof of mission order correctness through exhaustive testing is available at every stage of this process. The authors feel that such proofs are essential before actual deployment of autonomous mobile robots in hostile environments.

15. SUBJECT TERMS
Autonomous Underwater Vehicle (AUV), Unmanned Underwater Vehicle (UUV), Autonomous Robot Control, Turing Machine

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** ***(include area code)***
Unclassified	Unclassified	Unclassified	UU		

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

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Executive Vice President and
Provost

Further distribution of all or part of this report is authorized.

This report was prepared by:

R. B. McGhee
Emeritus Professor

D. P. Brutzman
Associate Professor

D. T. Davis
Assistant Professor

Reviewed by:

CDR Joseph Sullivan, Ph.D.
MOVES Institute

Released by:

Douglas Fouts
Interim Vice President and
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A *Mission Execution Automaton* (MEA) is a generalization of a Turing machine (TM) that allows either a human being or a sensor-based mobile robot to serve as an *external agent* to the finite state machine (FSM) part of a TM. This approach is in contrast to the standard definition of a TM that allows only an incremental tape recorder to fulfill this function. Thus, MEA constitute a *proper superclass* of TM, and have a potential for wider application. In this report, the possibility of using MEA for top level control of autonomous mobile robots is explored by means of a Lisp/Prolog computer simulation.

A *universal* MEA consists of a fixed *mission execution engine* and a set of *mission orders* to specialize the resulting automaton to a specific mission. That is, such a mission execution engine, together with its orders, define the finite state part of a *mission specific* MEA. This report presents a Prolog implementation of a mission execution engine and includes a set of orders, also written in Prolog, for an example "area search and sample" mission for an autonomous unmanned undersea vehicle (UUV). A Lisp simulation of a particular UUV and its environment permits testing of a complete mission in the given circumstances.

The software architecture used in this simulation is based on the *Rational Behavior Model* (RBM) in which the top *strategic* level is realized by means of an MEA. Below this is a *tactical* level, that coordinates the functioning of controllers for physical devices at an *execution* level. In this report, the tactical level is simulated by a combination of Lisp and Prolog, while the execution level is entirely in Lisp.

Functions implemented at the tactical level are called *behaviors*, and are defined in this report by Lisp function definitions in a hierarchical arrangement. This process is often called *behavior subsumption*. Thus, creation of more complex behaviors is accomplished by a bottom up process of *task abstraction*. On the other hand, executable goals are reached by a top down process of *recursive refinement* of strategic level goals.

This process terminates when goal refinement reaches far enough down to attain a *basis condition* in which only previously defined tactical level behaviors are called.

The report presents a methodology that allows for a gradual migration from execution of mission orders by a human tactical officer to execution by an entirely robotic tactical officer. Proof of mission order correctness through exhaustive testing is available at every stage of this process. The authors feel that such proofs are essential before actual deployment of autonomous mobile robots in hostile environments.

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

In [1, 2] the concept of a *Mission Execution Automaton* (MEA) is introduced. Briefly, an MEA is a generalization of a Turing Machine (TM) that allows either a human being or a sensor-based mobile robot to function as an *external agent* for the finite state machine (FSM) part of a TM. This is in contrast to the standard definition of a TM that allows only an incremental tape recorder to fulfill this function. Thus, MEA constitute a *proper superclass* of TM, and have a potential for wider application. Specifically, in [1], an example autonomous Unmanned Undersea Vehicle (UUV) area search and sample mission is implemented as a Lisp/Prolog computer simulation using the *Rational Behavior Model* (RBM) software architecture [3, 4], with the top (Strategic) level realized as an MEA.

The primary advantage of MEAs is that since *mission orders* are interpreted by them as finite state machines, the correctness of such orders can be established before download to an autonomous mobile robot through exhaustive interactive testing conducted by human mission experts. When mission orders are written in appropriately structured Prolog, MEAs offer the further advantage that such orders can be read declaratively by mission specialists, and procedurally by a Prolog compiler, so that no recoding is needed for human testing and mission rehearsal. However, with respect to validation of mission orders through exhaustive testing, experience shows that the number of phases in the mission must be small, probably not more than five or six, or else such testing will become impracticable due to a "combinatorial explosion" (too many cases for human accomplishment in an acceptable time). This report shows how this problem can be overcome by *recursive refinement* of commands (goals) from a strategic level MEA until a *basis condition* is reached in which only implemented *tactical* level tasks (or "behaviors" [3, 4]) are called to accomplish execution of a given command. This approach is used in this report to refine a depth first area search command from the MEA of [1] to a basis of Lisp functions derived from a less efficient Brownian motion (completely random) search procedure. This example concretely demonstrates the feasibility and effectiveness of combined use of recursive goal refinement and iterative

task abstraction ("subsumption") in autonomous mobile robot control software development using an MEA at the top level.

It is important to realize that strategic level mission control by MEA does not attempt to model the kind of complex dialogue that typically occurs between humans in execution of a mission in a manned submarine. Rather, MEA control is totally *authoritarian* in nature. That is, when this implementation of the RBM architecture is adopted, the actions of the subordinate tactical level in interacting with the strategic level are strictly limited to executing one of a predefined finite set of commands, and responding to a finite set of predefined queries with one of a finite set of answers. In this report, tactical level responses are limited to just two values, "yes" and "no." This is done to facilitate exhaustive testing of a given set of mission orders. Despite the apparent severity of these constraints, we believe that such restrictions may be exactly what is needed to produce a highly desirable type of *disciplined* behavior by an autonomous robot during mission execution. In addition, we believe that these constraints do not affect the *generality* of the mission control approach we are advocating. We provide examples in this report to support this view.

II. A FINITE STATE MODEL FOR HUMAN MISSION SPECIFICATION AND EXECUTION

A. UNIVERSAL MEA AND MISSION ORDERS

In [1, 2], the concept of a *universal* MEA *mission controller*, capable of carrying out any mission for any type of external agent when supplied with appropriately formatted mission orders is introduced. In these references, a prototypical reconnaissance mission for a UUV is also defined using structured natural human language, and subsequently translated into Prolog mission orders. The dialect chosen for this purpose is Allegro Prolog [6, 7], implemented in Common Lisp [8]. The code proposed for achieving the desired functionality, as presented in [1], is as follows in Figure 1. We now prefer to refer to such a controller as a *Mission Execution Engine* (MEE).

```
;C:/Documents and Settings/mcghee/My Documents/Papers/Tech Memos/Recursive Refinement/
;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: 15 July 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 same folder as
;mission-controller.cl, 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/Papers/Tech Memos/Recursive
Refinement/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 1. Prolog code for Universal Mission Execution Engine with Human External Agent Communication Functions

Corresponding example mission orders, also from [1], are as listed below in Figure 2.

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive Refinement/
;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 2. Prolog Mission Orders for Human Execution of Example UUV Area Search and Sample Mission

For readers unfamiliar with Allegro Prolog, a concise explanation of the syntax and semantics of this dialect is provided in [1, 2]. As a further aid to understanding,

Figure 3 below presents a more abstract (language independent, but not executable) representation of the logic of the example mission in the form of a state graph.

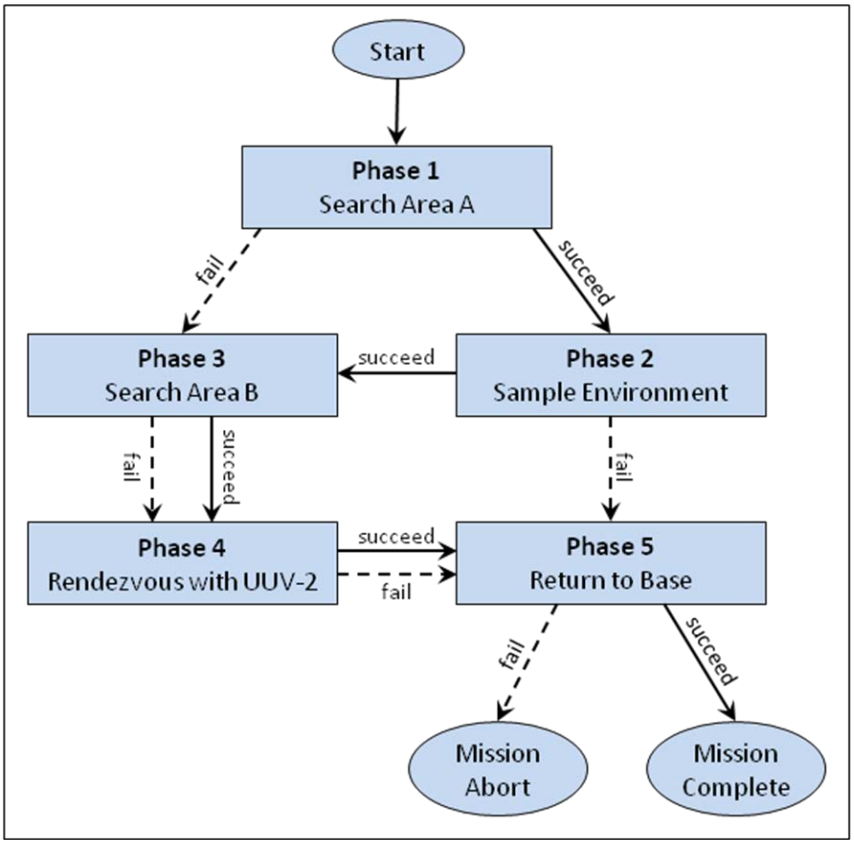

Figure 3. State Graph for Example UUV Area Search and Sample Mission

It should be noted that two of the phases enclosed by ovals in the above graph are *terminal* phases (or states) that have no successor states. In addition, there is a "Start" phase that has no predecessor state.

Figure 4 below provides examples of human interaction with Prolog during simulated execution of the above mission. For ease of understanding, user responses have been manually highlighted in bold in this figure.

Referring to Figures 2 and 3, it can be seen that each of the given examples is in agreement with the given mission orders. Specifically, in the first example, the mission is truncated, and the UUV returns to base due to its failure to obtain a sample in Area A. In the second example, Area A search fails, so sampling is omitted, but Area B search and rendezvous with UUV2 succeed. Nevertheless, since the UUV fails to return to base,

per mission orders, the mission is deemed to have failed. Finally, in the third example, Area A search and sampling succeed, but Area B search and rendezvous with UUV2 fail. Despite these failures, the mission succeeds because the UUV successfully returns to base.

```
CG-USER(1): (?- (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(2): (?- (execute_mission))
Search Area A!
Search successful?n
Search Area B!
Search successful?y
Rendezvous UUV2!
Rendezvous successful?y
Return to base!
At base?n
Mission failed.
Yes

No.

CG-USER(3): (?- (execute_mission))
Search Area A!
Search successful?y
Sample environment!
Sample obtained?y
Search Area B!
Search successful?n
Rendezvous UUV2!
Rendezvous successful?n
Return to base!
At base?y
Mission succeeded.
Yes

No.
```

Figure 4. Examples of Human Response to MEA Queries in Execution of Area Search and Sample Mission (User Responses in Bold)

At this point, several questions arise. First of all, is the above behavior that which is desired by the human mission specialist who wrote the initial natural language orders? Next, if this is the case, does exhaustive testing involving all other possible user response sequences correspond to the intentions of this or other mission specialists? Finally, can the mission orders be simplified so as to facilitate testing and understanding? All of these questions will be addressed in the remainder of this section of this report.

B. SIMPLIFICATION OF MISSION ORDERS

In [1, 2], there is no assumption that the inputs to an MEA, whether from a human or some form of robotic or mechanical external agent, are binary in nature. However, in the above described example, user responses to queries issued by the MEA mission orders are in fact binary. This being the case, some simplification of the mission orders can be achieved by omission of the *phase_complete* predicate appearing in the above Figure 2. This is possible because there are at most two branches out of each phase of these mission orders. This is in contrast to the more general *n-way* branching allowed for other types of MEA mission orders [2]. The result of this simplification is as listed in Figure 5 below:

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive Refinement/
;simplify1-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: 25 October 2011.

(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") (ask "Search successful" ?A) (affirmative ?A)
     (change_phase 1 2))
(<- (execute_phase 1) (change_phase 1 3))

(<- (execute_phase 2) (command "Sample environment") (ask "Sample obtained" ?A)
    (affirmative ?A) (change_phase 2 3))
(<- (execute_phase 2) (change_phase 2 5))

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

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

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

Figure 5. Simplification of Example Mission Orders Resulting from Omitting Phase_Complete Predicate

Testing the above code with the examples of Figure 4 above yields identical results, but the code is more compact and somewhat more readable. At this point we noted, partially as a result of the improved readability of the simplified mission orders,

that *change_phase* predicate calls are unconditional in Phase 3 and Phase 4, and consequently their queries are *irrelevant* to mission execution. This means that, without changing the way in which the commands are issued in execution of the given mission, these queries can be omitted, and Phases 3 and 4 can then be combined into a new phase, Phase 34. These changes lead to the further code simplification of Figure 6.

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive Refinement/
;Mission Orders Archive/simplified-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: 18 January 2012.

;This code can be executed only if it is first saved in /My Documents/Tech Reports/
;Recursive Refinement/ 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/Tech Reports/Recursive Refinement/.

;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") (ask "Search successful" ?A) (affirmative ?A)
     (change_phase 1 2))
(<- (execute_phase 1) (change_phase 1 34))

(<- (execute_phase 2) (command "Sample environment") (ask "Sample obtained" ?A)
    (affirmative ?A)
    (change_phase 2 34))
(<- (execute_phase 2) (change_phase 2 5))

(<- (execute_phase 34) (command "Attempt Area B search")
    (command "Attempt rendezvous with UUV2") (change_phase 34 5))

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

Figure 6. Reduced State Area Search and Sample Mission Orders

At this point it can be observed that, in the above figure, only four phases remain out of the five defined in Figure 2. Thus, using the terminology of finite state machine theory [9], it can be said that Figure 6 presents a *reduced state* version of the original mission orders. The significance of this reduction is explained in the next section of this report. In addition, it should be noted that the commands issued by State 34 have been

prefixed to include the word “attempt.” This was done in order to emphasize that subsequent commands are not conditioned on the outcome of such an attempt.

C. VALIDATION OF MISSION ORDERS THROUGH EXHAUSTIVE TESTING

To insure that the above code actually represents the intentions of the mission specialist, it can be executed interactively and exhaustively by that person, either acting alone, or in concert with a subordinate “tactical level” officer. Figure 7 below shows the results of such a test for the given code. It should be noted in the results listed that, for convenience, and for fidelity to actual screen capture results, manual highlighting of user responses as in Figure 4 has been omitted. This will also be the case in subsequent testing results presented later in this report.

```
CG-USER(1): (?- (execute_mission))
Search Area A!
Search successful?n
Attempt Area B search!
Attempt rendezvous with UUV2!
Return to base!
At base?n
Mission failed.
Yes

No.

CG-USER(2): (?- (execute_mission))
Search Area A!
Search successful?n
Attempt Area B search!
Attempt rendezvous with UUV2!
Return to base!
At base?y
Mission succeeded.
Yes

No.

CG-USER(3): (?- (execute_mission))
Search Area A!
Search successful?y
Sample environment!
Sample obtained?n
Return to base!
At base?n
Mission failed.
Yes

No.

CG-USER(4): (?- (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(5): (?- (execute_mission))
Search Area A!
Search successful?y
Sample environment!
Sample obtained?y
Attempt Area B search!
Attempt rendezvous with UUV2!
Return to base!
At base?n
Mission failed.
Yes

No.

CG-USER(6): (?- (execute_mission))
Search Area A!
Search successful?y
Sample environment!
Sample obtained?y
Attempt Area B search!
Attempt rendezvous with UUV2!
Return to base!
At base?y
Mission succeeded.
Yes

No.
```

Figure 7. Exhaustive Human Interactive Testing of Area Search and Sample Mission Orders

The authors recognize that obtaining and interpreting the above kind of results can be challenging, especially when more phases are involved than in this example. However, we have collectively reviewed Figure 7, and agree that the six cases listed correspond to what we intended when writing the above mission orders. Nevertheless, there were some surprises that arose in this process that led to considerable discussion. For example, before the above tests, we thought that the missions defined in [1] and [5] were identical. However, subsequently, we came to realize that the mission of [5] included no explicit *abort* phase. We then wondered if such a phase is needed, or can it be implicit? Our conclusion is that since the definition of an MEA mission controller in Figure 1 above includes an explicit abort phase, it is better to include this phase in mission orders. This is especially true during recursive refinement of orders since executing an abort command will generally involve some action, such as, for example, “scuttle” or “surface for recovery.” Such actions should be unambiguously defined in lower level code before undertaking autonomous mission execution by a real physical UUV.

Another important question is: "How do we know that the above test is exhaustive?" After all, the mission orders include three questions, each of which can be answered in two ways, so shouldn't there be eight cases to be tested? The answer to this question is "no," the reason being that some missions are executed with less than three responses being invoked. The response to the prompts CG-USER(1) and CG-USER(2) show this to be true. Specifically, the "Sample environment" command associated with Phase 2 of the mission is not issued in either of these cases because, as can be seen from the mission orders, Phase 2 is entered only upon success of Phase 1. This results in a total of only six cases rather than eight.

Some deeper questions arose from our discussions of the results of Figure 7. Specifically, referring to CG-USER(3), why should the vehicle return to base upon failing to get a sample from Area A? Wouldn't it be better for it to go on and attempt to sample Area B? For that matter, if the purpose of the mission is to obtain a sample, why not return to base after *success* of sampling in Area A rather than in the event of *failure* of this phase? Was this a mistake in the original human natural language mission definition given in [1]? Only appeal to a higher authority can answer this kind of question. However, this is an additional value of human mission rehearsal that we feel will prove to be essential in military or other hazardous missions. In particular, we believe that no potentially lethal robot mission orders should be executed by an operational vehicle until they have passed the kind of "Turing test" [10] exemplified by Figure 7 above. When all questions have been resolved concerning exhaustive testing results, then in the most meaningful sense of the term, the Prolog mission code can said to have been "proved correct." At this point, the highest ranking person participating in this test could be designated as the legally responsible individual authorizing its use as *executable specifications* for the development of corresponding mission orders to be downloaded to a *robot* external agent.

Before leaving the issue of mission order validation by exhaustive testing, it is important to note that the complexity of such testing grows roughly exponentially with the number of queries in a given set of mission orders. In particular, we have carried out full exhaustive testing of the five-phase version of the mission orders listed in Figure 2 above, and found that eighteen cases are involved. This is because the irrelevant queries

generated by the original States 3 and 4 can each be answered “yes” or “no,” thereby increasing the number of possible response sequences without affecting mission execution. Another way of putting this statement is that the answers to these queries have value only for data logging, and do not affect the sequence of commands issued in any particular mission execution scenario. Because of the additional queries generated by these states, the results of exhaustive testing of the mission orders of Figure 2 were substantially harder to obtain and interpret than those of Figure 7 above.

As a final remark, the above exhaustive testing of mission execution is possible only because the control flow graph of Figure 3 is loop free. If mission orders contain a loop, then the possibility of an infinite number of possible test cases arises. As demonstrated in [2], this can be dealt with by utilizing a *loop count* or *time out* failure mode in tactical level behavior implementation.

III. ENVIRONMENT AND UUV CAPABILITIES

So far, in this report, the selected mission has been described only at a very high level of abstraction. Specifically, nothing has been said about either the environment in which the mission is to take place, or the sensing and movement capabilities of the UUV that is to carry out the mission. To address this issue, suppose, for example, that the environment constitutes an (idealized) minefield laid out in a rectangular planar grid, with random placement of mines within the grid. Figure 8 below shows a simple table providing a graphical representation of such a minefield in a "list of lists" (Lisp) format. In this figure, a "1" entry represents a "mine" and a "0" represents "no mine." Following the usual Lisp convention [8], Row 0 is the top row and Column 0 is the left column. For simplicity, suppose further that the UUV tasked to reach a goal (located at an unknown position within the minefield) is constrained to move only in an up-down (north-south) or a right-left (east-west) direction within each cycle of motion.

```
((1 1 1 1 1 1 1 1)
 (1 0 0 0 0 0 1 1)
 (1 1 0 1 1 0 0 1)
 (1 0 1 0 1 0 0 1)
 (1 0 1 1 0 1 0 1)
 (1 1 0 0 1 0 0 1)
 (1 0 0 0 0 0 1 1)
 (1 1 1 1 1 1 1 1))
```

Figure 8. Example of Planar Rectangular Grid Minefield

To illustrate the coordinate system for this grid, note that element (3, 1) is a 0 in an area completely surrounded by six 1's. This means that this point is *isolated* and cannot be reached by a UUV operating outside this area. Noting that element (3, 3) is surrounded by four l's, if a UUV were to start there, it would be *immobilized,* and could go nowhere. Now suppose that the goal to be reached by the UUV is the point (5, 5). Then, if it were to start at in cell (6, 3), one way it could get to the goal would be by moving first two steps to the right to cell (6, 5), then up to cell (5, 5). This is the shortest path for this move. Obviously, more circuitous paths are possible, and would result if the vehicle were, for example, to first move up. Now the question is: "How could an

autonomous UUV accomplish such a traversal to the goal?." The answer depends delicately on the sensing and computing capabilities assumed for the vehicle. Two alternative possibilities are presented in this report.

One approach to searching such an area is to assume that the UUV can detect mines (by sonar, for example), but has no idea where the goal is until it finds it, and also is unable to determine its own location. By analogy to *bacterial* behavior, one (very inefficient) way for it to find the goal is to simply wander randomly; i.e., to conduct a search by "Brownian motion." The Lisp code in the file "brownian-motion-search.cl" listed in Figure 9 accomplishes this.

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive Refinement/
;brownian-motion-search.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: October 9 2011.

;This search method implements brownian motion search in a bounded obstacle field.  It does not
;utilize any form of terrain marking or terrain memory.  Location of goal is not known a
priori.
;The list *path-to-goal* is not used by the agent conducting the search.  It is provided solely
;for support of human debugging and performance analysis.


;-----------------------------------Environment and Search State------------------------------

(defvar *path-to-goal* nil)
(defvar *virtual-obstacle-list* nil)
(defvar *goal-location* nil)
(defvar *robot-location* nil)
(defvar *terrain* nil)
(defvar *area-A* (make-array '(8 8)
   :initial-contents '((1 1 1 1 1 1 1 1) ; An entry of 1 denotes an obstacle.
                       (1 0 0 0 0 0 1 1)
                       (1 1 0 1 1 0 0 1)
                       (1 0 1 0 1 0 0 1)
                       (1 0 1 1 0 1 0 1)
                       (1 1 0 0 1 0 0 1)
                       (1 0 0 0 0 0 1 1)
                       (1 1 1 1 1 1 1 1))))


;-------------------------------------Execution Level----------------------------------------

(defun obstaclep (location) ; location is *terrain* index list '(row column)
  (= (aref *terrain* (first location) (second location)) 1))

(defun possible-movep (location)
  (if (not (obstaclep location)) location))

(defun south-movep (location)
  (possible-movep (cons (1+ (first location)) (rest location))))

(defun north-movep (location)
  (possible-movep (cons (1- (first location)) (rest location))))

(defun west-movep (location)
  (possible-movep (list (first location) (1- (second location)))))

(defun east-movep (location)
  (possible-movep (list (first location) (1+ (second location)))))
```

```
(defun go-to (location) (setf *robot-location* location))


;---------------------------------------Tactical Level-------------------------------------

(defun possible-move-list (location)
  (remove nil (list (north-movep location) (south-movep location)
                    (west-movep location) (east-movep location))))

(defun order-random-move-from (location);Test prevents movement from completely enclosed start.
  (if (possible-move-list location) (make-random-move-from location)))

(defun random-select (list) (nth (random (length list)) list))

(defun make-random-move-from (location)
  (push location *path-to-goal*)
  (go-to (random-select (possible-move-list location))))

(defun brownian-find-path (start search-area goal-location nloop)
  (setf *path-to-goal* nil *robot-location* start *terrain* search-area *goal-location*
        goal-location)
  (do ((count (1- nloop) (1- count))
       (location *robot-location* (order-random-move-from location)))
      ((or (equal location *goal-location*) (null location) (zerop count))
           ;null means immobilized agent.
       (if (equal location *goal-location*)
           (pprint (reverse (cons location *path-to-goal*)))))))


;---------------------------------------Test Functions-------------------------------------

(defun test1 () (brownian-find-path '(2 2) *area-A* '(3 1) 100)) ;Isolated goal.

(defun test2 () (brownian-find-path '(3 3) *area-A* '(3 1) 100)) ;Immobilized start.

(defun test3 () (brownian-find-path '(6 3) *area-A* '(5 5) 100)) ;Short path.

(defun test4 () (brownian-find-path '(3 6) *area-A* '(6 2) 100)) ;Long path.
```

Figure 9. Lisp code for Brownian Motion Search

Examining the above code, it can be seen that there are six *global* variables declared: *goal-location*, *path-to-goal*, *virtual-obstacle-list*, *robot-location*, *area-A*, and *terrain*. Following this, a series of *execution level* functions are defined that simulate the UUV's ability to move and to detect obstacles (mines). Next, tactical level functions are specified to organize sensing and motion functions so as to traverse a minefield to find a goal whose location is not known *a priori*. These functions make use of a *possible move list* associated with any specified terrain location. For example, referring to the *area-A* array in the code, the possible move list from cell (5, 3) would include cells (5, 2) and (6, 3), but not cells (4, 3) or (5, 4). Next, the functions *random-select* and *make-random-move-from* accomplish a random move to one of the cells in the possible move list of a given location. Finally, the function *brownian- find-path* uses random motion to move about in the search space until either the goal has been found, or the *agent* (think of a single bacterium) "dies" after a specified number of steps. The test

functions *test1*, *test2*, *test3*, and *test4* provide a convenient means for testing the Brownian motion search code, as shown below in the example results of Figure 10.

```
CG-USER(1): (test1)
NIL

CG-USER(2): (test2)
NIL

CG-USER(3): (test3)

((6 3) (6 4) (6 5) (6 4) (6 5) (5 5))

CG-USER(4): (test4)
NIL

CG-USER(5): (test4)

((3 6) (4 6) (5 6) (5 5) (6 5) (6 4) (6 5) (5 5) (6 5) (6 4) (6 5) (6 4) (6 5) (5 5)
 (6 5) (6 4)(6 5) (5 5) (5 6) (5 5) (6 5) (5 5) (6 5) (5 5) (6 5) (5 5) (6 5) (5 5) (6 5)
 (6 4) (6 3) (6 4) (6 3) (5 3) (5 2) (5 3) (6 3) (5 3) (6 3) (6 2))
```

Figure 10. Results Obtained for Brownian Motion Search Test

Examining the results of Figure 10 shows that the call to *test1* failed. This is as expected since the goal is isolated in this case. Likewise, the call to *test2* failed, but in this case, this is because the search agent is immobilized at the start. In the call to *test3*, the goal was found but not by a minimum length path. Rather, the robot wandered around and crossed its own path once before finding the goal. In understanding this behavior, it is important to note that the agent is kept from wandering away from the minefield altogether because the search area is enclosed by a boundary of "mines." This being the case, completely random motion is certain (in a probabilistic sense), given enough time, to find the goal, providing that it is reachable from the start location.

On the first call to *test4*, the search fails because the agent "dies" before reaching the goal. This happens because the goal is 10 steps away by the shortest path, and in this instance random searching doesn't allow the agent to travel this far in its specified 100 step "lifetime." However, on the second call, the agent fortuitously wanders into the goal and returns the path found. It is important to realize that the search procedure itself makes no use of the global variable *path-to-goal*. Rather, the value of this path is simply returned to the human user of this code as an aid to understanding of the behavior of Brownian motion search. The next section of this report shows how searching can be improved if the agent uses the value of this variable in deciding on the next move.

IV. RECURSIVE COMMAND REFINEMENT AND DEPTH FIRST SEARCH

If it is assumed that an agent (such as a UUV) has a navigation system so that it knows its own position, then it can keep a list of places where it has been (the *path-to-goal* list) and thus search more efficiently. Moreover, with this capability, it can also keep track of places where it was forced to back up ("backtrack") and try another move, in the form of a global *virtual-obstacle-list*, and use this list to avoid blundering into the same situation again. This strategy is called "depth-first-search" [11]. Assuming that a UUV exists with tactical level software equivalent to the Lisp code in the "brownian-motion-search.cl" file listed in Figure 9, then depth first search of "Area A" can be carried out by a human calling by the given functions in this file from a keyboard according to the strategy of Figure 11.

Step 0: Initialize. Specifically, set *terrain* to *area-A*, and *robot-location* and *goal* to the desired values. Set *path-to-goal* and *virtual-obstacle-list* to *nil*. To eliminate the trivial case of search, the variables *goal* and *robot-location* should not be set to the same values. Evidently, neither one of them should be set to the location of an obstacle cell.

Step 1: Move forward. That is, move robot to an adjacent cell that is not an obstacle and is not on the path to the goal. If no such cell exists, or if at goal, then go to Step 2. Otherwise, add the new cell to the head of the *path-to-goal* list and repeat.

Step 2: If at goal, end and report success. If at starting location, end and report failure. Otherwise, go to Step 3.

Step 3: Backtrack search. That is, add the present cell to the *virtual obstacle list*, move to previous location, and pop the *path-to-goal* list. Go to Step 1.

Figure 11. Structured Natural Language Definition of Grid-Based Depth First Search Strategy

The question now is how can a command from the strategic level MEA such as "Search Area A" be carried out using depth first search together with the Lisp functions defined in Figure 9? One way to do this is to *recursively refine* strategic level commands

until a *basis condition* is reached in which only implemented tactical level functions are called. This approach will be taken in this section of this report.

The authors believe that the easiest way to obtain a correct set of mission orders for a new mission is through stepwise editing of an existing set of orders known to be correct for some other mission. Thus, starting with the orders of Figure 2 above, the following set of (incomplete) orders for carrying out the strategy of Figure 11 can be obtained.

```
;C:/Documents and Settings/mcghee/My Documents/Papers/Tech Memos/Recursive Refinement/
;Mission Orders Archive/2-state-mission-orders.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: 6 October 2011.

(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 "Initialize Area A search") (ask "Initialization completed" ?A)
     (affirmative ?A) (change_phase 1 5))
(<- (execute_phase 1) (change_phase 1 1))

(<- (execute_phase 5) (command "Determine location") (ask "At goal" ?A) (affirmative ?A)
    (change_phase 5 'mission_complete) (report "Mission succeeded"))
(<- (execute_phase 5)  (ask "At start" ?A) (affirmative ?A)
    (change_phase 5 'mission_abort) (report "Mission failed"))
(<- (execute_phase 5) (change_phase 5 1))
```

Figure 12. Two-Phase (Incomplete) Mission Orders for Depth First Search by Human Agent

The correctness of the above orders, so far as they go, is partially established by the testing results of Figure 13.

While the code of Figure 12 is shown to be syntactically correct by its successful compilation, the above tests reveal that it contains a loop in which Phase 1 returns to itself until it succeeds. A permanent initialization failure would thus cause the mission to be stuck in this phase with no escape possible. In addition, this code commands no motion, and so is incomplete. Continuing to use the code of Figure 2 as a model, the code of Figure 12 can be altered as depicted in Figure 13 to address these problems.

```
CG-USER(1): (tm)
Initialize Area A search!
Initialization completed?n
Initialize Area A search!
Initialization completed?y
Determine location!
At goal?n
At start?y
Mission failed.
Yes

No.

CG-USER(2): (tm)
Initialize Area A search!
Initialization completed?y
Determine location!
At goal?y
Mission succeeded.
Yes

No.
```

Figure 13. Partial Testing of Two-Phase (Incomplete) Depth First Search Mission Orders by Human Agent

```
;C:/Documents and Settings/mcghee/My Documents/Papers/Tech Memos/Recursive Refinement/
;Mission Orders Archive/4-state-mission-orders.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: 7 October 2011.


(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 "Initialize Area A search") (ask "Initialization completed" ?A)
     (affirmative ?A) (change_phase 1 2))
(<- (execute_phase 1) (change_phase 1 'mission_abort) (report "Mission failed"))

(<- (execute_phase 2) (command "Move to available cell") (ask "Move successful" ?A)
    (affirmative ?A) (change_phase 2 3))
(<- (execute_phase 2) (change_phase 2 5))

(<- (execute_phase 3) (command "Push previous cell location onto path-to-goal list")
    (ask "Action completed" ?A) (affirmative ?A) (change_phase 3 2))
(<- (execute_phase 3) (change_phase 3 3))

(<- (execute_phase 5) (command "Determine location") (ask "At goal" ?A) (affirmative ?A)
    (change_phase 5 'mission_complete) (report "Mission succeeded"))
(<- (execute_phase 5)  (ask "At start" ?A) (affirmative ?A)
    (change_phase 5 'mission_abort) (report "Mission failed"))
(<- (execute_phase 5) (change_phase 5 1))
```

Figure 14. Four-Phase (Incomplete) Mission Code for Execution by Human Agent

From Figure 15 below, it can be seen that while this code also compiles and executes, and corrects the two problems noted in the code of Figure 12, it fails to initiate backtracking when there is no new cell available to continue the depth first search, and instead (incorrectly) calls for an additional initialization.

```
CG-USER(1): (tm)
Initialize Area A search!
Initialization completed?n
Mission failed.
Yes

No.

CG-USER(2): (tm)
Initialize Area A search!
Initialization completed?y
Move to available cell!
Move successful?y
Push previous cell location onto path-to-goal list!
Action completed?y
Move to available cell!
Move successful?n
Determine location!
At goal?n
At start?n
Initialize Area A search!
Initialization completed?
```

Figure 15. Human Execution of Four-Phase Depth First Search Mission Orders

To fix the omission of backtracking, Phases 2 and 3 of Figure 14 above can be combined into a new Phase 2, and then two more phases can be added as in Figure 16 below. In this figure, Phase 5 is also modified to accommodate the other code changes. The authors feel obliged to admit that, to us, the resulting code is far from obvious, and that it was not easily obtained. Rather, many iterations involving exhaustive phase transition tests and group code review were required to arrive at these results. This process also resulted in minor modifications to our original natural language definition of depth first search so as to arrive at the formulation presented in Figure 11 of this report. Construction of a control flow graph (state graph) for various versions of mission orders was found to be very helpful during this debugging process. Figure 17 shows such a graph for the code of Figure 16. Evidently, this graph provides a more conventional description of control flow during code execution. However, it is not executable, and therefore not subject to exhaustive testing analogous to Figure 7. Its value, therefore, for the purposes of this report, is merely to aid in human understanding of the corresponding Prolog mission orders.

In reviewing and testing the following code, it should be noted that queries have been tagged with their associated state numbers. This technique proved to be valuable in finding coding and logic errors by tracing execution results as in Figure 18.

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive Refinement/
;Mission Orders Archive/stand-alone-human-depth-first-search.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: 5 January 2012.

;This code can be executed only if it is first saved in /My Documents/Tech Reports/
;Recursive Refinement/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/Tech Reports/Recursive Refinement.

;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) (change_phase 1 1.1))

(<- (execute_phase 1.1) (command "Initialize Area A search")
    (ask "Initialization completed-1.1" ?A) (affirmative ?A) (change_phase 1.1 1.2))
(<- (execute_phase 1.1) (change_phase 1.1 'mission_abort) (report "Depth first search failed"))

(<- (execute_phase 1.2) (command "Move forward") (ask "Success-1.2" ?A) (affirmative ?A)
    (change_phase 1.2 1.3))
(<- (execute_phase 1.2) (change_phase 1.2 1.4))

(<- (execute_phase 1.3) (command "Observe environment and test goal found")
    (ask "Goal found-1.3" ?A) (affirmative ?A) (change_phase 1.3 'mission_complete)
    (report "Depth first search succeeded"))
(<- (execute_phase 1.3) (change_phase 1.3 1.2))

(<- (execute_phase 1.4) (command "Backtrack search")
    (ask "Backtrack successful-1.4" ?A) (affirmative ?A) (change_phase 1.4 1.5))
(<- (execute_phase 1.4) (change_phase 1.4 'mission_abort) (report "Depth first search failed"))

(<- (execute_phase 1.5) (command "Observe environment and test for available cell")
    (ask "Available cell-1.5" ?A) (affirmative ?A) (change_phase 1.5 1.2))
(<- (execute_phase 1.5) (change_phase 1.5 1.4))
```

Figure 16. Five-Phase Standalone Mission Orders for Human Agent Execution of Grid-Based Depth First Search

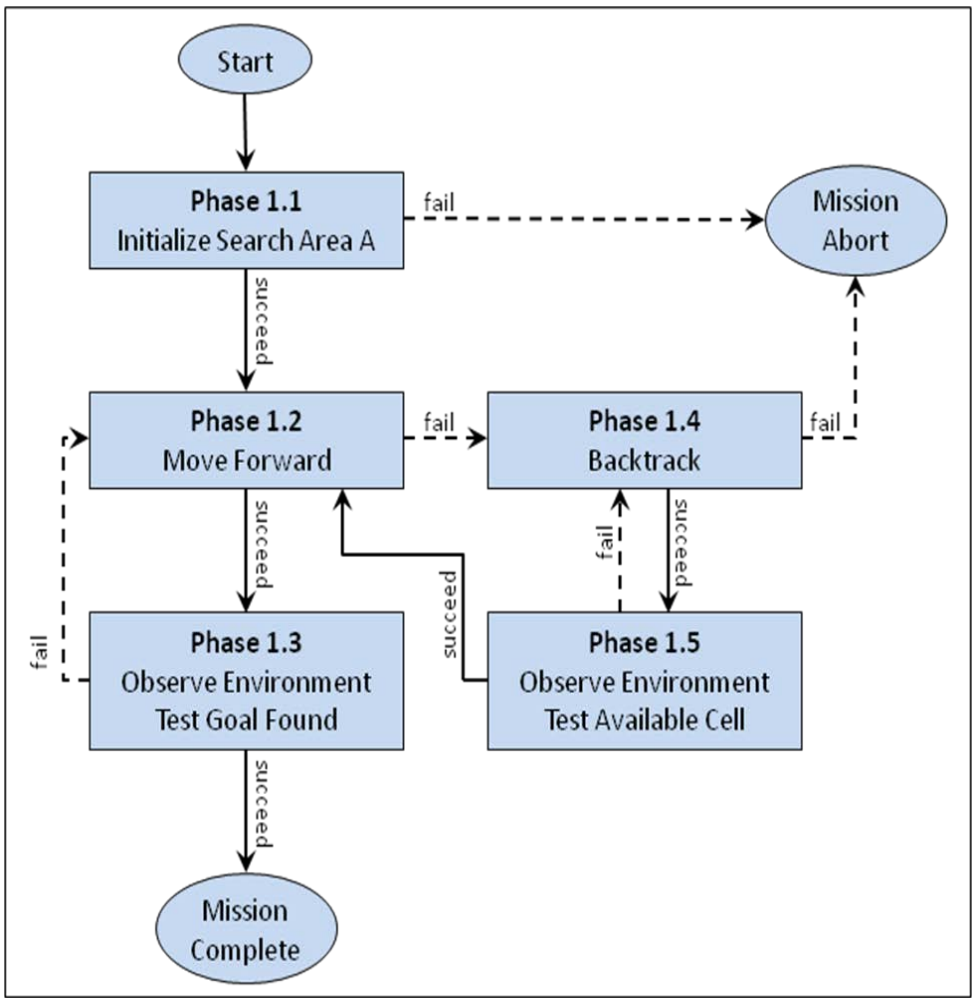

Figure 17. State Graph for Five-Phase Grid-Based Depth First Search Mission Orders

Figure 18 below shows the results of exhaustive testing of *state transitions* for the code of Figure 16.

```
CG-USER(1): (tm)
Initialize Area A search!
Initialization completed-1.1?y
Move forward!
Success-1.2?y
Observe environment and test goal found!
Goal found-1.3?n
Move forward!
Success-1.2?n
Backtrack search!
Backtrack successful-1.4?y
Observe environment and test for available cell!
Available cell-1.5?n
Backtrack search!
Backtrack successful-1.4?n
Depth first search failed.
Yes

No.
```

```
CG-USER(2): (tm)
Initialize Area A search!
Initialization completed-1.1?y
Move forward!
Success-1.2?y
Observe environment and test goal found!
Goal found-1.3?n
Move forward!
Success-1.2?n
Backtrack search!
Backtrack successful-1.4?y
Observe environment and test for available cell!
Available cell-1.5?y
Move forward!
Success-1.2?y
Observe environment and test goal found!
Goal found-1.3?y
Depth first search succeeded.
Yes

No.

CG-USER(3): (tm)
Initialize Area A search!
Initialization completed-1.1?n
Depth first search failed.
Yes

No.
```

Figure 18. State Transition Test for Five-Phase Depth First Search Mission Orders

Now a difficult question arises, namely, are these results correct? That is, do they embody the strategy of Figure 11 and do they always result in a search that halts with a correct conclusion? As shown earlier in this report, for the given top level mission orders, this question is answered in a relatively straightforward way by exhaustive testing of all possible tactical officer response sequences. This is possible in this case because the mission orders define an FSM with just four states. In the present case, however, due to the utilization of two lists of potentially unlimited length (**path-to-goal** and **virtual-obstacle-list**), the mission orders of Figure 16 define instead a type of TM [2, 12]. It is known that there can be no general algorithm for proof of correctness of TMs [11]. Nevertheless, the authors believe that Figure 16 correctly encodes depth first search. As a first step in proving this contention, observe that Figure 18 displays results for all possible state transitions defined in Figure 16. To see that this is so, consider first the results listed above following the prompts "GC-USER(1):" and "GC-USER(2):." Careful examination of the user responses in this section of this figure shows that all possible query responses have been invoked except for a "no" answer to queries from Phase 1.1. Results following "GC-USER(3):" include this case. It is the authors' opinion that all of

these responses are in accordance with the algorithm of Figure 11, and represent physically possible occurrences in the specified depth first search problem.

Despite the above assertion that all state transitions specified in Figure 16 are as desired, it does not follow that this rule set is *complete* and *correct*. That is, no argument has yet been made here that, when using these rules, user response sequences of any length always lead to correct mission execution for any terrain. However, such an argument can be made as follows. First of all, all robot movement is accomplished by Phases 1.2 and 1.4 in this code. Specifically, execution of Phase 1.2 accomplishes a forward move into a previously unexplored terrain cell (providing that such a cell exists). If this fails, repeated execution of Phase 1.4 results in backtracking until some other unexplored cell (if any) is encountered. This being the case, the robot executing this code explores exactly one new cell on each cycle until it halts with either success or failure. If success occurs, then the goal has been found as desired. If the goal is not found, then the entire (bounded) search space has been explored without success, and the process halts and indicates failure. In either case, a correct result is obtained without the inefficiency of exploring the same cell multiple times as typically occurs in Brownian motion search.

The authors contend that the above analysis amounts to an informal proof of correctness for the given mission orders. A formal proof based on predicate calculus [12] is beyond the scope of this report.

V. REFINEMENT OF TOP LEVEL MISSION ORDERS FOR HUMAN EXTERNAL AGENTS

What we wish to do now is to refine the "Search Area A" command using the "stand alone" code of Figure 16. This is accomplished by appropriately "splicing" the code of this figure into that of Figure 6. That is, the code of Figure 16 was developed and tested as if it were a top level mission specification. In fact, it is intended to represent the first stage of recursive refinement of the "Search Area A" command, and hence such a splicing (or some other form of function call) is appropriate to achieving this objective. The result of this change is listed in Figure 19.

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive Refinement
;/Mission Orders Archive/refined-area-search-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: 24 January 2012.

;This code can be executed only if it is first saved in /My Documents/Tech Reports/
;Recursive Refinement/Mission Orders Archive/refined-area-search-mission.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/Tech Reports/Recursive Refinement
/Mission Orders Archive.

;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))))


;------------------------------Top Level Mission Specification------------------------------

(<-- (execute_phase 1) (change_phase 1 1.1))

(<- (execute_phase 2) (command "Sample environment") (ask "Sample obtained" ?A)
    (affirmative ?A)
    (change_phase 2 34))
(<- (execute_phase 2) (change_phase 2 5))

(<- (execute_phase 34) (command "Attempt Area B search")
    (command "Attempt rendezvous with UUV2")
    (change_phase 34 5))

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



;-----------------------------------Phase 1 Refinement-----------------------------------

(<- (execute_phase 1.1) (command "Initialize Area A search")
    (ask "Initialization completed-1.1" ?A) (affirmative ?A) (change_phase 1.1 1.2))
```

```
(<- (execute_phase 1.1) (change_phase 1.1 34) (report "Area A search failed"))

(<- (execute_phase 1.2) (command "Move forward") (ask "Success-1.2" ?A) (affirmative ?A)
    (change_phase 1.2 1.3))
(<- (execute_phase 1.2) (change_phase 1.2 1.4))

(<- (execute_phase 1.3) (command "Observe environment and test goal found")
    (ask "Goal found-1.3" ?A) (affirmative ?A) (report "Search Area A succeeded")
    (change_phase 1.3 2))
(<- (execute_phase 1.3)(change_phase 1.3 1.2))

(<- (execute_phase 1.4) (command "Backtrack search") (ask "Backtrack successful-1.4" ?A)
    (affirmative ?A) (change_phase 1.4 1.5))
(<- (execute_phase 1.4) (change_phase 1.4 34)
    (report "Backtrack failed.  Area A search failed"))

(<- (execute_phase 1.5) (command "Observe environment and test for available cell")
    (ask "Available cell-1.5" ?A) (affirmative ?A) (change_phase 1.5 1.2))
(<- (execute_phase 1.5) (change_phase 1.5 1.4))
```

Figure 19. Area Search and Sample Human Agent Mission Orders with Depth First Search Refinement of Phase 1

Examples of testing of this code are presented in Figure 20, which illustrates human execution of a depth first search.

```
CG-USER(1): (tm)
Initialize Area A search!
Initialization completed-1.1?y
Move forward!
Success-1.2?y
Observe environment and test goal found!
Goal found-1.3?n
Move forward!
Success-1.2?n
Backtrack search!
Backtrack successful-1.4?y
Observe environment and test for available cell!
Available cell-1.5?y
Move forward!
Success-1.2?y
Observe environment and test goal found!
Goal found-1.3?y
Search Area A succeeded.
Sample environment!
Sample obtained?y
Attempt Area B search!
Attempt rendezvous with UUV2!
Return to base!
At base?n
Mission failed.
Yes

No.

CG-USER(2): (tm)
Initialize Area A search!
Initialization completed-1.1?y
Move forward!
Success-1.2?y
Observe environment and test goal found!
Goal found-1.3?n
Move forward!
Success-1.2?n
Backtrack search!
Backtrack successful-1.4?y
Observe environment and test for available cell!
Available cell-1.5?n
Backtrack search!
```

```
Backtrack successful-1.4?y
Observe environment and test for available cell!
Available cell-1.5?n
Backtrack search!
Backtrack successful-1.4?n
Backtrack failed.  Area A search failed.
Attempt Area B search!
Attempt rendezvous with UUV2!
Return to base!
At base?y
Mission succeeded.
Yes

No.
```

Figure 20. Sample Test Results for Human Execution of Area Search and Sample Mission with Refinement of Phase 1

Referring to the above figure, it can be seen that two complete mission scenarios have been tested. The first of these is for the case the "Search Area A" phase ends in success, the other is for the case of a failed search. In both cases, the exit from Phase 1 is correct, and the rest of the mission executes as expected. Since the code of Figures 6 and 16 have each separately been shown to be correct, the results presented in Figure 20 prove the combined code of Figure 19 to be correct. This is in contrast to direct exhaustive testing of the combined code, which involves far too many choices (potentially infinite) to be realistically accomplished through human interaction. That is, *it is neither practical nor necessary to complete Figure 20 to include all possible mission scenarios*. The solution to this type of problem, in general, is to recursively refine commands (as we have done above), with not more than five or six phases in each refinement, and with exhaustive testing of each refinement, until only behaviors previously defined at the tactical level are called (basis condition). The next section of this report deals with the alternative possibility of abstracting (subsuming) the previously defined Brownian motion search code to accomplish depth first search at the tactical level, thereby obviating the need for strategic level command refinement to accomplish this behavior.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. ITERATIVE ABSTRACTION OF TACTICAL LEVEL CODE TO ACHIEVE AUTONOMOUS AREA SEARCH

While the results of Figure 20 prove the correctness of the refined strategic level code for the given mission, there are still some problems remaining. First of all, the tactical level Brownian motion Lisp code of Figure 9 does not contain any functions corresponding to the either a command to *move forward* or to *backtrack search*. This means that either another (therefore "recursive") refinement of the code of Figure 19 to include a definition of these behaviors is required, or else that the code of Figure 9 be augmented by further function definitions to provide the functionality needed to enable robot search of Area A. Such *iterative abstraction* of this code corresponds to the usual idea of *behavior subsumption* [14]. An appropriate revised "abstracted" version of the referenced Lisp code, providing these behaviors, and also capable of performing depth first search on its own, is listed below. It should be noted that this abstraction also provides a definition of Area B terrain.

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive Refinement/
;subsumption-depth-first-tactical.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: January 24 2012.

;This search method implements depth first search in a bounded obstacle field.  Location
;of goal is not known a priori.

(load "C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive
Refinement/brownian-motion-search.fasl")


;----------------------------------Environment and Search State------------------------------

(defvar *area-B* (make-array '(8 8)
   :initial-contents '((1 1 1 1 1 1 1 1) ; An entry of 1 denotes an obstacle.
                       (1 0 0 0 0 0 0 1)
                       (1 0 0 0 0 0 0 1)
                       (1 0 0 0 0 0 0 1)
                       (1 0 0 0 0 0 0 1)
                       (1 0 0 0 0 0 0 1)
                       (1 0 0 0 0 0 0 1)
                       (1 1 1 1 1 1 1 1))))



;---------------------------------------Tactical Level-----------------------------------------

(defun initialize-search (start search-area goal-location)
  (setf *path-to-goal* nil *virtual-obstacle-list* nil *robot-location* start
        *terrain* search-area *goal-location* goal-location ))

(defun legal-move-list (location)
  (do* ((outlist nil (cons (legal-movep (first inlist)) outlist))
        (inlist (possible-move-list location) (rest inlist)))
```

```
       ((null inlist) (remove nil outlist))))

(defun legal-movep (location)
  (if (and (not (member location *virtual-obstacle-list* :test #'equal))
           (not (member location *path-to-goal* :test #'equal))) location))

(defun move-forward-from (location)
  (if (legal-move-list location) (and (push location *path-to-goal*)
                                      (go-to (random-select (legal-move-list location))))))

(defun backtrack-search-from (location)
  (if *path-to-goal* (and (push location *virtual-obstacle-list*)
                          (setf *robot-location* (pop *path-to-goal*)))))

(defun available-cellp (location) (legal-move-list location))

(defun move-from (location)
  (if (available-cellp location) (move-forward-from location)
                                 (backtrack-search-from location)))

(defun depth-first-search (start search-area goal-location)
  (initialize-search start search-area goal-location)
  (do* ((location *robot-location* (move-from location)))
       ((or (equal location *goal-location*) (null location))
        (if (equal location *goal-location*)
            (pprint (reverse (push location *path-to-goal*)))))))



;---------------------------------------Test Functions-------------------------------------


(defun test5 () (depth-first-search '(2 2) *area-A* '(3 1))) ;Isolated goal.

(defun test6 () (depth-first-search '(3 3) *area-A* '(3 1))) ;Immobilized start.

(defun test7 () (depth-first-search '(1 3) *area-A* '(2 5))) ;Short path.

(defun test8 () (depth-first-search '(3 6) *area-A* '(6 2))) ;Long path.

(defun test9 () (depth-first-search'(1 3) *area-B* '(3 5))) ;Open area.
```

Figure 21. Revised Tactical Level Lisp Code Including Depth First Search

Representative results from calling the *depth-search* function defined above are presented below in Figure 22. The results shown are evidently correct. In particular, the call to "pretty print" the *virtual obstacle list* shows that, as should happen, the entire available cell search space is filled in the failed attempt to reach the isolated goal involved in *test5*, and the robot ends up at the start location. Moreover, while as expected, the paths found in other cases are shorter than those shown in Figure 10 for Brownian motion search, they also confirm (from the two calls to *test8*) that depth first search does not always return a minimum length path. In order to assure the latter, a more complex form of search involving either an a priori map or the cooperation of multiple agents is needed. An example of such a search method is *breadth first* search [11] in which a reproducing and expanding group of agents moves a *frontier* forward

until the goal is found. Evidently, this strategy is not suitable for a search by single UUV as envisaged in this report.

```
CG-USER(1): (test5)
NIL

CG-USER(2): *robot-location*
(2 2)

CG-USER(3): (pprint *virtual-obstacle-list*)
((1 2) (1 3) (1 4) (1 5) (2 5) (2 6) (3 6) (3 5) (4 6) (5 6) (5 5) (6 5) (6 4) (6 3) (5 3)
 (5 2) (6 2) (6 1) (1 1))

CG-USER(4): (test6)
NIL

CG-USER(5): (test7)
((1 3) (1 4) (1 5) (2 5))

CG-USER(6): (test8)
((3 6) (4 6) (5 6) (5 5) (6 5) (6 4) (6 3) (5 3) (5 2) (6 2))

CG-USER(7): (test8)
((3 6) (4 6) (5 6) (5 5) (6 5) (6 4) (6 3) (6 2))

CG-USER(8): (test9)
((1 3) (1 2) (1 1) (2 1) (3 1) (4 1) (5 1) (5 2) (6 2) (6 3) (6 4) (5 4) (5 5) (4 5) (4 4)
 (4 3) (4 2) (3 2) (2 2) (2 3) (3 3) (3 4) (2 4) (1 4) (1 5) (1 6) (2 6) (3 6) (3 5))

CG-USER(9): (test9)
((1 3) (2 3) (2 4) (1 4) (1 5) (2 5) (3 5))
```

Figure 22. Representative Results from Lisp Depth First Search

The results of the two calls to *test9* listed above show that depth first search is a poor strategy for searching for a goal in an unknown location in an open area, since the paths found are *much* longer than the shortest path available for this example. A detailed treatment of some more suitable search strategies for low obstacle density terrain can be found in [15].

While the results of Figure 22 are believed to be correct, there is still a problem with respect to simulation of the selected UUV mission in that *none* of the Prolog code presented thus far in this report calls any tactical level Lisp functions, but rather depends entirely on human interaction during execution. That is, all of the Prolog code so far included in this report is useful only for “Turing test” debugging [1, 10]. This problem is addressed in following sections of this report.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. EXAMPLE MISSION ORDERS MODIFIED FOR HYBRID HUMAN/ROBOT AGENT EXECUTION

One of the strengths of the recursive refinement approach to code development in the RBM architecture is that shifting of responsibility for tactical level mission execution can be accomplished incrementally during debugging from a human tactical officer to a robot (software) tactical officer. With respect to the refined area search and sample mission orders of Figure 19, the Lisp functions developed in Figure 21 are available for this purpose. While such a gradual shifting of responsibility during code development is a tedious process, it has the great advantage of allowing stepwise verification of correctness. Obviously, the human tactical officer not only answers some strategic level queries during this process, but also functions as the *test agent* at each stage of code development.

In the code of Figure 23, the robot execution of Phase 1 is commanded directly from the given tactical level mission orders, while the human tactical officer receives status reports from the robot to permit verification of correct execution. He himself then interactively controls the execution of the remaining commands and queries arising in carrying out the given mission. It should be noted that participation by the robot in carrying out this mission is made possible by the addition below of *robot external agent communication functions* to the human communication functions previously provided in Figure 1.

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive Refinement/
;Mission Orders Archive/hybrid-agent-depth-first-tactical.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: 24 January 2012.

(require :prolog) (shadowing-import '(prolog:==)) (use-package :prolog) ;Start Prolog.
(load "C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive
Refinement/subsumption-depth-first-tactical.fasl")

;Robot external agent communication functions

(<-- (initialize_area_A_search)
     (report "Robot report: Area A search initialization in progress")
     (lisp (initialize-search '(1 3) *area-A* '(2 5))))
(<-- (move_forward) (is ?x (move-forward-from *robot-location*)) (princ ?x) !  (affirmative
?x))
(<-- (goal_found) (is ?x *robot-location*) (is ?y *goal-location*) !  (== ?x ?y))
(<-- (backtrack) (is ?x (backtrack-search-from *robot-location*)) (princ ?x) !
     (affirmative ?x))
(<-- (available_cell_found) (is ?x (available-cellp *robot-location*)) !  (affirmative ?x))
```

```
;Utility functions

(defun random-failure (n) (if (zerop (random n)) T)); Failure 1 out of n times on average.

(<-- (change_phase ?old ?new) (retract ((current_phase ?old)))
     (asserta ((current_phase ?new))))
(<-- (success ?n) (is ?x (random-failure ?n)) (== ?x nil))


;------------------------------Top Level Mission Specification-------------------------------

(<-- (execute_phase 1) (change_phase 1 1.1))

(<- (execute_phase 2) (command "Sample environment") (ask "Sample obtained" ?A)
    (affirmative ?A) (change_phase 2 34))
(<- (execute_phase 2) (change_phase 2 5))

(<- (execute_phase 34) (command "Attempt Area B search")
    (command "Attempt rendezvous with UUV2") (change_phase 34 5))

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


;---------------------------------------Tactical Level---------------------------------------

;Area A search rules

(<- (execute_phase 1.1) (initialize_area_A_search) (phase_completed 1.1))
(<- (phase_completed 1.1) (success 3)
    (report "Robot report: Search Area A initialization succeeded")
    (change_phase 1.1 1.2))
(<- (phase_completed 1.1)
    (report "Robot report: Search Area A initialization failed.  Phase execution aborted")
    (change_phase 1.1 34))

(<- (execute_phase 1.2) (move_forward) (not (== ?x nil))
    (report "Robot report: Move forward succeeded") (change_phase 1.2 1.3))
(<- (execute_phase 1.2) (report "Robot report: Move forward failed") (change_phase 1.2 1.4))

(<- (execute_phase 1.3) (goal_found) (report "Robot report: Search Area A succeeded")
    (change_phase 1.3 2))
(<- (execute_phase 1.3) (report "Robot report: Goal not found") (change_phase 1.3 1.2))

(<- (execute_phase 1.4) (backtrack) (report "Robot report: Backtrack succeeded")
    (change_phase 1.4 1.5))
(<- (execute_phase 1.4) (change_phase 1.4 34) (report "Robot report: Search Area A failed"))

(<- (execute_phase 1.5) (available_cell_found) (report "Robot report: Available cell found")
    (change_phase 1.5 1.2))
(<- (execute_phase 1.5) (report "Robot report: No available cell found")
    (change_phase 1.5 1.4))
```

Figure 23. Mission Orders Modified for Human/Robot Cooperative Execution

Figure 24 below shows that, using the above code, the UUV correctly carries out Phase 1 autonomously, while other phases of the selected mission are accomplished by human interaction. It should be noted that on the first attempt at mission execution shown in this figure, initialization fails, and this failure is reported to the human tactical officer. This is because of the inclusion of a random *success* predicate in the *(execute_phase 1.1)* definition in Figure 23 to exercise both the success and failure branches of this phase. It should also be noted that in the last example in this figure, the

user chose to terminate the simulated mission execution by simply typing a .”” (dot) symbol in response to a query. This is a convenient built in feature of Allegro Prolog.

```
CG-USER(1): (tm)
Robot report: Area A search initialization in progress.
Robot report: Search Area A initialization failed.  Phase execution aborted.
Attempt Area B search!
Attempt rendezvous with UUV2!
Return to base!
At base?y
Mission succeeded.
Yes

No.

CG-USER(2): (tm)
Robot report: Area A search initialization in progress.
Robot report: Search Area A initialization succeeded.
(1 2)Robot report: Move forward succeeded.
Robot report: Goal not found.
(2 2)Robot report: Move forward succeeded.
Robot report: Goal not found.
NILRobot report: Move forward failed.
(1 2)Robot report: Backtrack succeeded.
Robot report: Available cell found.
(1 1)Robot report: Move forward succeeded.
Robot report: Goal not found.
NILRobot report: Move forward failed.
(1 2)Robot report: Backtrack succeeded.
Robot report: No available cell found.
(1 3)Robot report: Backtrack succeeded.
Robot report: Available cell found.
(1 4)Robot report: Move forward succeeded.
Robot report: Goal not found.
(1 5)Robot report: Move forward succeeded.
Robot report: Goal not found.
(2 5)Robot report: Move forward succeeded.
Robot report: Search Area A succeeded.
Sample environment!
Sample obtained?y
Attempt Area B search!
Attempt rendezvous with UUV2!
Return to base!
At base?n
Mission failed.
Yes

No.

CG-USER(3): (tm)
Robot report: Area A search initialization in progress.
Robot report: Search Area A initialization succeeded.
(1 2)Robot report: Move forward succeeded.
Robot report: Goal not found.
(1 1)Robot report: Move forward succeeded.
Robot report: Goal not found.
NILRobot report: Move forward failed.
(1 2)Robot report: Backtrack succeeded.
Robot report: Available cell found.
(2 2)Robot report: Move forward succeeded.
Robot report: Goal not found.
NILRobot report: Move forward failed.
(1 2)Robot report: Backtrack succeeded.
Robot report: No available cell found.
(1 3)Robot report: Backtrack succeeded.
Robot report: Available cell found.
(1 4)Robot report: Move forward succeeded.
Robot report: Goal not found.
(1 5)Robot report: Move forward succeeded.
Robot report: Goal not found.
(2 5)Robot report: Move forward succeeded.
Robot report: Search Area A succeeded.
```

```
Sample environment!
Sample obtained?n
Return to base!
At base?y
Mission succeeded.
Yes

No.

CG-USER(4): (tm)
Robot report: Area A search initialization in progress.
Robot report: Search Area A initialization succeeded.
(1 4)Robot report: Move forward succeeded.
Robot report: Goal not found.
(1 5)Robot report: Move forward succeeded.
Robot report: Goal not found.
(2 5)Robot report: Move forward succeeded.
Robot report: Search Area A succeeded.
Sample environment!
Sample obtained?.
Error: Dot context error.  [file position = 4287]
[condition type: READER-ERROR]
```

Figure 24. Example Joint Mission Execution by Human and Robot Tactical Officers

In examining the above results, it should be noted that the Prolog-based Area A search problem solved above corresponds exactly to the *test7* Lisp function call of Figure 22. As they should, results agree. Specifically, in the first example, Area A search initialization failed, so Area A search also fails, and Phase 1 execution is aborted. In the next two cases, as can be seen from the robot position values returned, the robot first moves to the left, then backtracks through the start position at (1, 3), and then continues moving forward until reaching the goal at (2, 5). In the last case, the robot first moves to the right, and so finds the goal without backtracking.

Evidently, the transfer of responsibilities from human to robot could be continued until the human tactical officer functions only as an observer. When this has been done, the mission is ready for replacement of the simplified execution level functions utilized in this report by real robot execution level function calls, and further testing by simulation using realistic UUV dynamics [16]. When this phase has been accomplished, in water testing can commence [4, 17].

Before proceeding further, it should be noted that, if a suitable communication link is available then, if desired, as was done above, a remotely located human tactical officer could continue to serve as an external agent to an MEA mission controller *simultaneously* with the robot vehicle external agent in real time mission execution. This is the method used at present by aerial *drones* and subsea ROVs. This approach also provides an *evolutionary path* for the development of fully autonomous mobile robots. If

the human tactical officer is retained, then MEA mission orders could potentially put constraints on his actions to ensure that the robot responds only to “ethical” commands from such a *ground controller*. We intend to explore this possibility in a future technical report.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. FULLY AUTONOMOUS MISSION SOURCE CODE AND EXECUTION LOGS

Up to this point in this report, we have been gradually moving from missions in which the response to strategic level commands is provided entirely by a human tactical officer toward missions increasingly based on robotic execution. We also have used subsumption of lower level behaviors by higher order behaviors to permit a smaller degree of mission goal refinement. The limit of this evolution toward autonomous execution is reached when the role of the human tactical officer is reduced to *observation* and *evaluation*. Figure 25 below provides one example of code for fully autonomous (robotic) execution of the given area search and sample mission.

```
;C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive Refinement/
;Mission Orders Archive/robot-agent-random-success.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: 31 October 2011.

(require :prolog) (shadowing-import '(prolog:==)) (use-package :prolog) ;Start Prolog.
(load "C:/Documents and Settings/mcghee/My Documents/Tech Reports/Recursive
Refinement/subsumption-depth-first-tactical.fasl")

;Robot external agent communication functions

(<-- (search_area_A) (success 2))
(<-- (sample_environment) (success 2))
(<-- (search_area_B) (success 2))
(<-- (rendezvous_UUV2) (success 2))
(<-- (return_to_base) (success 2))

;Utility functions

(defun random-failure (n) (if (zerop (random n)) T)); Failure 1 out of n times on average.

(<-- (change_phase ?old ?new) (retract ((current_phase ?old)))
     (asserta ((current_phase ?new))))
(<-- (success ?n) (is ?x (random-failure ?n)) (not (== ?x nil)))


;--------------------------------Top Level Mission Specification------------------------------

(<-- (execute_phase 1) (command "Search Area A") (search_area_A)
     (report "Robot report: Search Area A succeeded") (change_phase 1 2))
(<- (execute_phase 1) (report "Robot report: Search Area A failed") (change_phase 1 3))

(<- (execute_phase 2) (command "Sample environment") (sample_environment)
    (report "Robot report: Sample environment succeeded") (change_phase 2 3))
(<- (execute_phase 2) (report "Robot report: Sample environment failed") (change_phase 2 5))

(<- (execute_phase 3) (command "Search Area B") (search_area_B)
    (report "Robot report: Search Area B succeeded") (change_phase 3 4))
(<- (execute_phase 3) (report "Robot report: Search Area B failed") (change_phase 3 4))

(<- (execute_phase 4) (command "Rendezvous UUV2") (rendezvous_UUV2)
    (report "Robot report: Rendezvous UUV2 succeeded") (change_phase 4 5))
(<- (execute_phase 4) (report "Robot report: Rendezvous UUV2 failed") (change_phase 4 5))

(<- (execute_phase 5) (command "Return to base") (return_to_base)
```

```
    (change_phase 5 'mission_complete) (report "Robot report: Return to base succeeded")
    (report "Mission succeeded"))
(<- (execute_phase 5) (change_phase 5 'mission_abort)
    (report "Robot report: Return to base failed") (report "Mission failed"))
```

Figure 25. Mission Orders for Simulation of Fully Autonomous "Area Search and Sample" Mission Execution

Examples of execution of the code from Figure 25 are depicted below in Figure 26.

```
CG-USER(1): (tm)
Search Area A!
Robot report: Search Area A succeeded.
Sample environment!
Robot report: Sample environment succeeded.
Search Area B!
Robot report: Search Area B succeeded.
Rendezvous UUV2!
Robot report: Rendezvous UUV2 succeeded.
Return to base!
Robot report: Return to base succeeded.
Mission succeeded.
Yes

No.

CG-USER(2): (tm)
Search Area A!
Robot report: Search Area A failed.
Search Area B!
Robot report: Search Area B succeeded.
Rendezvous UUV2!
Robot report: Rendezvous UUV2 failed.
Return to base!
Robot report: Return to base failed.
Mission failed.
Yes

No.

CG-USER(3): (tm)
Search Area A!
Robot report: Search Area A succeeded.
Sample environment!
Robot report: Sample environment failed.
Return to base!
Robot report: Return to base succeeded.
Mission succeeded.
Yes

No.

CG-USER(4): (tm)
Search Area A!
Robot report: Search Area A succeeded.
Sample environment!
Robot report: Sample environment failed.
Return to base!
Robot report: Return to base failed.
Mission failed.
Yes

No.
```

Figure 26. Examples of Simulated Execution of Fully Autonomous Mission Code

Inspection of Figure 25 shows that no actual methods have been provided for execution of strategic level commands. Rather, the robot communication functions defined for this purpose constitute merely *stub* functions with a random probability of success or failure. These must be replaced with calls to defined tactical level behaviors for a real vehicle before execution in an actual in water test can occur. In such a test, area search could be accomplished either by rule-based depth first search as in Figure 23, or by a subsumption behavior as in Figure 21, or by some entirely different algorithm as in [15]. However, despite these alternatives, it is important to realize that this code is nevertheless *generic* for the given area search and sample mission, and could be used with no modification other than to replace the above "stubs" with real calls to defined vehicle behavior functions (in any language) for any physical vehicle capable of responding to the given strategic level commands.

Another feature of the above code is that it is based on the original five-state code for this mission, and not on the reduced four state version. This is because the authors believe that is easier to read the autonomous version of this code in this form. That is, while a human can be expected to understand that a sequence of unconditional commands (as in Figure 20) is to be carried out in the order issued, it is better to use the five-state orders to make this explicit to a robot, at least for code testing.

Consideration of the test cases of Figure 26 shows that only four examples are listed while there are eighteen possibilities. Since, for the code of Figure 25, success or failure of any mission phase is random, with an equal chance of either outcome on any given case of mission execution, a very large number of trials are likely to be needed before all eighteen possible response sequences are observed. We argue however, that this is not necessary. Instead, since the correctness of the basic coding of the given mission has already been verified by exhaustive human testing, all that matters is that all of the state transitions associated with the code of Figure 25 have been observed to be correct. This is true (by chance) of Figure 26. Specifically, the first case appearing in Figure 1 presents a correct outcome for the success of each of the five mission phases. The next three cases show correct behavior for every possible phase failure. Thus no further demonstration of the overall correctness of the code of Figure 25 is needed. This being the case, Figures 25 and 26 demonstrate that the fundamental objective of this

report has been attained with respect to the given area search and sample mission. That is, a validated computer simulation of this mission in a form suitable for transformation to real time embedded UUV code has been presented and tested.

As a final remark, it is important to recognize that while Prolog and Lisp provide convenient languages for simulation studies, they may or may not provide the best basis for realizing a MEA in a real time system. This is a topic for further research. For now, we have previously presented an MEA realization in Java together with XML [15, 16]. We are also hopeful that a programming language based on a finite state machine overlay on subsumption based code can also be used for this purpose [14]. No doubt further research will reveal still more possibilities, and this is an important line of work to follow for achieving clearly defined, unambiguously executed, interoperable robot missions.

IX. HOW TO EXECUTE CODE IN THIS REPORT

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 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 Figure 7 and similar figures in this report, should produce the indicated results.

Of course the *load* function calls in your code should be modified to match your file structure. As a step toward such modification, the authors advocate that load commands first be commented out, and manual loading be used instead, from the bottom up in terms of file dependencies. The correct path to each file can then be noted from the legend appearing at the top of an editor pane, and subsequently used to edit *load* function arguments.

THIS PAGE INTENTIONALLY LEFT BLANK

X. SUMMARY AND CONCLUSIONS

A. SUMMARY OF RESULTS OBTAINED

This report is concerned with the development of a methodology for overlaying a software-based field programmable mission controller onto the real-time control software of an existing autonomous mobile robot. The flexibility desired in mission control is analogous to that of a conventional manned submarine in carrying out a set of formal written orders. A key issue in our work is finding a way to write mission orders so that they can be read and executed in simulation form both by a human mission expert, and by a digital computer, *without recoding*. We present such a methodology in this report using Prolog and Lisp together to achieve a working simulation model. We also describe how to use such a simulation model as executable specifications for writing real time code for control of a physical robot. In order to illustrate our ideas and mathematical models, we develop our study in the context of mission control for a UUV.

The report begins in a top down fashion with a brief review of the mathematical concept of a generalization of a Turing Machine (TM) called a *Mission Execution Automaton* (MEA). A Prolog implementation of a universal MEA mission controller called a *Mission Execution Engine* (MEE), along with a specific set of *mission orders* is presented. These orders are for the "area search and sample" mission previously studied extensively at our institution [1]. We then observe that each phase of this mission corresponds to a state of a finite state machine, and that a form of *state reduction* can be applied to this machine resulting in a simplified testing protocol. Such tests can prove the correctness of a given set of orders through exhaustive testing of all possible event sequences. This, of course, requires that the mission orders be loop free. We complete and evaluate such a simplification and subsequent testing for the given mission.

To provide a concrete example of a system to which an MEA might be added, we next define a (greatly simplified) vehicle and associated terrain. Specifically, the "terrain" consists of a rectangular planar grid in which every cell is either clear or blocked ("mined"). We then constrain vehicle motion to a strictly north-south or east-west direction, one cell at a time. We also give the simulated vehicle an ability to sense whether or not a given cell contains an obstacle, using a "sonar" system with range

limited to the four neighbors of the current robot location. Starting with these "execution level" capabilities, we then apply a *subsumption* approach to evolving to higher and higher levels of vehicle behavioral capabilities, with each new behavior "subsuming" those that support it. Specifically, using Lisp, we first develop *tactical level* code to enable completely random search of the terrain (Brownian motion) to find a goal in an unknown location in the search space. We observe that this is analogous to the way bacteria find food, and note the inefficiency of this kind of search.

Following the implementation of Brownian motion search, we next turn our attention to refining an *area search* command from the MEA to the tactical level into a more efficient "depth first search." This is accomplished by first defining depth first search in terms of a TM, and then writing tactical level human interactive Prolog code to implement the FSM part of the TM. At this point, the "tape" of this machine resides in the mind of the human tactical officer. To test the logic of this *goal refinement*, we execute several examples to show that the given mission orders correctly implement the state transitions of the specified FSM. Having done this, since exhaustive testing of a TM is generally not possible, we present an informal proof that, in this particular case, when applied to finite search spaces, the machine proposed always halts with a correct result.

In the above development, for the purpose of proving code correctness, depth first search was treated as a top level mission goal. Of course that is not our intent. Rather we wish the code developed to function as a rule-based refinement of the "Search Area A" command issued by the mission controller for the given mission. To accomplish this objective, we introduce a code *splicing* technique that allows the Prolog refinement code to be accessed in a way analogous to a Lisp function call. This permits human interactive execution of an entire area search and sample mission with autonomous depth first search of Area A. Test results for several cases using this code are presented to verify the correctness of code splicing, and therefore of the entire combined code.

Despite the success of the above experiment, at this point all queries are still answered by a human tactical officer rather than by Lisp function calls. This is because the orders from the refined Prolog code do not "reach down far enough." That is, the given Brownian motion code does not define needed behaviors such as "move forward" and "backtrack search." To remedy this situation, additional Lisp code is written to

implement these and other behaviors called by the refined code. Beyond this, a yet higher level function is written to directly execute depth first search. Execution of a number of examples satisfies several necessary conditions for the correctness of this "abstracted" code.

The final goal of this report is achieved by the presentation of validated code for fully autonomous execution of the simulated area search and sample mission. The method used in reaching this goal is to replace human interactive function calls one at a time. This is found to be a highly effective code development technique. Moreover, it is noted that when not all such functions are automated, "human in the loop" *drones* or ROVs result. The possibility exists in such cases that mission orders for such vehicles could be written so that they would refuse to obey illegal orders.

B. CRITIQUE OF EXAMPLE MISSION ORDERS

The "area search and sample" mission used in this report was chosen as a somewhat arbitrary example of a prototypical UUV mission. It leaves out many details that would be required in a real mission. For example, it does not include an explicit *initialization* phase. Rather, initialization is carried out implicitly in the calls to defined tactical level behaviors. Moreover, and perhaps more serious, is the omission of an explicit *transit* phase to get from the launch point to Area A. On the other hand, leaving issues such as these to phase refinement at either the strategic or the tactical level results in a top level mission validation test suite involving only six cases. This consideration may override any advantage in mission understanding to be gained by expanding the number of phases at the top of the strategic level. More experience with real missions for real robots is needed to deal realistically with this issue.

The particular implementation of grid-based depth first search used here is surely too crude for a real mission. Specifically, restricting obstacle sensing and vehicle motion to just four directions is unnecessary and unrealistic. It is straightforward to allow eight directions to be considered. Should this prove to be inadequate, an arbitrarily number of search and motion directions can be explored at the cost of an increase in algorithm complexity.

It is important to recognize that the example mission used in this report is configured at the top level in a "fail safe" way. That is, all but one phase can fail without

complete mission failure. This is typical of military mission orders in general, since such orders often are carried out in an adversarial environment and usually involve carefully thought out "contingency" responses. Simple linear "scripts" associated with scientific missions sometimes fail to explicitly address mission phase failure in a fail safe manner. We think that this shortcoming is unnecessary, and believe that it is better, even for non-military missions, to explicitly address alternatives in case of phase failure as we have done here. Much more experience with real vehicle missions is needed to better understand this matter.

C. CONCLUSIONS

The concept of a Mission Execution Automation can be used to enable autonomous operation of robotic vehicles by adding an additional layer of software to deal with mission contingencies, resulting in a *Rational Behavior Model* (RBM) software architecture. Prolog and Lisp can be used together as an effective means for simulating such a system. In particular, the methodology presented in this report provides for a stepwise and smooth transition from a natural human language set of mission orders for a manned system to a fully autonomous real time robotic system, with each stage of software development serving as executable specifications for the next stage. Final installed software need not make use of either Prolog or Lisp, but a few physical experiments in natural environments show that such a choice can be effective [4, 17]. More research, involving real vehicles operating in the physical world is needed to further establish the viability of the RBM approach and to determine the best programming languages for its implementation in various situations.

When goal refinement results in a subordinate FSM definition, then that machine can optionally be considered strategic level code or tactical level code, without compromising the option of accomplishing proof of correctness of mission orders through exhaustive testing. However, when such refinement results in the definition of a TM, as in this report, then the refinement code should be placed at the tactical level, as was done here, in order to preserve the generic *provability* property of strategic level code.

MEA provide a formalism for the extension of TMs to systems capable of controlling mobile robots carrying out complex missions in real time in the physical

word. A similar kind of automaton called a “Markov state-space model” [18] has recently been shown to be effective in modeling and analyzing risks associated with the at sea autonomous operation of long duration UUVs. We believe that it will prove valuable to explore possible synergies between this application of finite state machine theory with those proposed in the present report. We also think that it would be beneficial to relate MEA more completely to FSM, to permit a more formal treatment of issues such as state minimization and proof of correctness. We further believe that such an effort would allow removal of the constraint that mission orders be loop free by including a *time-out* or a *count-out* feature in tactical level functions [2, 17].

D. RECOMMENDATIONS FOR FUTURE WORK

To date, successful in water tests have been carried out with two vehicles using the RBM architecture, involving both Prolog and Java realizations of an MEA [4, 15], and we intend to conduct further experiments with vehicles available at the NPS. Even so, finding additional partners for such investigations is one of our highest research priorities.

In addition to planned in-water UUV testing, experimentation in a virtual environment utilizing the NPS AUV Workbench [16] is ongoing. We are also in the process of developing a complete MEA realization for ground robots using the AUV Workbench and ground robots available at NPS. This inquiry might include adding a suitable open source ISO compliant Prolog compiler to the AUV Workbench code.

Within the realm of simulation studies, we are seeking a “fourth layer” for the RBM architecture that would abstract the UUV software development paradigm advanced in this report to a higher level. In particular, we would like to formally define the role of a *mission specialist* and allow such a person to participate in ROV or drone operations at a level above the MEA mission controller. Moreover, when the role of the mission specialist is well understood, some of his/her knowledge might be incorporated into a *compiler* to automatically generate executable vehicle code from a higher level mission specification. Such a compiler could potentially impose *ethical constraints* on mission orders.

We look forward to dialogue with others interested in the further development FSM methods for mission planning, execution, and analysis for autonomous robotic vehicles and related systems.

LIST OF REFERENCES

1. 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

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

3. Byrnes, R.B., et al., "The Rational Behavior Software Architecture for Intelligent Ships," *Naval Engineers Journal*, pp. 43-55, March, 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. 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. Available at https://savage.nps.edu/AuvWorkbench/website/documentation/papers/papers.html

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

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

8. Graham, P., *ANSI Common Lisp*, Prentice Hall, 1996.

9. Kohavi, Z., and Niraj, K.J., *Switching and Finite Automata Theory*, McGraw Hill, 2010.

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

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

12. Minsky, M.L., *Computation: Finite and Infinite Machines*, Prentice Hall, 1967.

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

14. Woithe, H.C., and Kremer, U., "A Lightweight Scripting Engine for the Slocum Glider" *Proc. IEEE Oceans'11 Conf.*, Kona, Hawaii, September 2011.

15. 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. Available at https://savage.nps.edu/AuvWorkbench/website/documentation/papers/papers.html

16. Brutzman, D.P., "Autonomous Unmanned Vehicle Workbench (AUVW) Rehearsal and Replay: Mapping Diverse Vehicle Telemetry Outputs to Common XML Data Archives," *Proc. of the 15th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August, 2007. Available at https://savage.nps.edu/AuvWorkbench/website/documentation/papers/papers.html

17. 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.

18. Brito, M.P., and Griffith, G., "A Markov Chain State Transition Approach to Establishing Critical Phases for AUV Reliability," *IEEE Journal of Ocean Engineering*, Vol. 36, No. 1, pp. 139-149, January, 2011.

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