11/26/19

Saved in OneDrive/Documents/Publications/Papers/Self-Monitoring/Tech Memo

Technical Memorandum
Naval Postgraduate School
Monterey, CA 93950

Rational Behavior Model (RBM): Achieving Fail-Safe and Ethically Constrained Missions by Human/Robot Teams Using Exhaustively Testable Self-Monitoring Software

Robert McGhee, Don Brutzman, and Curtis Blais

Correspondence: robertbmcghee@gmail.com

1. Introduction

In [1], the authors introduced a formalization of automated manned or unmanned vehicle mission specification and control by means of an extension of a Turing machine (TM) called a "Mission Execution Automaton" (MEA). This formalization, in turn, and the methodology associated with it, derive from an earlier defined software architecture for autonomous robots called the "Rational Behavior Model" (RBM) [2]. Whereas the above publications used the "Prolog" *logic programming* language [3] for top-level mission logic definition and coding, the primary purpose of the present memorandum is to present an implementation of the same architecture in the *functional programming* language "Common Lisp" [4]. Our main reason for doing this is that we have found that many more programmers are acquainted with Lisp than with Prolog, and therefore it should be easier for them to convert Lisp code of the type presented here to their own favorite language, if desired. Another reason for using Lisp is merely to show by example that, while we once thought differently [2], there is nothing special about Prolog as a language for executable specification of mission logic except for the higher level of abstraction it facilitates. The reader can form his/her own opinion on this issue by comparing Appendix A with Appendices B, C, and D of this memorandum. Appendix E shows how to convert Lisp code for human mission execution into code for autonomous robot execution.

The second contribution of this memorandum is the development of the algorithm and code of Appendix F that provides a listing of all possible paths through the prototypical five-state mission flow-graph presented in [1]. This output allows the person or organization authorizing such a mission to validate its correctness in the strongest possible way. That is, the authorizing authority needs no programming skills at all in order to fully understand all possible mission scenarios, and can therefore be held responsible, both morally and legally, for the results of actual mission execution.

The remainder of this memorandum elaborates on the above two topics.

II. Manned and Remotely Operated Missions

A. Lisp Implementation of MEA for Human Mission Execution

Referring to Appendix A, it can be seen that the “search and sample” mission defined by Fig. 7 of [1] is defined in Lisp by making each phase a top-level software object. It can also be seen that, as in the referenced figure, each mission phase has associated with it a command and a ternary-valued outcome relating to command execution. In particular, each phase terminates with “success”, “failure”, or “constraint”. In the latter case, from [1], the meaning of a “constraint” response is that, based on the judgement of a human *external agent* (present, perhaps remotely, during command execution), for ethical reasons, execution time-out, or any other reason, the given mission phase was not allowed to reach completion (or perhaps not even started). This feature, part of the definition of an MEA, is usually not present in mission orders intended to be executed by a human with authority and ability to confer with higher authorities and apply human judgement to such “exceptions”. Rather, MEA are intended to serve as automated mission controllers when this is not the case. Figure 1 below shows typical results obtained in execution of the referenced search and sample mission when directed by the code of Appendix A.

```
CG-USER(1): (run)
Search Area A!
Enter execution outcome as s, f, or c:c
Rendezvous with Vehicle2!
Enter execution outcome as s, f, or c:c
Proceed to recovery!
Enter execution outcome as s, f, or c:c
"Mission aborted."

CG-USER(2): (run)
Search Area A!
Enter execution outcome as s, f, or c:f
Search Area B!
Enter execution outcome as s, f, or c:s
Rendezvous with Vehicle2!
Enter execution outcome as s, f, or c:c
Proceed to recovery!
Enter execution outcome as s, f, or c:s
"Mission complete."
```

Fig. 1. Typical Results of Mission Execution Using a Human Being to Respond to MEA Commands/Queries. (Bold font applied after execution to highlight operator input. The symbol “s” stands for “Success”, “f” for “Failure, and “c” for “Constraint.)

Examining the first trial of Fig. 1 above, and comparing to the first example of Fig. 8 of [1], it can be seen that these two results agree. On the other hand, [1] contains another result not included above, while the second result above is not included in [1]. Thus, three possible response sequences (mission scenarios) have been presented so far here and in [1]. This raises the question: how many distinct response sequences are possible in executing the subject mission? In the case that it is understood that all mission phases must be attempted, regardless of prior outcomes (typical for a fully human-directed mission, sometimes called a *mission script* [1]), there are evidently three outcomes for each phase and five such phases. Thus, the total number of possible outcome sequences is $3^5 = 243$. Clearly, it is not reasonable to expect a human to correctly execute a simulation of all such cases to demonstrate mission correctness. If this is not apparent, the reader is invited to copy the code of Appendix A below into an

Allegro editor pane, and then attempt to execute all possible cases. Difficult as this may seem, this task is simplified for the subject search and sample mission because, as the three cases discussed above show, not all outcome sequences for this mission are of length 5. In fact, as shown Section V below, due to various “shortcut” paths through the mission flowchart that bypass one or more phases, only 75 outcome sequences are actually possible for this mission. Still, as further discussed in Section V, it is the authors’ view that even 75 cases are too many for reliable manual exhaustive testing, and that the automated method listed in Appendix F is to be preferred.

B. Interpretation of Results

Careful examination of the results of Fig. 1 above makes it clear that a set of mission orders is being carried out by a human being under the direction of some sort of higher authority that behaves like an MEA. An interesting question for a person encountering this keyboard and screen interaction experience with no knowledge of its implementation is: Is there a remote human providing the listed responses to keyboard entries? The answer is that there could be or not. Specifically, if the dialogue of Fig. 1 is associated with operation of a manned submarine [1, 2], then orders could be coming from the human submarine commander, with responses from the officer of the deck. On the other hand, as demonstrated by the code of Appendix A, the commander could in fact be a software realization of an MEA. “Lifting the hood” by reading the code of Appendix A shows that, for the results of Fig. 1, “commander1” is in fact a software object and not a “hardware” human. So is Appendix A an example of artificial intelligence? The authors believe that the answer to this question would have been “yes” at some point in the past since, without looking at the code, it is impossible to tell if the results observed involve two humans or one. However, at present, it is our view that most AI researchers today would prefer the answer “no” for reasons discussed in the next paragraph.

The crux of the RBM architecture is that the top level of the mission controller is a loop-free MEA. Such a machine can have only a finite number of dialogues such as those of Fig. 1 above. This means that it can be evaluated for every possible execution situation and vetted by a human mission expert as providing the correct commands in every circumstance. This in turn means that the mission rule set defines a problem in *propositional logic*, not *predicate logic* [5, 6]. That is, propositional logic deals with *instances* of classes whereas predate logic deals with logical relations *among* classes themselves and involves unbound variables and quantifiers [5, 6]. For this reason, we would say that the RBM Architecture does *not* use “AI” at the top level. This is of the utmost importance since it avoids the “HAL” paradox demonstrated in the famous sci-fi movie *2001*, in which a computer with inferencing capability (HAL) reasons from general principles that to meet some overarching goals it would be better to sacrifice the space ship commander than to return him to the ship. An MEA uses no inferencing at all and if “wrapped around” lower level elements of robot or drone software would not permit this to happen. In this sense a top level MEA can “restrain” lower level software elements which could contain behaviors using inferencing, and therefore capable of “going crazy”.

To make the above considerations more concrete, we believe that it is useful to consider the examples of driverless automobiles and aircraft autopilots. With regard to the later, it is reasonable to think of a top level MEA consisting of five or so phases such as: “takeoff”, “climb”, “cruise”, “descent”, and “landing”. Within each of these phases, if the autopilot and the human pilot disagree, this could signal a “constraint” outcome that would result in some emergency action to correct the disagreement. This potentially could avoid some fatal commercial airline crashes of the sort reported in recent news reports. With regard to driverless vehicles, if a top level MEA were to monitor the removal of a driver’s hands from the steering wheel, this could likewise signal a constraint violation in certain circumstances, leading to corrective action, such as bringing the vehicle to a safe stop. The authors think that it is urgent that the possibility of these two applications of MEAs to prevent death or injuries to human passengers be evaluated as soon as possible. If results are found to be positive, legislation requiring such "self-monitoring" characteristics in life-critical software would then become appropriate.

The following is an example of a policy/legal statement as described above: "Henceforth, in this jurisdiction, the top-level flow of control for all real-time software supporting life-critical systems must be strictly written in a way confirmable by a human supervisor. Such mission logic is typically describable by a hierarchical directed acyclic graph (DAG or "process flow graph") and therefore capable of strict validation. Every such level of the robot control software must be fail-safe and ethically constrained. Further, such a graph must be exhaustively testable using a software realization as a MEA, and approved by a single responsible individual. Subsequent definition and testing of subordinate code blocks must be accomplished for each of the nodes of the approved DAG mission logic."

Further refinement is needed to express such requirements in a sound legal manner. We expect to experiment with such declarations of confirmability requirements on a case-by-case basis as a regular part of upcoming mission development.

Section III. below shows how a universal Mission Execution Engine (MEE) can be "factored out" of the code of Appendix A using Prolog. It also includes examples of two missions, also expressed in Prolog, that demonstrate the sense in which the MEE presented is universal.

III. Universal Mission Execution Engine (MEE) Using Prolog

For Turing Machines, early on, there developed the idea of a "Universal Turing Machine" (UTM) in which the execution mechanics of *any* TM are embodied in a "Universal Mission Execution Engine", while the finite state machine (FSM) part specifying the specific computation to be made (the "mission") is encoded on the TM "tape". This realization created a formal mathematical model for the modern "stored program" digital computer. Fortunately, such a decomposition is likewise possible for an MEA. Appendix B provides Prolog code for such a machine.

To understand the code of Appendix B, it is necessary to first realize that it is written in the "Allegro" dialect of Prolog and not in the ISO Standard "Edinburg" version [3]. While the syntax of the Allegro dialect is explained in [7], and in Allegro online documentation, it is useful to present a brief summary of this material here. Specifically, referring to Appendix B, in a line beginning with a back arrow, the first expression is true if all following it are true. Thus, the first line in the "mission execution rule set" says that a mission is executed if it is initialized and then the current phase is executed, followed by repeated subsequent phase execution until the "done" criterion is satisfied. That is, this one line mechanizes Fig. 2 of [1]. Subsequent lines of code can be understood in the same way providing that it is understood that "?X" as well as "?A", "?Q", etc. indicate "unbound" variables (variables with no value assigned). This feature is rarely found in other computer languages, and makes it difficult at first to read or write Prolog code. It is generally accurate and useful to think of unbound variables as being analogous to *pronouns* in natural human language.

Prolog is executed by trying to find bindings for all unbound variables from the given facts and "inference" rules. When no bindings can be found, Prolog backs up ("backtracks") until one or more is found. The symbol "!" ("cut") stops this action by acting like a trapdoor when moving from left to right while searching for a binding. That is, backtracking from right to left in a Prolog rule statement must stop when a cut is encountered. These features are remarkably powerful and give Prolog great power in expressing logical relationships such as those that arise in defining and executing multi-stage human or machine missions. The reader is invited to consult [3] for a deeper understanding of these ideas.

Appendix C presents mission orders in Prolog form for the search and sample mission currently under consideration in this memorandum. If Appendices B and C are both loaded and compiled, then results like Fig. 2 below, analogous to those of Fig. 1, can be obtained. The importance of these results is that separation of MEE code from mission code has been achieved. Of course, this could be done directly in Lisp without using Prolog, but we believe that this would be very much harder, and we have not taken this step.

```
CG-USER(1): (run)
Search Area A!
Did goal succeed (s), fail (f), or abort (c)?s
Sample environment!
Did goal succeed (s), fail (f), or abort (c)?s
Search Area B!
Did goal succeed (s), fail (f), or abort (c)?s
Rendezvous with Vehicle 2!
Did goal succeed (s), fail (f), or abort (c)?s
Proceed to recovery!
Did goal succeed (s), fail (f), or abort (c)?s
Mission succeeded.
Yes
No.

CG-USER(2): (run)
Search Area A!
Did goal succeed (s), fail (f), or abort (c)?f
Search Area B!
Did goal succeed (s), fail (f), or abort (c)?s
Rendezvous with Vehicle 2!
Did goal succeed (s), fail (f), or abort (c)?c
Proceed to recovery!
Did goal succeed (s), fail (f), or abort (c)?s
Mission succeeded.
Yes
No.

CG-USER(3): (run)
Search Area A!
Did goal succeed (s), fail (f), or abort (c)?c
Rendezvous with Vehicle 2!
Did goal succeed (s), fail (f), or abort (c)?c
Proceed to recovery!
Did goal succeed (s), fail (f), or abort (c)?c
Mission aborted.
Yes
No.
```

Fig. 2: Typical Results Obtained for Universal MEE with Mission Orders Separately Loaded

At this point it is appropriate to consider just how “universal” in fact is the MEE of Appendix B? Evidently, changing to more or less phases in a mission is easily accomplished by simply using the phases of Appendix C as a template. To address the question of less phase *outcomes*, the authors copied from [7] the binary outcome version (no constraints on phase execution) of the search and sample mission and included it in this memorandum as Appendix D. When compiled and loaded together with the referenced MEE code, results as listed below in Fig. 3 were obtained. Evidently, the MEE also functions correctly for this code. Per prior discussion on the complexity of proving correctness of mission orders by exhaustive testing, if it can be avoided, we do not recommend mission orders with more than three possible outcomes in any phase, nor more than five or six mission phases. Instead, we believe that *phase abstraction* should be applied when it appears that more than this number of phases is needed [1]. Furthermore, if there are phases with more than three outcomes possible, restricting such an MEA to even fewer phases to enable human vetting of exhaustive testing results may become necessary.

```
CG-USER(1): (run)
Search Area A!
Search successful?y
Sample environment!
Sample obtained?n
Return to base!
At base?y
Mission succeeded.
Yes
No.

CG-USER(2): (run)
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(3):
```

Fig. 3: Typical Results Obtained for Universal MEE with Binary Branching Mission Orders

IV. Autonomous Robots

Sections II and III above are concerned with situations where a human being is involved more or less continuously during mission execution. This includes such applications as conventional manned vehicle operations, drone missions, and current generation “driverless” passenger vehicles. In some circumstances, notably for unmanned submarines, stealth considerations may dictate that there be no communication at all between a human operator and a robot vehicle during mission execution. Appendix E below shows how an onboard MEA can direct a mission in such a case. Figure 4 below shows typical results obtained in executing the code of Appendix E.

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

CG-USER(2): (run)
"Mission complete."

CG-USER(3): (results)
(PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Constraint." PHASE4
"Rendezvous with Vehicle2!" "Failed." PHASE5 "Proceed to recovery!" "Success."
"Mission complete.")

CG-USER(4): (run)
"Mission complete."

CG-USER(5): (results)
(PHASE1 "Search Area A!" "Constraint." PHASE4 "Rendezvous with Vehicle2!" "Failed."
PHASE5 "Proceed to recovery!" "Success." "Mission complete.")

CG-USER(6): (run)
NIL

CG-USER(7): (results)
NIL

CG-USER(8): (run)
"Mission complete."

CG-USER(9): (run)
"Mission aborted. Vehicle recovered"

CG-USER(10): (results)
(PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Failed." PHASE5
"Proceed to recovery!" "Constraint." "Mission aborted. Vehicle recovered")

CG-USER(11):
```

Fig. 4. Results of Execution of Several Missions by Simulated Autonomous Vehicle.
(Simulator value of "NIL" means that vehicle was lost and no results are available).

It is useful to note that considerable experience with manned vehicle or drone operations generally is needed to construct good MEA rules for autonomous vehicle missions. Also, whereas "expert systems" intended to replace humans in some activities must generally deal with incompleteness and conflicts in rule sets, these two problems cannot arise when an MEA is in control of an autonomous vehicle mission. This results from the exhaustive testability of MEA as demonstrated in the following Section V.

A further observation relative to the above results is that there is no discernable pattern to the outcome of successive calls to mission execution. Examination of the code of Appendix E shows that this is because the outcome of any phase is determined by a call to (random 3). This function call delivers a random integer from the set [0, 1, 2] with equal probability for each value, thereby mimicking (albeit primitively) the randomness of actual mission execution by a real autonomous vehicle.

V. Proof of Correctness of Mission Orders by Exhaustive Testing

While axioms (rule sets) in predicate logic cannot in general be proved complete and consistent, they can be (by exhaustion of all cases) in propositional logic [5,6]. In other words, there can be no "unforeseen circumstances" if no quantifiers or categorical statements are used in mission definition. This is the case for missions defined by loop-free MEAs. Such missions are equivalently defined by "directed acyclic graphs" (DAG) in the form of "process flow charts" such as Fig. 7 of [1]. Appendix F presents Lisp code for finding all paths from Phase1 to Phase5 for the "2018-JOE" mission under consideration in this memorandum. Fig. 5 below shows results obtained by executing this code.

```
CG-USER(1): (find-all-paths-to-goal)

((PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Success." PHASE3
"Search Area B!" "Constraint." PHASE4 "Rendezvous with Vehicle2!" "Constraint."
PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Success." PHASE3
"Search Area B!" "Constraint." PHASE4 "Rendezvous with Vehicle2!" "Failed." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Success." PHASE3
"Search Area B!" "Constraint." PHASE4 "Rendezvous with Vehicle2!" "Success." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Success." PHASE3
"Search Area B!" "Failed." PHASE4 "Rendezvous with Vehicle2!" "Constraint." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Success." PHASE3
"Search Area B!" "Failed." PHASE4 "Rendezvous with Vehicle2!" "Failed." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Success." PHASE3
"Search Area B!" "Failed." PHASE4 "Rendezvous with Vehicle2!" "Success." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Success." PHASE3
"Search Area B!" "Success." PHASE4 "Rendezvous with Vehicle2!" "Constraint." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Success." PHASE3
"Search Area B!" "Success." PHASE4 "Rendezvous with Vehicle2!" "Failed." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Success." PHASE3
"Search Area B!" "Success." PHASE4 "Rendezvous with Vehicle2!" "Success." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Constraint." PHASE4
"Rendezvous with Vehicle2!" "Constraint." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Constraint." PHASE4
"Rendezvous with Vehicle2!" "Failed." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Constraint." PHASE4
"Rendezvous with Vehicle2!" "Success." PHASE5)

 (PHASE1 "Search Area A!" "Failed." PHASE3 "Search Area B!" "Constraint." PHASE4
"Rendezvous with Vehicle2!" "Constraint." PHASE5)

 (PHASE1 "Search Area A!" "Failed." PHASE3 "Search Area B!" "Constraint." PHASE4
"Rendezvous with Vehicle2!" "Failed." PHASE5)

 (PHASE1 "Search Area A!" "Failed." PHASE3 "Search Area B!" "Constraint." PHASE4
"Rendezvous with Vehicle2!" "Success." PHASE5)
```

```
 (PHASE1 "Search Area A!" "Failed." PHASE3 "Search Area B!" "Failed." PHASE4
"Rendezvous with Vehicle2!" "Constraint." PHASE5)

 (PHASE1 "Search Area A!" "Failed." PHASE3 "Search Area B!" "Failed." PHASE4
"Rendezvous with Vehicle2!" "Failed." PHASE5)

 (PHASE1 "Search Area A!" "Failed." PHASE3 "Search Area B!" "Failed." PHASE4
"Rendezvous with Vehicle2!" "Success." PHASE5)

 (PHASE1 "Search Area A!" "Failed." PHASE3 "Search Area B!" "Success." PHASE4
"Rendezvous with Vehicle2!" "Constraint." PHASE5)

 (PHASE1 "Search Area A!" "Failed." PHASE3 "Search Area B!" "Success." PHASE4
"Rendezvous with Vehicle2!" "Failed." PHASE5)

 (PHASE1 "Search Area A!" "Failed." PHASE3 "Search Area B!" "Success." PHASE4
"Rendezvous with Vehicle2!" "Success." PHASE5)

 (PHASE1 "Search Area A!" "Success." PHASE2 "Sample environment!" "Failed." PHASE5)

 (PHASE1 "Search Area A!" "Constraint." PHASE4 "Rendezvous with Vehicle2!"
"Constraint." PHASE5)

 (PHASE1 "Search Area A!" "Constraint." PHASE4 "Rendezvous with Vehicle2!" "Failed."
PHASE5)

 (PHASE1 "Search Area A!" "Constraint." PHASE4 "Rendezvous with Vehicle2!" "Success."
PHASE5))

CG-USER(2):
```

Fig. 5. All Paths to Goal Generated by Code of Appendix F

Examining Fig. 5, it can be seen that a total of 25 unique paths exist connecting Phase1 to Phase5. But why does the code executed not continue on to the terminal states "Mission Complete" and "Mission Abort"? This is because Phase5 is a "choke point" in the mission graph in the sense that all paths to either terminal state must pass through it. Appending to each of the above paths the three possible exits from Phase5 increases the total number of paths through the subject graph to 75. However, "the game is over" when Phase5 is reached (because no more commands remain), so there is no reason to ask for human expert vetting of results beyond those of Fig. 5.

Turning to the code of Appendix F, it is apparent that a "breadth first" approach has been taken. Specifically, starting at Phase1, there must be exactly three exits to following phases. These are all produced by calling the function (method) called "start-mission-execution-tree". After this function has been executed, the "mission execution tree" is one layer deep with Phase1 at the top, and Phase2, Phase3, and Phase4 at the next layer down (level 1). This can be verified by exhaustive examination of all path results of Fig. 5. Level 1 nodes at this time represent the "search frontier" of the breadth first search for all paths to Phase5. The next layer down in this tree is obtained by expanding each frontier node to its three successor nodes at the next level. This is accomplished for any given frontier node by the method "expand-frontier-node". This method is used in turn by "advance-execution-tree-frontier" to construct the entire level 2 frontier. The latter method also tests to see if Phase5 has been reached, and stores such paths in the "all-paths-to-goal" slot of "agent1". The mission execution tree is finished when level 4 is complete. At this point, all paths through the mission flow graph terminate in Phase5.

To elaborate on the above discussion, referring to the code, it can be seen that the list of paths to the goal is built up using the Lisp “push” function with the consequence that first results appear at the right side (bottom of Fig. 5) of the resulting list. Thus, it can be seen that there are 4 paths of length three, 12 paths of length four, and 9 paths of length five, for a total of 25 paths from mission start to mission finish. Other features of the algorithm of Appendix F are likewise observable in this list of results.

VI. Physical Testing of RBM and MEA

At the time of our first at-sea testing of the viability of the RBM software architecture, with the *Phoenix* autonomous submarine [8], we mistakenly believed (along with most of the AI community) that autonomous mission execution is equivalent to theorem proving. As a consequence, our submarine had two onboard computers, one to run Prolog for top level mission control and another for all real-time control of maneuvering and other low-level functions. As a consequence, although Phoenix was never lost at sea, we had no way of proving that our mission specification (in Prolog) was what we had intended.

By the time our second autonomous submarine, *Aries*, was ready for at-sea testing, we realized that Prolog was not needed, and that one on-board computer would be enough [8]. However, Aries still had no ethical constraints, and would, instead, execute any mission plan without regard to unintended collateral effects. Moreover, because our Aries mission coding relied solely on human understanding of mission goals, and we had yet to develop any theoretical underpinnings for this code, we still had no means of showing that mission orders represented our intentions. This situation subsequently lead us to the MEA formalism developed here and in [1, 7, 10].

With the results contained in this memorandum, we feel that we are now ready for more at-sea or other physical experiments. Since we no longer have a suitable vehicle at our disposal for conducting real missions using MEA mission control, we are now looking, and will continue to look, for academic or commercial partners to join with us in such an effort.

VII. Summary, Conclusions, and Recommendations

For the first time, through the MEA and RBM formalisms, there is a mathematically sound, and provable, way of specifying and executing ethically-constrained and fail-safe missions by manned or unmanned vehicles. This memorandum contains new code and new explanations relating to these ideas. With regard to code, Appendices A, B, C, and F can be used to develop and test new missions with only minor modifications to the code presented. While Appendix E might also be so used, it can alternatively serve as executable specifications for real-time onboard code development for the given mission in an object-oriented imperative language such as C++ or Java.

It is of fundamental importance that an MEA abstraction/refinement hierarchy uniquely establishes human responsibility and culpability analogous to the “chain of command” in military hierarchies. This means that, when using this approach, it is not possible for individuals in this chain to hide behind excuses such as: “I am not a programmer” and “The robot did it by itself”. We urge the further exploration of the methodology presented here, at the earliest possible time, by experimentation with real manned or unmanned vehicles engaged in meaningful missions.

References

1. Brutzman, D. P., Blais, C. L., Davis, D. T, and McGhee, R. B., “Ethical Mission Definition and Execution for Maritime Robots Under Human Supervision”, *IEEE Journal of Ocean Engineering*, Vol. 43, No. 2, April, 2018.

2, Byrnes, R. B., A. J. Healey, R. B. McGhee, M. L. Nelson, S. Kwak, and D. P. Brutzman. "The Rational Behavior Software Architecture for Intelligent Ships." *Naval Engineers Journal*, March 1996: 43-56.

3. Rowe, N.C., Artificial Intelligence through Prolog, Prentice Hall, Englewood Cliffs, NJ 07632, 1988

4. Koschmann, Timothy, *The Common Lisp Companion*, John Wiley and Sons, 1990.

5. Hofstadter, D. R., *Godel, Escher, Bach: an Eternal Golden Braid*, Ed. 2, Basic Books, Inc., New York,1999.

6. Wikipedia, “First-order Logic,” https://en.wikipedia.org/wiki/First-order_logic.

7. Brutzman, D. P., McGhee, R. B., and Davis D. T., “An Implemented Universal Mission Controller with Run Time Ethics Checking for Autonomous Unmanned Vehicles – a UUV Example”, *OES-IEEE Autonomous Underwater Vehicles 2012.* Southampton, UK, 2012.

8. Brutzman, D., et al, “The Phoenix Autonomous Underwater Vehicle”, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, Ch. 13, pp. 323-360, ed. by Kortenkamp, D., et al, MIT Press, Cambridge, MA 02142, 1998.

9. Davis, D. T. *Design, Implementation, and Testing of a Common Data Model Supporting Autonomous Vehicle Compatibility and Interoperability*. Ph.D. Dissertation, Computer Science, Naval Postgraduate School, Monterey, CA, 2006.

10. D. T. Davis, D. P. Brutzman, C. L. Blais, and R. B. McGhee, “Ethical Mission Definition and Execution for Maritime and Naval Robotic Vehicles: A Practical Approach,” in *Proc. IEEE/MTS Oceans 2016*, Monterey, CA, USA,

Appendix A: Lisp Code for Manned or Remotely Operated Vehicle Mission Execution

```
;This code written in ANSI Common Lisp, Allegro 10.1 enhancement, from Franz, Inc., by
;Prof. Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School,
;Monterey, CA 93943. Date of latest update: 13 May 2019.

;The mission coded below is taken from Fig. 7, pg. 434, in "Ethical Mission Definition
;and Execution for Maritime Robots Under Human Supervision", IEEE Journal of Oceanic
;Engineering (JOE), Vol. 43, No. 2, April, 2018.

;Comment1: As of the time of this writing, the default Allegro IDE editor window print
;font is oversized, and does not print a full-width copy of the IDE Debug or Editor
;windows. To fix this, go to the top of the Allegro opening screen and click "Tools".
;Then, from the drop-down menu, select "Options". Next, on the resulting "Options"
;pop-up screen, select, "Fonts". Then click on "Editor Printouts". When a window pops up
;asking "Revert to the default fixed-width font?", answer "No". A "Font" window will
;pop up. In this window, scroll down fonts to "Courier New". Scroll "Print Size" to "8".
;Click "OK" at bottom of this window, and it will disappear. Finally, click "OK" in
;"Options" window, and it too will disappear. The result of all this manipulation will
```

```
;be that clicking "Print" on the drop-down menu from the "File" tab of the IDE opening
;screen will print an exact copy of what you see on the IDE editor or debug windows.
;You will be glad you did all of this.

;Comment2: The reason that "Courier New" is recommended above is that it is a "fixed-
;width" font. This means that when Lisp code is viewed on screen or in a printout,
;characters, especially parentheses, line up vertically, facilitating human
;comprehension.

;Comment3: At some point, after you install Allegro 10.1, the compiler will ask you "Do
;you want to start a project". Answer "No." Saying "Yes" will introduce you to needless
;complexity not needed to run the attached code. Instead, after installing and opening
;Allegro, starting with an electronic copy of this report, highlight the code you want
;to run and then use ctrl-c keys to copy. Next, place the mouse cursor in the editor window and
;paste with ctrl-v. Now using the "File" menu, select "save as" and enter a file name of your
;choosing in pop-up menu. Next, click on the dumptruck icon at top of screen, select your
;file from popup menu,and double click on it. The debug window will show some system files
;loading and will then present a prompt as shown in Fig. 1 above. Now type "(run)" to
;prompt and respond to queries as in Fig. 1. Try you own responses and see if the results
;correspond to the mission flow graph.

;It should be noted that any other 5-phase mission can be exhaustively tested by
;appropriately editing the phase initialization function below. Evidently, missions of
;fewer than five phases can be tested in the same way by simply not initializing unused
;phases. Creation of mission orders with more than five phases is not recommended since
;the list of all possible dialogs between the mission orders agent and the mission execution
;agent is likely to be too long for accurate human validation. Instead, as discussed in the
;referenced JOE paper, "phase abstraction" can be used to combine phases and thereby
;reduce the number of phases in top level mission orders.


(defvar commander1)
(defvar phase1)
(defvar phase2)
(defvar phase3)
(defvar phase4)
(defvar phase5)
(defvar 2018-JOE-orders)

;Above declarations have no function other than to stop compiler complaints about
;use of global ("special") variables.

(defclass mission-phase ()
  ((command :accessor command)
   (successor-list :accessor successor-list)));Elements are (outcome next-phase).

(defclass human-agent-mission-execution-engine ()
  ((external-agent-response :accessor external-agent-response :initform nil)
   (current-execution-phase :accessor current-execution-phase :initform 'phase1)
   (successor-list-index :accessor successor-list-index :initform 0)))

(defun create-5-mission-phases ()
  (setf phase1 (make-instance 'mission-phase)
        phase2 (make-instance 'mission-phase)
        phase3 (make-instance 'mission-phase)
        phase4 (make-instance 'mission-phase)
        phase5 (make-instance 'mission-phase)))

(defmethod initialize-phase ((phase mission-phase) command successor-list)
  (setf (command phase) command
    (successor-list phase) successor-list))

(defun initialize-2018-mission-phases ()
  (initialize-phase phase1 "Search Area A!"
                    '(("Success." phase2) ("Failed." phase3) ("Constraint." phase4)))
  (initialize-phase phase2 "Sample environment!"
                    '(("Success." phase3) ("Failed." phase5) ("Constraint." phase4)))
  (initialize-phase phase3 "Search Area B!"
                    '(("Success." phase4) ("Failed." phase4) ("Constraint." phase4)))
  (initialize-phase phase4 "Rendezvous with Vehicle2!"
                    '(("Success." phase5) ("Failed." phase5) ("Constraint." phase5)))
  (initialize-phase phase5 "Proceed to recovery!"
                    '(("Success." "Mission complete.") ("Failed." "Mission aborted.")
                      ("Constraint." "Mission aborted."))))

(defmethod issue-command ((mission-commander human-agent-mission-execution-engine))
  (let* ((phase (current-execution-phase mission-commander))
         (command (command (eval phase))))
    (issue-order command)))

(defmethod ask-result ((mission-commander human-agent-mission-execution-engine))
  (let* ((result (ask-outcome)))
    (setf (external-agent-response mission-commander) result)))

(defmethod set-next-phase ((mission-commander human-agent-mission-execution-engine))
  (let* ((phase (current-execution-phase mission-commander))
         (successor-list (successor-list (eval phase)))
         (index (successor-list-index mission-commander))
         (next-phase (second (nth index successor-list))))
    (setf (current-execution-phase mission-commander) next-phase)))

(defmethod set-successor-list-index ((mission-commander human-agent-mission-execution-engine))
  (let* ((index (convert-outcome-to-index (external-agent-response mission-commander))))
    (setf (successor-list-index mission-commander) index)))

(defun execute-phase ()
  (issue-command commander1)
  (ask-result commander1)
  (set-successor-list-index commander1)
  (set-next-phase commander1))

(defun create-2018-JOE-orders ()
  (setf 2018-JOE-orders (create-5-mission-phases))
  (initialize-2018-mission-phases))

(defun start ()
  (create-2018-JOE-orders)
  (setf commander1 (make-instance 'human-agent-mission-execution-engine)))

(defun issue-order (command)
  (format t "~A" command)
  (terpri))

(defun ask-outcome ()
  (format t "Enter execution outcome as s, f, or c:")
  (let ((value (read)))
       (if (member value '(s f c))
           value
          (ask-outcome))))

(defun execute-mission ()
  (if (equal (current-execution-phase commander1) 'phase5) (execute-phase)
    (progn (execute-phase)(execute-mission))))

(defun run ()
  (start)
  (execute-mission))

(defun convert-outcome-to-index (x)
  (cond ((equal x 's) 0)
        ((equal x 'f) 1)
        ((equal x 'c) 2)))
```

Appendix B: Prolog Code for Universal Mission Execution Engine

```
;This code written in ANSI Common Lisp, Allegro 10.1 enhancement, from Franz, Inc., by
;Prof. Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School,
;Monterey, CA 93943. Date of latest update: 11 May 2019.
```

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

;After this code has been entered into Allegro Editor window and compiled, it is necessary to
;similarly enter and compile a mission. File "2019 Mission Orders" included in this report
;provides an example. Entering "(run)" to Lisp prompt then runs both files together.

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

;Facts
(<-- (current_phase 0)) ;Starting phase.
(<-- (current_phase_outcome s))

;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))
(<-- (ask_outcome ?A) (ask "Did goal succeed (s), fail (f), or abort (c)" ?A))

;Utility functions
(<-- (change_phase ?old ?new) (retract ((current_phase ?old))) (asserta ((current_phase ?new))))
(<-- (update_outcome) (ask_outcome ?A) (abolish current_phase_outcome 1) (asserta
((current_phase_outcome ?A))))


;Test functions (illustrate format for calling predicates from Lisp)
(defun run () (?- (execute_mission)))
(defun update () (?- (update_outcome)))
(defun mission-phase () (?- (current_phase ?X)))
(defun outcome () (?- (current_phase_outcome ?X)))
```

Appendix C: Prolog Code for Ternary Branching Search and Sample Mission

```
;This code written in ANSI Common Lisp, Allegro 10.1 enhancement, from Franz, Inc., by
;Prof. Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School,
;Monterey, CA 93943. Date of latest update: 11 May 2019.

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

;Mission specification

(<-- (execute_phase 1) (command "Search Area A") (update_outcome)
                      (current_phase_outcome s) (change_phase 1 2))
(<- (execute_phase 1) (current_phase_outcome f) (change_phase 1 3))
(<- (execute_phase 1) (current_phase_outcome c) (change_phase 1 4))

(<- (execute_phase 2) (command "Sample environment") (update_outcome)
                      (current_phase_outcome s) (change_phase 2 3))
(<- (execute_phase 2) (current_phase_outcome f) (change_phase 2 5))
(<- (execute_phase 2) (current_phase_outcome c) (change_phase 2 4))

(<- (execute_phase 3) (command "Search Area B") (update_outcome)
                      (current_phase_outcome s) (change_phase 3 4))
(<- (execute_phase 3) (current_phase_outcome f) (change_phase 3 4))
(<- (execute_phase 3) (current_phase_outcome c) (change_phase 3 4))

(<- (execute_phase 4) (command "Rendezvous with Vehicle 2") (update_outcome)
                      (current_phase_outcome s) (change_phase 4 5))
(<- (execute_phase 4) (current_phase_outcome f) (change_phase 4 5))
(<- (execute_phase 4) (current_phase_outcome c) (change_phase 4 5))

(<- (execute_phase 5) (command "Proceed to recovery") (update_outcome)
    (current_phase_outcome s) (change_phase 5 'mission_complete) (report "Mission succeeded"))
(<- (execute_phase 5) (current_phase_outcome f) (change_phase 5 'mission_abort)
    (report "Mission failed"))
(<- (execute_phase 5) (current_phase_outcome c) (change_phase 5 'mission_abort)
    (report "Mission aborted"))
```

Appendix D: Prolog Code for Binary Branching Search and Sample Mission

```
;This code written in ANSI Common Lisp, Allegro 10.1 enhancement, from Franz, Inc., by
;Prof. Robert B. McGhee (robertbmcghee@gmail.com) at the Naval Postgraduate School,
;Monterey, CA 93943. Date of latest update: 11 May 2019.

(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 "Attempt Area B search") (change_phase 3 4))

(<- (execute_phase 4) (command "Attempt rendezvous with UUV2") (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"))
```

Appendix E: Lisp Code for Autonomous Robot Mission Specification and Simulated Execution

```
;This code is written in ANSI Common Lisp, Allegro 10.1, complimentary copy available
;at Franz.com. All code written by Prof. Robert B. McGhee (robertbmcghee@gmail.com) at
;the Naval Postgraduate School, Monterey, CA 93943. Date of latest update: 19 April 2019.

;Mission coded below is taken from Fig. 7, pg. 434, in "Ethical Mission Definition and
;Execution for Maritime Robots Under Human Supervision", IEEE Journal of Oceanic
;Engineering (JOE), Vol. 43, No. 2, April, 2018.


(defvar controller1)
(defvar phase1)
(defvar phase2)
(defvar phase3)
(defvar phase4)
(defvar phase5)
(defvar 2018-JOE-orders)

;Above declarations have no function other than to stop compiler complaints about
;use of global ("special") variables.

(defclass mission-phase ()
  ((command :accessor command)
   (successor-list :accessor successor-list)));Elements are (outcome next-phase).

(defclass robot-agent-mission-execution-engine ()
  ((external-agent-response :accessor external-agent-response :initform (random 3))
   (current-execution-phase :accessor current-execution-phase :initform 'phase1)
   (mission-log :accessor mission-log :initform '(phase1))))

(defun create-5-mission-phases ()
  (setf phase1 (make-instance 'mission-phase)
        phase2 (make-instance 'mission-phase)
        phase3 (make-instance 'mission-phase)
        phase4 (make-instance 'mission-phase)
        phase5 (make-instance 'mission-phase)))

(defmethod initialize-phase ((phase mission-phase) command successor-list)
  (setf (command phase) command
    (successor-list phase) successor-list))

(defun initialize-2018-mission-phases ()
  (initialize-phase phase1 "Search Area A!"
                    '(("Success." phase2) ("Failed." phase3) ("Constraint." phase4)))
  (initialize-phase phase2 "Sample environment!"
                    '(("Success." phase3) ("Failed." phase5) ("Constraint." phase4)))
  (initialize-phase phase3 "Search Area B!"
                    '(("Success." phase4) ("Failed." phase4) ("Constraint." phase4)))
  (initialize-phase phase4 "Rendezvous with Vehicle2!"
                    '(("Success." phase5) ("Failed." phase5) ("Constraint." phase5)))
  (initialize-phase phase5 "Proceed to recovery!"
                    '(("Success." "Mission complete.")
                      ("Failed." nil)
                      ("Constraint." "Mission aborted. Vehicle recovered"))))

(defmethod mission-log-increment ((onboard-MEE robot-agent-mission-execution-engine))
  (let* ((phase (current-execution-phase onboard-MEE))
         (command (command (eval phase)))
         (successor-list (successor-list (eval phase)))
         (index (external-agent-response onboard-MEE))
         (outcome(first (nth index successor-list)))
         (next-phase (second (nth index successor-list))))
    (list command outcome next-phase)))

(defmethod update-mission-log ((onboard-MEE robot-agent-mission-execution-engine))
  (let* ((log (mission-log onboard-MEE))
         (index (external-agent-response onboard-MEE))
         (phase (current-execution-phase onboard-MEE))
         (increment (mission-log-increment onboard-MEE)))
    (setf (mission-log onboard-MEE)
          (if (not (and (= index 1) (equal phase 'phase5))) (append log increment)))))

(defmethod ask-outcome ((onboard-MEE robot-agent-mission-execution-engine))
  (let* ((outcome (random 3)))
    (setf (external-agent-response onboard-MEE) outcome)))

(defmethod set-next-phase ((onboard-MEE robot-agent-mission-execution-engine))
  (setf (current-execution-phase onboard-MEE) (first (last (mission-log onboard-MEE)))))

(defun execute-phase ()
  (ask-outcome controller1)
  (update-mission-log controller1)
  (set-next-phase controller1))

(defun create-2018-JOE-orders ()
  (setf 2018-JOE-orders (create-5-mission-phases))
  (initialize-2018-mission-phases))

(defun start ()
  (create-2018-JOE-orders)
  (setf controller1 (make-instance 'robot-agent-mission-execution-engine)))

(defun execute-mission ()
  (if (equal (current-execution-phase controller1) 'phase5) (execute-phase)
    (progn (execute-phase)(execute-mission))))

(defun run ()
  (start)
  (execute-mission))

(defun results ()
  (pprint (mission-log controller1)))
```

Appendix F: Lisp Code for Finding All Possible Mission Execution Outcomes

```
;This code is written in ANSI Common Lisp, Allegro 10.1, complementary copy available
;at Franz.com. All code written by Prof. Robert B. McGhee (robertbmcghee@gmail.com) at
;the Naval Postgraduate School, Monterey, CA 93943. Date of latest update: 18 April 2019.

;Mission coded below is taken from Fig. 7, pg. 434, in "Ethical Mission Definition and
;Execution for Maritime Robots Under Human Supervision", IEEE Journal of Oceanic
;Engineering (JOE), Vol. 43, No. 2, April, 2018.

(defvar agent1)
(defvar phase1)
(defvar phase2)
(defvar phase3)
(defvar phase4)
(defvar phase5)
(defvar 2018-JOE-orders)

;Above declarations have no function other than to stop compiler complaints about
;use of global ("special") variables.

(defclass mission-phase ()
  ((command :accessor command)
   (successor-list :accessor successor-list)));Elements are (outcome next-phase).

(defun create-5-mission-phases ()
  (setf phase1 (make-instance 'mission-phase)
        phase2 (make-instance 'mission-phase)
        phase3 (make-instance 'mission-phase)
        phase4 (make-instance 'mission-phase)
        phase5 (make-instance 'mission-phase)))

(defun create-2018-JOE-orders ()
  (setf 2018-JOE-orders (create-5-mission-phases))
  (initialize-2018-mission-phases))

(defclass DAG-find-all-paths-agent ()
  ((current-search-phase :accessor current-search-phase :initform 'phase1)
   (successor-list :accessor successor-list :initform nil)
   (successor-list-index :accessor successor-list-index :initform 0)
   (all-paths-to-frontier :accessor all-paths-to-frontier :initform nil)
   (all-paths-to-goal :accessor all-paths-to-goal :initform nil)
   (new-paths-list-length :accessor new-paths-list-length :initform nil)
   (new-paths-list :accessor new-paths-list :initform nil)
   (new-path-segments-list :accessor new-path-segments-list :initform nil)))

;DAG means "Directed Acyclic Graph".

(defun update-successor-list-index (index modulus) (mod (1+ index) modulus))

(defmethod initialize-phase ((phase mission-phase) command successor-list)
  (setf (command phase) command
        (successor-list phase) successor-list))

;The system function "eval", as used below, obtains pointer (address) for a global object.
```

```
(defmethod get-new-path-segment ((agent DAG-find-all-paths-agent))
  (let* ((phase (current-search-phase agent))
         (command (command (eval phase)))
         (successor-list (successor-list (eval phase)))
         (index (successor-list-index agent))
         (outcome (first (nth index successor-list)))
         (next-phase (second (nth index successor-list)))
         (path-segment (list phase command outcome next-phase)))
    (setf (successor-list-index agent) (update-successor-list-index index 3))
    (push path-segment (new-path-segments-list agent))))

(defmethod update-new-path-segments-list ((agent DAG-find-all-paths-agent))
  (setf (new-path-segments-list agent) nil)
  (dotimes (i 3) (get-new-path-segment agent))
  (new-path-segments-list agent))

(defmethod extend-path ((agent DAG-find-all-paths-agent))
  (let* ((all-paths (all-paths-to-frontier agent))
         (new-segment (pop (new-path-segments-list agent)))
         (path (first all-paths)))
    (push (connect path new-segment) (new-paths-list agent))))

(defmethod start-mission-execution-tree ((agent DAG-find-all-paths-agent))
  (setf (new-paths-list agent) (update-new-path-segments-list agent))
  (dotimes (i 3) (store-new-path agent))
  (setf (current-search-phase agent) (flf (all-paths-to-frontier agent)))
  (all-paths-to-frontier agent))

(defmethod expand-frontier-node ((agent DAG-find-all-paths-agent))
  (update-new-path-segments-list agent)
  (dotimes (i 3) (extend-path agent))
  (pop (all-paths-to-frontier agent))
  (setf (current-search-phase agent) (flf (all-paths-to-frontier agent)))
  (new-paths-list agent))

(defmethod store-new-path ((agent DAG-find-all-paths-agent))
  (let* ((new-path (pop (new-paths-list agent))))
    (if new-path (if (not (equal 'phase5 (fl new-path)))
                     (push new-path (all-paths-to-frontier agent))
                     (push new-path (all-paths-to-goal agent))))))

(defmethod extend-all-paths-to-frontier ((agent DAG-find-all-paths-agent))
  (let* ((n (length (all-paths-to-frontier agent))))
    (progn (dotimes (i n) (expand-frontier-node agent))
           (setf (new-paths-list-length agent) (length (new-paths-list agent)))
           (all-paths-to-frontier agent))))

(defmethod advance-execution-tree-frontier ((agent DAG-find-all-paths-agent))
  (extend-all-paths-to-frontier agent)
  (store-all-new-paths agent)
  (setf (current-search-phase agent) (flf (all-paths-to-frontier agent))))

(defmethod store-all-new-paths ((agent DAG-find-all-paths-agent))
  (dotimes (i (new-paths-list-length agent)) (store-new-path agent)))

(defun create-agent1 ()
  (setf agent1 (make-instance 'DAG-find-all-paths-agent)))

(defun initialize-2018-mission-phases ()
  (initialize-phase phase1 "Search Area A!"
                    '(("Success." phase2) ("Failed." phase3) ("Constraint." phase4)))
  (initialize-phase phase2 "Sample environment!"
                    '(("Success." phase3) ("Failed." phase5) ("Constraint." phase4)))
  (initialize-phase phase3 "Search Area B!"
                    '(("Success." phase4) ("Failed." phase4) ("Constraint." phase4)))
  (initialize-phase phase4 "Rendezvous with Vehicle2!"
                    '(("Success." phase5) ("Failed." phase5) ("Constraint." phase5)))
  (initialize-phase phase5 "Proceed to recovery!"
                    '(("Success." "Mission complete.") ("Failed." "Mission aborted.")
                      ("Constraint." "Mission aborted."))))
```

```
(defun start ()
  (create-2018-JOE-orders)
  (create-agent1)
  (start-mission-execution-tree agent1))

(defun connect (list1 list2)
  (if (equal (first (last list1)) (first list2))
      (append list1 (rest list2))
      list1))

(defun flf (list) (first (last (first list))))

(defun fl (list) (first (last list)))

(defun find-all-paths-to-goal ()
  (start)
  (dotimes (i 3) (advance-execution-tree-frontier agent1))
  (pprint (all-paths-to-goal agent1)))

(defun run ()
  (find-all-paths-to-goal))
```