

High Level Mission Programming Support for Autonomous Underwater Vehicles

Gary Giger, Liping Xue
CSE Department
The Pennsylvania State University
University Park, PA 16802, USA
Phone: 814-865-9505
Fax: 814-865-3176
{giger,lxue}@cse.psu.edu
<http://www.cse.psu.edu>

Sekhar Tangirala
Unmanned Vehicle Systems
Department
Applied Research Laboratory/PSU
University Park, PA 16802
Phone: 814-865-6531
Fax: 814-865-1615
cxt148@psu.edu
<http://www.arl.psu.edu>

Mahmut Kandemir
CSE Department
The Pennsylvania State University
University Park, PA 16802, USA
Phone: 814-865-9505
Fax: 814-865-3176
kandemir@cse.psu.edu
<http://www.cse.psu.edu>

ABSTRACT

While there has been considerable research in the past to build Autonomous Underwater Vehicles (AUVs) taking into account issues such as weight, length, power supply, potential payloads and networking capabilities, programming missions for these devices remains a very important issue. This is because unless we can program these autonomous devices using a high-level programming language, the resulting code representing the mission will be messy, difficult to maintain, and more importantly error prone. Our goal in this paper is to discuss the mission programming software for AUVs being developed at the Pennsylvania State University. We will present the existing functionality of this mission programming software, which is integrated into a Mission Controller. We will also discuss enhancements to the existing mission programming software such as representing the mission programming language using a grammar; adding improved syntax checking as

well as more powerful language constructs, and propose a redesign of the mission controller that consists of a front-end and back-end to work more efficiently with these enhancements. In addition, we present an example to demonstrate how this system operates in practice.

Keywords

Mission Programming Language, Autonomous Underwater Vehicles, Real-Time Looping, Compilers.

1. INTRODUCTION

Autonomous underwater vehicles (AUVs) are designed to perform required, complex mission planning. While there has been considerable research in the past into building AUVs taking into account issues such as weight, length, power supply, potential payloads and networking capabilities (see for example [3, 4, 5, 8] and the references therein), programming missions for these devices remains a very important issue. Unless we can program missions

for these devices using a high-level programming language, the resulting missions will be messy, difficult to maintain, and more importantly, error prone. Considering the fact that many AUV-based operations are critical, buggy missions are very dangerous and a more systematic approach needs to be taken in order to reduce these errors.

There has not been much prior work on developing high-level programming languages for AUV missions. For example, Chappell *et al* [2] suggest expressing AUV missions as sequences of statements in a mission file that make up a Common Control Language (CCL) that is compiled and executed by the control systems of the AUV. Stokey *et al* [3] propose using a CCL containing commands for an AUV and data messages for sensors aboard the AUV to control one or more vehicles operating in an area. Davis and Brutzman [4] propose using a CCL that not only contains task level commands for the vehicles, but also includes meta-commands used for vehicle specific information and comments that will not hinder the mission. Davis [7] suggests using a context free grammar to implement an AUV control language.

A high-level AUV Mission Programming Language (MPL) was developed at the Applied Research Laboratory/Pennsylvania State University (ARL/PSU) to make the job of the operator easier and less error prone when creating a mission for AUVs. Missions written in this

high-level MPL are input to a hybrid-systems based mission controller, also developed at ARL/PSU [1]. While this simple high-level MPL proved adequate for initial testing and even for many operational missions of various AUVs, it gradually became evident that a more powerful method of specifying mission orders was necessary as the mission controller was used to tackle more challenging and complex missions. Some of the limitations were the lack of looping constructs and conditional statements, the lack of a formal and rigorous method of checking syntax of the missions as well as some semantics. These were identified as features that would convert the simple high-level MPL into a powerful and widely applicable mission programming language.

Our goal in this paper is to discuss the technical details of a high-level MPL that is currently being developed at ARL/PSU for the mission-level control of AUVs. More specifically, our discussion will center on the two components within the mission controller that directly processes this high level MPL: a front-end and a back-end. The job of the former is to read the user specified input mission written in this high level MPL, check it for syntax against a specified grammar, and map it to an intermediate form called the mission order file. The back-end reads the orders in this file and executes them handling any looping constructs and conditional

statements that might exist in the user specified mission. Each statement in the mission order file, while representing a complex mission order, may be treated as a simple primitive that is interpreted by the mission controller. One of the important features of our approach is that we can program the mission controller using a different high-level MPL just by modifying the grammar that is used in the front-end syntax checker; no modification needs to be made to the back end.

The rest of this paper will pursue the idea of expanding this high level MPL. Section 2 introduces the mission controller architecture including the existing parser and order specification and also proposes a new parser along with transforming the current high-level MPL into a BNF Grammar-based language to create a more powerful, rigorous, and flexible high level MPL. This section also discusses how this high level MPL differs from other conventional high-level languages. Section 3 presents an example to demonstrate how the improved mission controller operates with a given AUV mission. The last section provides concluding remarks by summarizing the current status of the system and the future of our work.

2. MISSION CONTROLLER ARCHITECTURE

2.1 Current Mission Controller Setup

A hybrid, model based mission controller has been developed for the control of AUVs at ARL/PSU [1]. The hybrid mission controller is organized hierarchically as shown in Figure 1. Each of the modules that make up the mission controller hierarchy is a hybrid system, and the entire mission controller is modeled as a set of interacting hybrid systems. Modules at any level may command other modules at that and lower levels and send responses to that and higher levels. All levels in the mission controller hierarchy may assign vehicle commands directly by placing appropriate commands in the shared database. At the lowest level of the hierarchy is the underwater vehicle (plant) along with the vehicle controllers (VCs). The vehicle and the vehicle controllers have a hybrid state-space (which might, in some vehicles, be a purely continuous state space), and serve as the plant for the higher-level mission controller (MC), which is also hybrid in nature. The vehicle controller and the mission controller communicate through an interface layer symbolically represented by MC2VC (mission controller to vehicle controller) and VC2MC (vehicle controller to mission controller).

As seen in Figure 1, the mission controller is organized in a three-tier hierarchy, and all communication between modules is restricted to event synchronization and shared data. Command events propagate down the mission controller hierarchy and response events propagate up the mission controller hierarchy via event synchronization. A particular module initiates an event and its recipients are controlled by an event dependency table that may be static or dynamic.

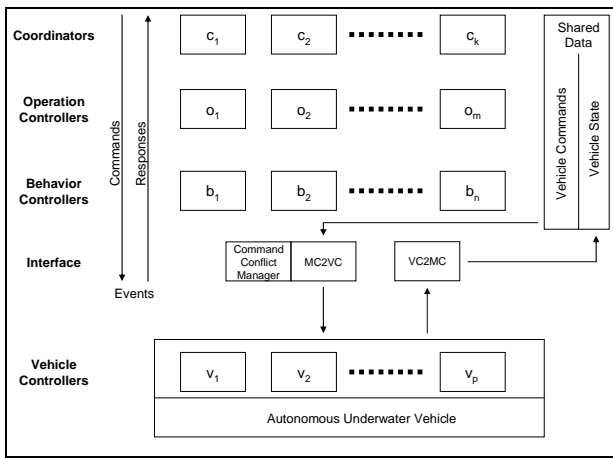


Figure 1 – Hybrid mission control architecture.

An event may also initialize parameters within modules in the hierarchy. Command events take the general form $do_m^n(command, params)$, where m is the requesting controller module, n is the receiving controller module, $command$ is the task to be performed and may take on values such as initialize, abort, etc., and $params$ are parameters and initial states for the receiving module. Similarly, response events are in the general form

$done_n^m(response, results)$, where $response$ is an indication of the completion of the commanded task and may take on values such as normal, abnormal, etc., and $results$ are parameters returned to the requesting module on task completion. The lowest level of the mission controller is comprised of Behavior Controllers, where a behavior may be thought of as a skill or ability that an autonomous system possesses, which enables it to perform specific mission tasks (thrive) while remaining safe (survive). Behavior Controllers directly interface with the vehicle controllers and are therefore vehicle-centric. They require executions of sequences of vehicle maneuvers. The middle level of the mission control hierarchy consists of Operation Controllers, where an operation represents a mission segment or phase that is integral to the completion of the overall AUV mission, and is user/mission-centric. These correspond directly to user supplied mission orders and command/sequence the behavior controllers to achieve their objectives. The highest level of the mission controller consists of the Mission Coordinators, which are responsible for sequencing and scheduling operations in order to complete the mission while ensuring the safety of the vehicle. Mission coordinators are typically of three types: Sequential, Interrupt-Driven, and Safety. The sequential coordinator is responsible

for executing a mission consisting of a sequence of operations; the interrupt-driven coordinator is responsible for executing a time or state-based interrupt driven sequence of operations; and a safety coordinator ensures safe operation of the vehicle. When an interrupt-driven operation is due, the currently executing sequential operation is suspended, if necessary, until the interrupt-driven operation has been executed. Sequential operation is resumed until the next (if any) interrupt-driven order is due. Interrupts are classified and prioritized so that some may have priority over sequential operations, while others do not and may therefore not be able to interrupt certain classes of sequential operations. The safety coordinator has priority over all other coordinators. When an unsafe operating condition is detected, the commands from the safety coordinator supercede all other commands and seek to move the vehicle into a safe region or abort the mission if necessary. These priorities are implemented by event dependencies and synchronization.

A mission is therefore defined as a coordinated sequence of operations, each of which is a sequence of behaviors, and possibly vehicle controller commands. Each behavior is, in turn, a sequence of commands to the vehicle subsystem controllers via the MC2VC interface. AUV state information is collected by sensors and periodically transferred by the VC2MC

interface to the shared database. This state information is made available to all modules in all levels of the mission controller hierarchy. Similarly, vehicle commands, assigned and manipulated by all levels in the mission controller are stored in the shared database and sent to the AUV by the MC2VC interface. Formally, let B denote the set of behaviors, O denote the set of operations, and V denote the set of vehicle subsystem controllers. A mission, m is defined as $m \in M \subset (O+V)^*$, where $(O+V)^*$ is the set of all sequences containing elements of O and V , and M is the set of all possible missions. Similarly, each operation $o_j \in (B+V)^*$, and each behavior $b_k \in V^*$.

The job of the existing front-end of the current mission controller is to read the user specified AUV mission written in this high level MPL. The mission controller contains a hand written parser, which first reads a mission order specification file which specifies the elements of various orders including their data types, their enumerated values if applicable, their units if applicable, and their default values if any. This order specification file is also used to specify critical elements, which must be specified in the mission order file, and alternatively used to specify optional elements that take on default values or are simply ignored if not specified explicitly. Order elements may also be specified as being critical conditional on specific values of

other order elements. In the user specified AUV mission file, each order is a block consisting of several elements and their values (and units) separated by delimiters. When a user specified AUV mission file is read, each order block is checked against the mission order specification for that order, as specified by the mission order specification file, for compliance. Properly specified orders are inserted into the appropriate mission queue (sequential, or timed which is the only interrupt mechanism at this time), while incompletely or incorrectly specified orders are flagged with error messages, which indicate the location and nature of the error. Once all of the orders have been read and the queues populated, mission controller is ready to execute the missions in a coordinated fashion as described above.

2.2 The New Setup of the Mission Controller

While this simple high level MPL proved adequate for initial testing and even for many operational missions of various AUVs, it gradually became evident that a more powerful method of specifying mission orders was necessary as the mission controller was used to tackle more challenging missions. The phrase “more challenging missions” encompasses both the addition of new orders (complex and simple) to the existing high level MPL and the ability of

the mission controller as a whole to handle more complex language constructs such as looping.

The hand written parser, although it works well with the current setup of the mission controller, does not provide a very convenient way to handle the addition of new orders. Each time a new order is added to the existing high level MPL, the hand written parser needs to have new code added to it in order to process the new orders. Adding new code to existing code always introduces the possibility of software bugs. The logic of the new code might be faulty and the resulting bugs might not be apparent at first, not to mention the possibility that the new code could break already existing functionality. In short, the maintainability of this code will become more difficult.

Since we needed to find a more practical way to add additional orders to the high-level MPL, this necessitated a redesign of the mission controller to create a front-end and a back-end. The original hand-written parser was replaced with code generated using the tools Lex and Yacc [9]. Lex and Yacc are used to perform parsing operations on a stream of input such as a text file. To take full advantage of these tools, a Lex specification was created that contained a set of regular expressions that occur in our high-level MPL and the existing mission order specification was transformed from its current form to the form

```

launch_order -> launch_start scheduled arrival_time
               adcp_init trim_init
launch_start -> START_ORDER : LAUNCH_ORDER
scheduled    -> SCHEDULED : order_type | <No Token>
order_type   -> SEQ_ORDER | TIMED_ORDER
arrival_time -> ARR_TIME : float time_smhu | <No Token>
adcp_init    -> ADCP_INIT : enum_fpn | <No Token>
trim_init    -> TRIM_INIT : enum_fpn | <No Token>
time_smhu    -> SECS | MINS | HOURS | UTC
enum_fpn     -> HOVER | PARTIAL | NONE

```

Figure 2 - A snippet of an order from the high-level MPL represented as a BNF grammar.

of a BNF grammar and added to a Yacc specification file. Figure 2 shows part of this BNF grammar that represents a launch order that is part of the high level MPL. Together Lex and Yacc were used to generate the C source code for the front end that replaces the existing hand written parser. Figure 3 shows a diagram of the new front-end that contains the new generated parser. Now, when a new order needs to be added or an existing order needs to be modified, only the Lex and Yacc specifications need to be changed and new code for the parser generated. This increases maintainability and reduces the incidence of software bugs. One only needs to verify that the BNF grammar in the Yacc specification and the regular expressions in the Lex specification are correct in order to eliminate any bugs that may appear in the generated source code.

In addition to adding new orders and modifying existing orders in the existing MPL, the level of complexity with regards to how these orders are executed also increases. With a conventional high-level programming language

such as C, when a line of code is encountered (e.g., printf()), it is executed once, and then, the execution moves on to the next line of code. If one wants to execute this line of code more than once, one would either need to specify this line of code more than once in the program or simply

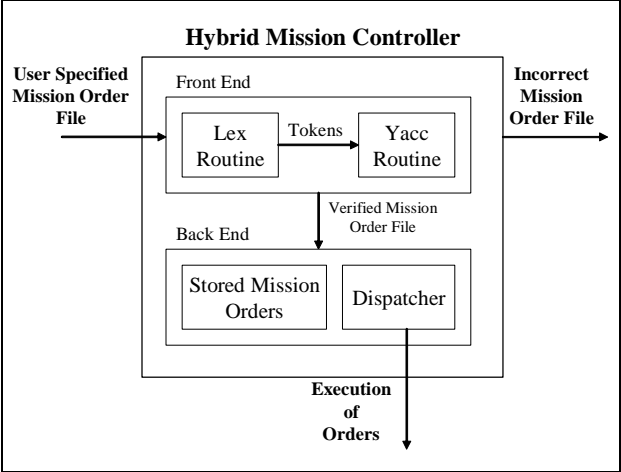


Figure 3 - The front-end and back-end components of the mission controller.

add this line of code inside the body of a loop. We ran into the same problem when creating complex mission files. A situation arose where we needed to execute a specific set of commands an *a priori* undetermined number of times. This paved the way for the addition of looping constructs as part of the high-level MPL.

To be able to handle loops, new additions are required in the back-end (also shown in Figure 3) of the mission controller as well. The looping mechanism added to the back-end of the mission controller can handle both single loops as

well as nested loops. An example of a user specified mission file that contains looping is shown in Figure 4. This example shows three nested loops and a fourth loop, which is independent from the other loops. With the addition of these looping constructs, new data structures needed to be added to allow the back end to handle these constructs. A Dependency

```

Launch Order
Orders
// While Loop 1
While (Condition1) {
  Orders
  // While Loop 2
  While (Condition2) {
    Orders
    // While Loop 3
    While (Condition3) {
      Orders
    }
    Orders
  }
  Orders
}
Orders
// While Loop 4
While (Condition4) {
  Orders
}
Orders
Rendezvous Order

```

Figure 4 - Looping constructs in the mission order file. Note that orders represent any of the orders that exist in the high-level MPL.

Graph was chosen to represent the structure of the looping constructs in the mission order file. Each node in the graph represents a loop that exists in the mission order file. The graph has directed edges that indicate which loop is nested inside of other loops. For example, if a node has a directed edge from itself to another node, then this indicates that the loop represented by this node is nested inside of the loop represented by

the other node. Figure 5 shows the dependency graph representation of the loop structure from Figure 4. Here *while loop 2* is nested inside *while loop 1* and thus node 2 has a directed edge pointing to node 1. Since *while loop 3* is nested inside of *while loop 2*, we have a directed edge from node 3 pointing to node 2. Note that *while loop 4* is independent from the other loops and, therefore, has no directed edges to any of the other nodes. The graphs that represent the loop structure in the mission order file may be

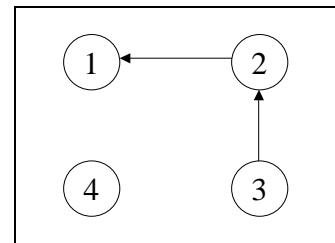


Figure 5 - The dependency graph representation of the looping constructs from Figure 4.

completely connected or disjoint depending on how the looping structure is defined inside of the mission order file.

The Dependency Graph is constructed when the mission order file is read by the back-end of the mission controller. The Dependency Graph is represented using an adjacency list, which is simply an array of records, where each record represents a node in the graph that contains three fields, the Loop ID, loop condition, and parent field. The Loop ID is a unique value that is used to identify the loop. A

loop ID is generated for each loop encountered by the front-end when the user specified mission file is checked for syntax and the intermediate file (the mission order file) is generated. The loop condition needs to be met in order for the loop to continue executing the orders in its body. The parent field is used to represent the directed edge from the current node to another node if the loop represented by the current node is nested within another loop. If the loop is not part of any outer loop, then this field is left NULL. As the back end reads the mission order file, a new node is added to the dependency graph for each new loop encountered. Since the dependency graph is represented using an adjacency list implemented as an array, the adjacency list can also be treated as a hash table. This is important when inserting a node into the graph. Treating the adjacency list as a hash table makes inserting a node very efficient (i.e., in $O(1)$ time) because a node's place can be found by a simple hash function such as $h(k) = k \bmod m$, where m is the size of the hash table. Perfect hashing is used for the hash table, which means that every node inserted into the hash table has its own unique location within the table. This also makes it very efficient when retrieving a node from the hash table since this operation can also be done in $O(1)$ time.

The dependency graph is used in conjunction with the looping mechanism that has been added to the back-end of the mission

controller. This looping mechanism is additional logic that handles all looping constructs contained within the mission order file. With respect to sequential orders contained within the body of a specific loop, the looping mechanism determines which of these orders will be executed next. Interrupt-driven orders such as timed orders can also be associated with a specific loop. When timed orders are read from the mission order file and loaded into the Interrupt-Driven queue, any loops that these timed order are part of are maintained in the Interrupt-Driven queue. When a loop that contains timed orders is entered, the looping mechanism activates these timed orders so the interrupt-driven coordinator can begin to check the criteria for these orders to execute.

2.3 Conventional High-Level Languages and Our High-Level MPL

Our high-level MPL is similar to other high level languages such as C in that it supports conditional statements, use of data types, variables, and looping constructs. But our language has a few key differences from the other languages. Conditional statements necessitate the definition of variables that map into specific variables within the mission controller representing the state of the vehicle and/or the controller. In this sense the concept of a variable here is more restrictive than that in a general

programming language, in that only variables which correspond to existing data structures within the mission controller are allowed.

Our language also supports the idea of default values for order elements. Some of the elements for particular orders are optional, that is, the user does not need to explicitly list these elements when specifying those particular orders. When the order is parsed and optional elements are omitted, the compiler will supply a default value for these omitted elements. The default values for optional elements listed in the grammar are stored in a separate configuration file. If the user requires a value other than the default value, then the user can explicitly list the optional element with a new value. The grammar in Figure 2 shows the element *arrival_time* as optional by including the string *ARRIVAL_TIME float time_smhu* or *<no token>*. Here the *<no token>* indicates that the element is optional and is the same as ϵ (the empty string) in a BNF Grammar.

While loop constructs are employed with structures similar to general programming languages with one important distinction. In a general programming language the loop condition is checked after the entire body of the loop has executed. In our high level MPL, on the other hand, the loop condition is checked after each individual order in the body of the loop has

finished. If the looping condition fails during one of these checks the remaining orders in the loop body, if any, are ignored and the loop is exited. Furthermore, an interrupt-driven order mechanism is inserted into the queue of the interrupt-driven coordinator when a specific loop is entered. This order mechanism also monitors the condition of the loop. When the condition fails, this order mechanism interrupts the currently executing sequential order and aborts this order and all other loops in the current loop body. The interrupt-driven order mechanism is used to ensure that when the looping condition fails, the currently executing order in the loop is terminated and both the order and the loop are exited gracefully. This type of looping behavior is needed in our application due to the execution time of certain mission orders. For example, let's say we have a survey order that takes several hours to execute and this survey order is part of a loop. If the loop condition is only checked after the entire loop body executes (as with a general programming language) and if the loop condition turns false part way through the execution of the order, this survey order will continue to execute even though the loop condition is false. This will not only waste available resources on the AUV (e.g. remaining power), but it might also affect how the remainder of the mission is executed or whether or not the remainder is executed at all. It is important that loop conditions are checked

periodically and frequently to ensure that the mission orders are executed and exited in a timely manner so AUV resources can be fully utilized.

3. AN EXAMPLE

An example that shows how a simple mission order file is read and processed by the Mission Controller will now be discussed. For simplicity, only “waypoint” orders will be used as sequential orders in this example; and also, a generic *Time Order* will be used to represent orders in interrupt-driven order queue. Previously, Figure 4 showed a skeleton structure for a mission order file that contained generic *orders*. Figure 6 shows a mission order file with the same looping structure as in Figure 4, but with actual waypoint orders instead of generic

```

Launch Order
Waypoint Order 1
// While loop 1
While (Condition1) {
    Timed Order 1
    Waypoint Order 2
    // While loop 2
    While (Condition2) {
        Waypoint Order 3
        // While loop 3
        While (Condition3) {
            Timed Order 2
            Waypoint Order 4
            Waypoint Order 5
        }
    }
}
Waypoint Order 6
// While loop 4
While (Condition4) {
    Waypoint Order 7
    Waypoint Order 8
}
Rendezvous Order
    
```

Figure 6 - Mission order file with sequential orders, timed orders, and looping constructs.

orders. Note that the waypoint orders specified in this mission order file are a simplified version of the actual waypoint order (i.e. no details such as elements of the waypoint order are shown for clarity).

This user specified mission order file is first read and verified by the parser in the front-end of the Mission Controller. Basically, the parser checks if the types of the values of the elements in each order are correct, whether or not the values of the elements are within prescribed

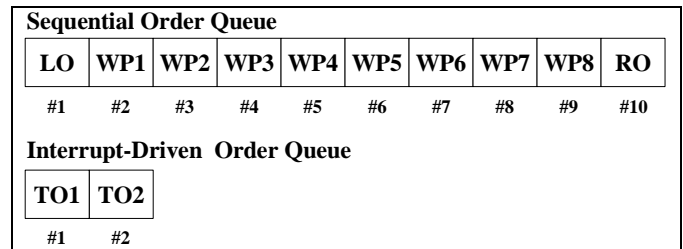


Figure 7 - Abstract representations of the sequential order queue and timed order queue along with the corresponding order number for each respective queue.

ranges, and for the existence of syntax errors. Figure 7 shows abstract representations of the queues after all of the orders, both sequential and timed, are loaded by the back-end of the Mission Controller.

As the orders are loaded into the appropriate queues, the Dependency Graph is also constructed. Figure 8 shows the tabular representation of the dependency graph based on the looping constructs from the mission order file from Figure 6, which includes the loop ID field,

Key	LoopID	Parent	Condition
1	1	-1	Cond1
2	2	1	Cond2
3	3	2	Cond3
4	4	-1	Cond4

Figure 8 - Tabular representation of the dependency graph from Figure 5. This representation closer resembles what the records look like when stored in the actual array data structure.

the parent field and the condition field as discussed earlier in Section 2.2. Once the queues in the Mission Controller are populated with the orders from the mission order file and the Dependency Graph is built, the mission controller can execute the mission by dispatching the orders from the queues. Referring to the current example, the very first sequential order that is loaded by the Mission Controller is the Launch Order. Figure 9a shows the state of sequential order queue before this order is loaded and Figure 9b gives the state of the sequential order queue after the launch order is loaded. Figure 9c shows the state of the sequential order queue when the current position of execution is inside the nested loop structure shown in Figure 6. Finally, Figure 9d illustrates the state of the sequential order queue after the execution of the nested loop structure. When an order contained inside the loop is executing, the orders associated with that loop are not removed from the sequential order queue until the loop has exited.

Timed orders are handled concurrently with the sequential orders. A dynamic set of

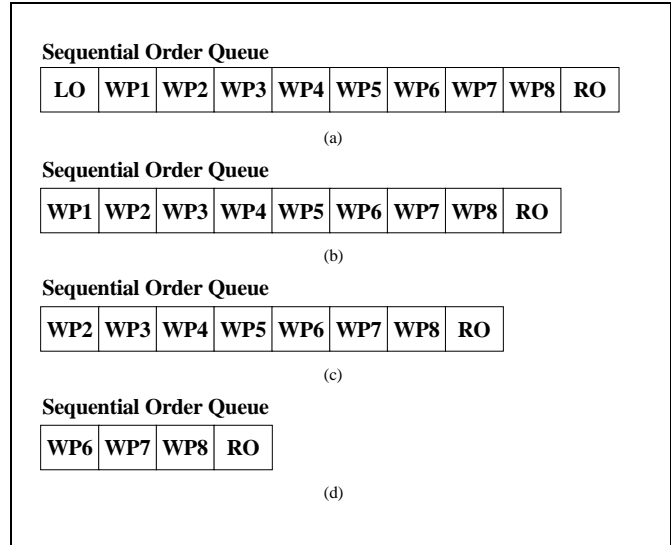


Figure 9 - (a) The state of the sequential order queue before loading the launch order (b) The state of the sequential order queue after loading the launch order. (c) The state of the sequential order queue after the first nested loop structure is entered (d) The state of the sequential order queue after the first nested loop structure has finished execution.

Loop IDs for timed orders is maintained that indicates which timed orders belong to the current execution block. In other words, when a loop is entered, its loop ID is added to this set of active loops. When a loop is exited, its ID is removed from this set of active loops. Maintaining such a set allows the interrupt-driven coordinator to determine which timed orders to load and which ones not to load. For example, if Waypoint Order 4 (Figure 6) was executing this means that we have entered *while loops 1, 2, and 3*. The dynamic set of active loops would contain Loop ID's {1,2,3}. The interrupt-driven coordinator will determine which order has the next earliest start time. If the next order selected was *Timed Order 2*, the interrupt-driven

coordinator would then check if this order is part of an active loop. Since *while loop 3* is active, then *Timed Order 2* would be loaded as the next timed order to execute by the interrupt-driven coordinator. If the current execution was on *Waypoint Order 3*, the set of active loop ID's would be {1,2}. If the interrupt-driven coordinator again would select *Timed Order 2* as the timed order with the next earliest start time, this timed order would not be returned as the next timed order to execute by the

interrupt-driven coordinator since loop 3 is not part of the set of active loop ID's. The Sequential Order Coordinator and the interrupt-driven coordinator both determine which orders need to be loaded next based on the set of active loop ID's and the looping constructs provided in the mission order file. These two coordinators continue to load the appropriate orders until there are no orders left in either order queue.

4. CONCLUDING REMARKS AND FUTURE WORK

Creating missions for AUVs can sometimes be tedious. Not only must one ensure that the missions are correct, but that the verification tools used are also correct in their functionality. This is why a high-level mission programming language (MPL) for AUVs makes the job of the operator easier and less error prone when creating a mission for these vehicles. The high-level language constructs can allow the user

to specify a mission without worrying about specific details of the vehicle. Furthermore, representing this high level language using a grammar and generating source code based on this grammar using the tools Lex and Yacc further reduces that chance of errors when compared to maintaining the source code for hand written parsers. The user only needs to confirm that the grammar is correct and once the grammar is correct all of the source code can be generated for the parser, which makes it easy for users to modify existing orders and add new orders to this high-level language. This characteristic along with the looping and conditional constructs turned our simple high-level MPL into a powerful and widely applicable MPL.

Future work includes designing a tool that will allow a user to graphically layout a mission using a diagramming tool and front end GUI (similar to [6]). Once the user is finished, this tool will translate the graphical representation of the mission to a text-based version and generate the mission order file. Currently our work concentrates on controlling one AUV. The next steps for this research will be the control of multiple vehicles. This could require creating a top-level mission order file, determining how to break up this mission into sub-missions and the assignment of these sub-missions to various vehicles in the group of AUVs.

REFERENCES

- [1] S. Tangirala, R. Kumar, S. Bhattacharyya, M. O'Connor, and L. E. Holloway. "Hybrid-Model based Hierarchical Mission Control Architecture for Autonomous Underwater Vehicles," *Proceedings of the 2005 American Control Conference*, Portland, OR, June 2005.
- [2] S. Chappell, S. Mupparapu, R. Komerska, and D. R. Blidberg. "SAUV II High Level Software Architecture," *14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 2005.
- [3] R.P. Stokey, L.E. Freitag, and M.D. Grund. "A Compact Control Language for AUV acoustic communication," *Oceans 2005 - Europe*, vol.2, no. pp. 1133- 1137 Vol. 2, 20-23 June 2005.
- [4] D. T. Davis, and D. Brutzman. "The Autonomous Unmanned Vehicle Workbench: Mission Planning, Mission Rehearsal, and Mission Replay Tool for Physics-Based X3D Visualizations," *14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 2005.
- [5] C. N. Duarte, C. Buzzell, G. R. Martel, D. Crimmins, R. Komerska, S. Mupparapu, S. Chappell, D. R. Blidberg, and R. Nitzel. "A Common Control Language to Support Multiple Cooperating AUVs," *14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 2005.
- [6] I. Woodrow, C. Purry, A. Mawby, and J. Goodwin. "Autonomous AUV Mission Planning and Replanning – Towards True Autonomy," *14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 2005.
- [7] D. Davis. "Automated Parsing and Conversion of Vehicle-Specific Data into Autonomous Vehicle Control Language (AVCL) Using Context-Free Grammars and XML Data Binding," *14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 2005.
- [8] S. S. Mupparapu, S. G. Chappell, R. J. Komerska, D. R. Blidberg, R. Nitzel, C. Benton, D. O. Popa, A. C. Sanderson. "Autonomous systems monitoring and control (ASMAC) - an AUV fleet controller," *Autonomous Underwater Vehicles*, 2004 IEEE/OES, vol., no.pp. 119- 126, 17-18 June 2004.
- [9] J. R. Levine, T. Mason, and D. Brown. "Lex & Yacc", *O'Reilly & Associates, Inc.*, October 1992.