A Lightweight Scripting Engine for the Slocum Glider

Hans Christian Woithe, Ulrich Kremer
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
Email: {hcwoithe,uli}@cs.rutgers.edu

***Abstract*— The Slocum Glider is a buoyancy driven Autonomous Underwater Vehicle (AUV) capable of deployments lasting several weeks or months. The layered control architecture used by the vehicle is difficult to program and restrictive. As part of previous work we have developed a more flexible programming framework capable of performing dynamic feature tracking. However, the gliders new and more computationally capable Linux Single Board Computer (SBC) results in additional energy demands.**

In this paper, we describe a lightweight scripting engine we have developed to allow the execution of code as part of the glider's existing layered control system. In this scheme, the SBC creates code and initiates remote code execution on the glider's stock infrastructure. In order to save energy, the SBC may enter a low power mode during the remote code execution. The SBC may be woken up periodically, or by specified events encountered during remote code execution. The resulting system can provide the enhanced computational capabilities of the SBC system, but only in situations where it is necessary, thereby potentially enabling significant energy savings.

The lightweight scripting engine is not restricted to our prototype system but can also be used on a stock glider. The engine provides an alternative programming infrastructure for marine science application programmers to implement and test novel algorithms for their vehicle, thereby enhancing the usability of a stock glider. Along with the engine, we have developed a higher level programming language and compiler, and showcase the flexibility of the system by using it to track a thermocline.

I. Introduction

The Slocum Glider is an Autonomous Underwater Vehicle (AUV) produced by Teledyne Webb Research (TWR) [1]. It is capable of prolonged missions lasting several weeks or months to gather scientific data from the world's oceans [2]. With AUVs becoming more commonplace, the complexity of the sensors and the need for robust and flexible control infrastructures for the platforms has also risen. In our previous work [3], we described our efforts to provide such a framework for the Slocum Glider and showcased its capabilities by performing dynamic feature tracking of a thermocline (a rapid change of temperature within a small column of water).

The infrastructure of [3] consisted of the integration of a Linux Single Board Computer (SBC) into the glider, a prototype domain specific language and compiler to program and control the vehicle from the SBC, and the necessary retrofitting of the glider's software on both of its two Persistor CF1 computing systems [4]. The objective was to provide the necessary groundwork to make the vehicle a more effective tool for researchers, and to enable complex algorithms that cannot be performed on the current computing infrastructure.

However, the addition of the SBC comes at the cost of an increase in power requirements. For example, if the SBC is used to power manage sensors, the employed management algorithm must ensure that it saves more energy than needed to run it on the SBC to justify its use. For data processing, it may be advantageous for the SBC to be powered only when enough data becomes available to perform the calculations. Finally, if the SBC is used to control the glider's flight, there may still be autopiloting opportunities where it can be powered off to let the legacy system control fly the vehicle. Thus, a system is needed that allows for the flexibility of the existing prototype system but at a lower energy cost so that the vehicle can continue to sustain long term deployments.

To fulfill this requirement we have developed a lightweight scripting engine, named `GLOC`, for the Slocum Glider. It is designed to operate on the glider's native processor so that researchers with standard vehicles can also develop new algorithms for the glider. The engine is implemented as a behavior for the vehicle's layered control system. As in our previous work [3], this is to ensure that other higher priority behaviors can override the actions requested by a script.

Because the language of the scripting engine is rather low level, we have also developed a `BASIC`-like higher level programming language and compiler called `GBASIC`. This sample language illustrates the flexibility of the engine and is a reference point for other future languages that may be developed to program the vehicle. Using this infrastructure we perform dynamic feature tracking of a thermocline which is not possible with stock gliders today. The investigation of different energy saving strategies using the SBC and the `GLOC` engine are part of our future work.

II. Background

The scripting engine we have developed for the Slocum Glider is part of the vehicle's control system. To gain a detailed perspective into the design and implementation of the engine, we describe the previous efforts which form the basis of our work. In Section II-A, the overall philosophy, lineage, and usage of the control system is discussed. The focus of Section II-B is a description of our prototype programming

framework that is still under development. Components of the framework are used by the engine, and the scripting engine itself may later be absorbed into this work.

A. Layered Control

Conceptually, the existing control system on the Slocum Glider has its roots in the Massachusetts Institute of Technologies' (MIT) Artificial Intelligence (AI) laboratory. There, Brooks' developed the layered control architecture [5] where a robot's control system is described as being decomposed into task achieving behaviors. These behaviors can perform complex tasks by gradually increasing their level of competence. This is accomplished by having more complex layers build on top of simpler, more robust lower layers. Higher level layers in this architecture inject data into the lower layers to suppress their normal data flow.

The described layered control architecture served as the basis for the control system used on the Odyssey II AUV developed as part of MIT's Sea Grant [6]. The implementation diverges, however, from the original design in that a particular behavior's arbitration of output commands is not restricted to only that behavior. A given layer in this architecture can instead take into account the results produced by other behaviors. For example, as in [6], an obstacle avoidance algorithm may try to avoid an object immediately ahead of the vehicle. The decision to steer left or right to avoid the object may not be important, as long as collision is avoided. Another behavior may in fact prefer a specific direction over another to avoid the shore nearby. This design can conceptually allow for the satisfaction of both behaviors in the control system.

The Slocum Glider's control software is a descendant from Odyssey's control system. Behaviors, such as a *dive_to* and *climb_to*, are written in mission files, from highest to lowest priority. An example of a behavior with high priority is the *abend* behavior which assists in keeping the vehicle safe. Safety behaviors include guarding the glider from diving too deeply or staying underwater for too long without communication with a control center.

At the start of a mission a flight engineer will decide which mission file is to be loaded and executed on the vehicle. The mission file is then parsed and used to initialize the layered control system of the vehicle. The vehicle, at each four second control cycle, will begin the arbitration of what to do next by executing the bottommost active behavior. The output of a behavior are commands which will be passed to the next higher level in the layered control system. The final set of commands to be executed during the later stages of the cycle is produced by the topmost behavior and is the result of the arbitration of all active layers.

The layered control architecture performs well for many applications but it has some disadvantages. It can be difficult to create new missions because it is not always clear how the layers interact since behaviors can produce different output commands based on their state. A given behavior can be a construct of multiple sub-behaviors which themselves have state. Certain portions of a mission may need a specific subset of behaviors to be activated to produce the desired effect. Therefore, writing missions is error prone and requires a significant verification, testing, and debugging effort.

Another important limitation when writing missions for the glider is that users are limited to the behaviors supplied by the manufacturer. Although core behaviors may exist to achieve a particular task, the proper coordination of the behaviors is not easily accomplished. As a result of these programmability issues we have begun the development of a domain specific language for the Slocum Glider [3].

B. Previous Work

The current glider programming framework has its limitations. Because writing new missions is not straight forward, users generally limit themselves to using existing missions created by the manufacturer. A deployment's parameters are specified through the use of mission argument files which are loaded alongside the actual mission file. In many cases, little or no change is required to the actual mission file itself, but merely its accompanying argument files.

Although some users venture into writing their own missions, few are likely to write their own new behaviors for the vehicle's layered control system. To develop for the system, a tool chain for the glider's Persistor CF1 processors is required [4]. Any source changes that need testing must first be compiled, and the resulting binary must be transferred and flashed to the Persistors. Because of the complexity and difficulty of programming the glider, we are developing a new programming infrastructure for the vehicle.

In [3], we introduced our initial prototype framework for the Slocum Glider. The prototype included a new domain specific programming language and compiler, software hooks into the existing control system to gain access to the vehicle, and a Linux Single Board Computer (SBC) with a runtime system to generate commands for the software hooks. An overview of the design is shown Fig. 1. To illustrate the new capabilities possible using this novel architecture, we performed dynamic feature tracking of a thermocline off the coast of New Jersey.

The glider contains two 16 MHz Persistor CF1 processors: the flight controller is responsible for navigating the vehicle as instructed by a flight engineer, while the other science processor is responsible for interacting with and logging the data produced by scientific sensors. The two CPUs can communicate via a serial connection to exchange flight and sensor information. The hooks for the prototype involved modifications to the software residing on both Persistors.

A driver on the science processor, also known as a proglet, was added to interact with the SBC over a serial connection. The protocol between the SBC and the proglet lets the SBC read, write, and request data to and from the glider using the existing communication infrastructure of the Persistors. In this scheme, simply put, the science processor acts as a proxy to read and write to the memory of the flight Persistor.

On the flight controller, a new hook behavior was introduced. Given that the proglet on the science computer allows the SBC to write to the glider's memory, a mechanism must

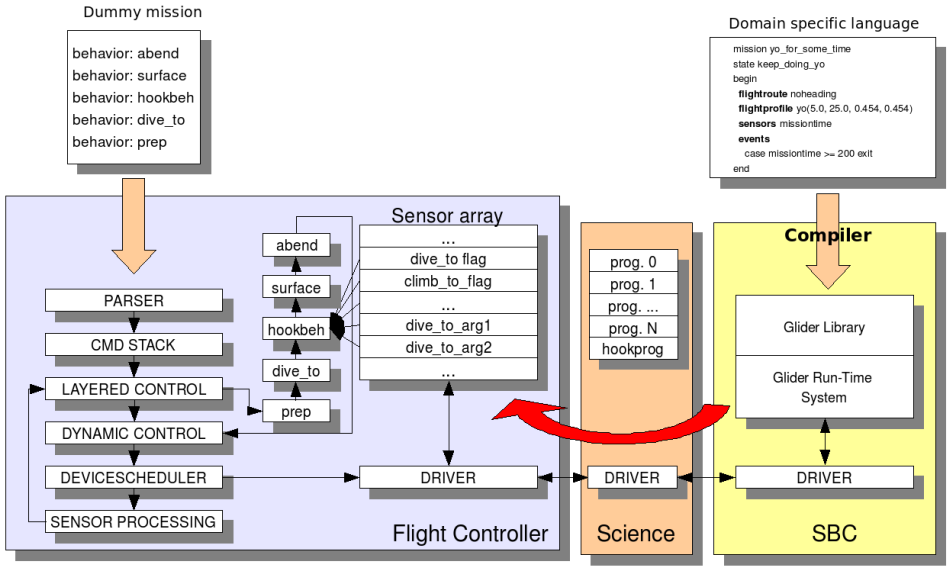


Fig. 1. The overall design of the prototype software infrastructure as presented in [3]. The hooks into the existing system can been seen as part of the layered control system on the flight controller and as a hook proglet on the science processor.

exist on the flight controller to interpret the commands sent by the SBC. When active, the hook behavior will execute as part of the layered control's arbitration process. When specific flags and parameters are sent by the SBC, the hook behavior will dynamically create and execute the SBC requested behaviors in each control cycle. More specifically, these behaviors are sub-behaviors of the hook behavior much like a `yo` behavior consists of a sequence of `dive_to` and `climb_to` behaviors.

The described scheme allows the vehicle to be controlled by the runtime system on the SBC while still making use of the existing control system and safety checks. For example, the SBC may request the glider to dive below the crush depth of the vehicle, however, the abend behavior which is still at a higher layer of priority will override the outputs produced by the rogue hook behavior. This retrofitted system opens the door for applications on the Linux based computing environment to dynamically control the vehicle. It has been used to successfully detect and track a thermocline, and is the basis for the scripting engine we have created.

III. Lightweight Scripting Engine

The current programming environment for the Slocum Glider is limited since users are confined to the behaviors produced by TWR. Creating new mission files can also be cumbersome, because of the complex interactions between behaviors. We hope to improve the programmability of the glider as well as add additional functionality that will make the vehicle an overall more effective tool. By creating a scripting engine, we aim to increase the scope of applications that can be performed on the vehicle.

The initial goal for the development of the scripting engine was to interact with the infrastructure described in Section II-B. However, an additional goal is to provide application programmers with an alternative mechanism to easily developing new algorithms for the vehicle. This is particularly useful during the simulation and testing phases of newly designed algorithms where the engine can act as a test bed before the algorithm is independently created, for example, as its own behavior.

The scripting engine, like the hook behavior, is implemented as a behavior and resides in the layered control system. It is therefore able to take advantage of some of the safety features present in the layered control system. The `GLOC`[1] language resembles a simple assembly language and is inspired by `ILOC` [7], an intermediate language for optimizing compilers developed at Rice University. Currently, `GLOC` has over thirty instructions, including instructions to load and store data to and from the glider's sensor list and the engine's registers, perform mathematical and logical operations, produce output, as well as perform jumps and conditional branches to labels.

Although an assembly level language is typically considered to be more difficult to program than a higher level language, it has several advantages. Parsers for complex languages often require a more complex set of tools and libraries to implement them. Some parsing techniques are also memory intensive and would use too much of this scarce resource on the vehicle's 1MB Persistor processor. In addition, the size of the code base may be a concern since larger codebases are typically harder to maintain and take longer to debug.

Due to the aforementioned reasons, the design decision was to make the scripting language very simple. The code base of the core of the engine itself is compact, measuring under 600 lines of code including documentation. Although this does not necessarily ensure reliability, the engine has thus far been easy to maintain. Another advantage is that it is simple to parse and

[1]GLOC: Glider intermediate Language for Optimizing Compilers

```
 1  nregs 8
 2  nlbls 7
 3  ninstr 27
 4  label 0
 5  loads 444,r0
 6  loadd 15.0,r1
 7  cmpgt r0,r1,r2
 8  cbr r2,3,4
 9  label 3
10  loadi 128,r3
11  stores r3,1387
12  jumpi 1
13  label 4
14  yield
15  jumpi 0
16  label 1
17  loads 444,r4
18  loadd 5.0,r5
19  cmplt r4,r5,r6
20  cbr r6,5,6
21  label 5
22  loadi 0,r7
23  stores r7,1387
24  jumpi 2
25  label 6
26  yield
27  jumpi 1
28  label 2
29  yield
30  jumpi 2
```

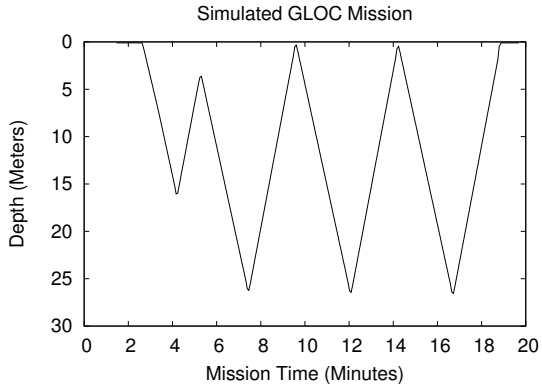


Fig. 2. A sample three yo mission executing a GLOC script. The glider mission (not shown) is instructed to dive and climb three times between 2–25 meters. The script behavior instead attempts to first fly a single yo between 5–15 meters before relinquishing control and letting the other yo behavior complete its mission. Both behaviors are active at the same time, with the GLOC scripting engine at a higher priority in the layered control stack. The resulting simulated flight path is shown to the right.

interpret, and has a small memory and processing footprint. The exact memory and processing requirements are dependent on the script being executed, but all of our experiments so far have only required a few kilobytes of memory and have added at most 30 milliseconds to the glider's four second control cycle.

A sample of a GLOC script is listed in Fig. 2 along with the simulated flight profile of the vehicle's mission. The example illustrates the interactions between a mission's behaviors and the behaviors induced by a GLOC script. The glider mission specifies a sequence of three yos, where a yo consists of a dive and climb operation. Instead, the GLOC script first instructs the glider to perform a single yo between 5–15 meters. In the layered control architecture, the script behavior is at a higher level than the yo behavior and thus supersedes the yo. As shown by the flight profile, the script successfully accomplishes its task and then lets the glider proceed with the rest of the mission.

When a glider mission is executed with the GLOC behavior, the engine first loads and allocates the memory required by the specified script. The beginning of the script file, lines 1–3 of Fig. 2, specifies the number of registers, labels and instructions that the script will use. This allows for the behavior to statically allocate all required private memory at one time, and makes deallocation of the memory easy at the end of a mission. This mechanism follows the general design pattern used for behaviors throughout the glider software.

Labels in GLOC are numbered and serve as targets for jump and conditional branch instructions. These instructions allow for control flow to occur in the scripting engine and are the building blocks for conditional statements such as an if and loops such as a while. The jump targets may be specified directly by a number or indirectly through a value contained inside a register. Conditional branch instructions jump to the first label when the condition holds true and jump to the second label if false.

GLOC is a language for a simple reduced instruction set computing (RISC) architecture [8]. A register contains data values such as floating point numbers or integers. Mathematical and logical instructions require their input values to be in registers and write their output target registers. Registers can be populated with values using a number of load instructions. The loadd and loadi instructions of Fig. 2 assign a floating point and integer values to their given target registers, respectfully. A loads instruction however loads a register with data from the glider's sensor array. The sensor array, is part of the pre-existing glider behavior programming architecture. The load instruction in line five assigns the value of the vehicle's current depth to register zero. This is because the 444th sensor variable in the glider's sensor array is designated for the depth information. Writing data to the sensor array is possible via the stores instruction.

The scripting engine is able to gain flight control of the vehicle by using the hook behavior. The engine, through the hook behavior, can dynamically create and execute sub-behaviors by setting the appropriate flags and parameters in the glider's sensor array, much like the SBC does in our previous work [3]. Lines 10 and 11 correspond to such an interaction between the components. Sensor 1387 is a variable that is checked periodically by the hook behavior to see which sub-behaviors are to be created. A value of 128 activates a climb_to behavior whose parameters in this particular case have been predefined in the glider's mission file. Other behaviors can be simultaneously activated by setting appropriate flags through the hook behavior's variable.

The GLOC engine is lightweight and can quickly execute scripts as part of the layered control system. However, the exact overhead is reliant on the code being executed. It is currently the responsibility of the programmer to ensure that only a limited amount of code is executed as behaviors are not preempted by the glider software. In GLOC, the yield instruction informs the engine that the program wishes to relinquish execution for the current control cycle. It is in this manner that cooperative multitasking is achieved. The user

```
1  label: state1
2  if m_depth > 15.0 then
3    SCI_RUHP_BEHS = 128
4    goto state2
5  endif
6  yield
7  goto state1
8  label: state2
9  if m_depth < 5.0 then
10   SCI_RUHP_BEHS = 0
11   goto state3
12 endif
13 yield
14 goto state2
15 label: state3
16 yield
17 goto state3
```

Fig. 3. A sample GBASIC program to perform a single yo between 5–15 meters. This program is the source of the compiler generated GLOC script shown in Fig. 2

must be aware that taking a large quantum of execution could lead to undesired control cycle overruns.

We believe that our scripting environment is flexible and robust, and will increase the scope of applications that can now be performed on the glider. The engine is also not restricted to performing tasks independently, but can collaborate on computation and data processing tasks with the SBC. One of the main motivations for GLOC is to reduce the energy consumption of the vehicle by alleviating the need to have the Linux SBC be powered at all times. Powering off the SBC may be desirable in many scenarios. If an application on the SBC does not require a large processing workload for a portion of its execution, the task could instead be executed remotely by the scripting engine. In this scenario, the SBC can enter a low power mode or power off entirely until it is needed again. A transfer into low power mode may also be profitable when data processing is not worthwhile until a large data set has been acquired by onboard sensors.

In combination with the SBC, tasks can be performed concurrently with the scripting engine. The design of the original prototype has changed since [3] in that the SBC can now communicate directly with the flight controller and no longer requires the science computer to act as a proxy. This direct connection between the two computers is more robust and an overall more sound design. We hope that the programming infrastructure will take full advantage of the scripting engine in the future, and allow for the automatic code generation of lightweight tasks to be generated, transferred, and executed.

IV. GBASIC Language

To showcase the capabilities of the scripting engine and to improve the programmability of the scripting system we have created a subset of a BASIC-like programming language, called GBASIC. Although the development of our domain specific programming language is still in progress, the implementation of GBASIC can serve as a reference point for other language designs and compilers that target GLOC.

The compiler for GBASIC was implemented using Python and the Python Lex-Yacc (PLY) toolset. This toolset is comparable to standard Lex and Yacc tools used in compiler construction [9]. The code base for the compiler is small with approximately 700 lines of source code. As the language develops and more BASIC inspired statements are added, the code size will slightly increase.

Currently, the compiler does not perform any optimizations on the GLOC code to reduce either its code size or memory consumption. However, type checking and casting is implemented since it is the responsibility of the compiler, or the programmer, to ensure that the GLOC code running on the actual vehicle is safe.

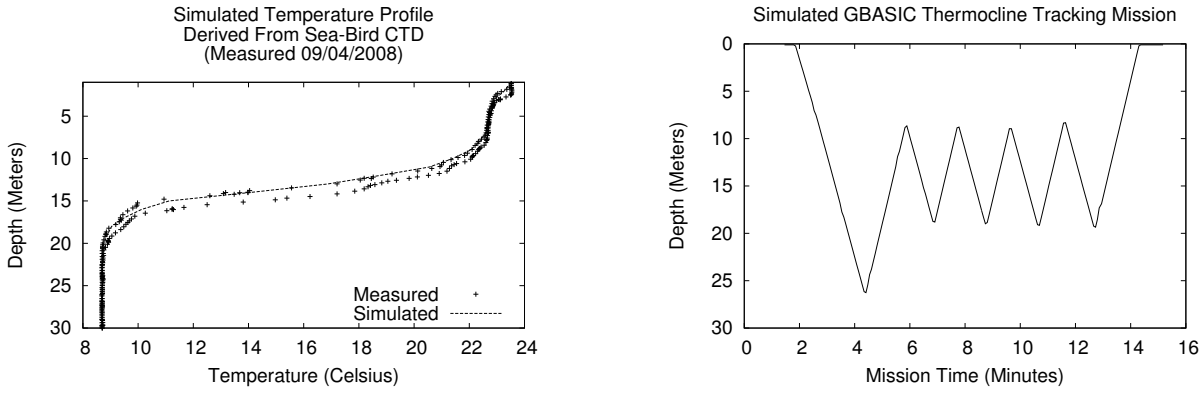
The GBASIC language has support for variables and one dimensional arrays of integer or floating point values. Like GLOC, GBASIC can express mathematical, relational and logical operations. The language contains the `label`, `goto` and `if` statements, useful to control the flow of execution. Although not yet implemented, the creation of `while/wend`, `do/loop`, and `for/next` loops should be trivial as they can be constructed from the already built constructs.

An example of a GBASIC program is listed in Fig. 3. The GLOC code presented in Fig. 2 is in fact the output code generated by our GBASIC compiler of the program in Fig. 3. The higher level language is more readable and thus makes it easier to debug. Labels, for example, are not just numbers as in GLOC, but can have descriptive names. Built-in vehicle variables such as `m_depth` can also be called directly by their name, as specified in the glider's `Masterdata` documentation. Although GBASIC may not be the most appropriate language to develop programs for the Slocum Glider, it illustrates that higher level languages can be constructed for our scripting engine.

V. Thermocline Tracking

Similar to our previous work on thermocline tracking [3], the new scripting engine adds functionality to the vehicle that is not available on a standard glider. We have implemented the thermocline tracking algorithm of [10] in GBASIC. The compiled GLOC code was executed by the scripting engine in a simulated thermocline tracking mission.

In [10], Petillo et al. developed a thermocline tracking algorithm for use within the MOOS-IvP autonomy system [11]. The algorithm collects temperature and depth data from a Conductivity, Temperature, and Depth (CTD) sensor and places the readings into depth bins. In our implementation, one meter depth bins are used. When a dive or climb leg has been completed by the vehicle, the depth bins are averaged. The vertical derivatives, the change of temperature over the change of depth, are then calculated for each bin. The average of the vertical derivatives is used to determine the upper and lower bounds of the thermocline. Any depth bin whose vertical derivative is greater than the average derivative is considered to be part of the thermocline. The algorithm requires an initial



(a) (b)

Fig. 4. (a) Water column temperatures acquired with a Sea-Bird Conductivity, Temperature, and Depth (CTD) profiling sensor; a simulated representation of the column's temperatures derived from the measured data. (b) Flight profile of a simulated glider tracking the thermocline in (a).

dive profile and periodic resets of the depth bin data to ensure variations of the thermocline are successfully detected. For our evaluation, however, we do not perform resets as the thermocline data is simulated.

The vertical temperature profile used as the basis for the data in the simulation is shown in Fig. 4(a). The water column was measured using a Sea-Bird CTD sensor, and the thermocline shown was tracked using a Slocum Glider equipped with our previous prototype system [3]. The `GBASIC` code to perform the simulated tracking mission was under 130 lines of code. This included data and `GBASIC` code used to fake the simulated temperature profile of Fig. 4(a). The non-optimized compiled code executed by the scripting engine was just over 300 lines of `GLOC`. The resulting flight profile which successfully performed tracking of the thermocline is shown in Fig. 4. Algorithms, like the discussed thermocline tracking algorithm, can be easily implemented using our new scripting engine and opens the door to a world of new applications for the glider that were not possible before.

VI. Conclusion

In this paper we have described the design and implementation of a lightweight scripting language for the Slocum Glider. The intent of the scripting language was twofold: to enable marine science application programmers to develop novel algorithms within the context of a legacy glider, and to provide an alternative program execution mechanism within our Slocum Glider programming architecture that includes a Linux SBC. In the latter context, the scripting engine may be used to conserve energy by offloading trivial execution tasks from the SBC to the glider computer, thereby allowing the SBC to be powered down.

The `GLOC` scripting engine is implemented as a behavior and can make use of another behavior we have created to dynamically instantiate sub-behaviors as part of layered control. `GBASIC`, a higher level language and accompanying compiler were designed and implemented to illustrate the potential of `GLOC` as an intermediate program representation. A thermocline tracking application, programmed in `GBASIC`, showed the effectiveness of our discussed approach. Although not yet fully feature complete, the described work is a step toward making the Slocum Glider a more effective research tool.

VII. Future Work

The scripting engine has not yet been implemented as part of our Slocum Glider simulation infrastructure [12], [13], Integrating a `GLOC` interpreter within the simulation environment could provide another way for programmers to test their programs. Additionally, when compared to the TWR's simulator, it can run faster than real time. This could greatly reduce the time required for debugging, as whole missions can be simulated in seconds or minutes, rather than days or months. Although it is not a complete replacement to executing the code in a real glider, it can provide the user with better insight into developing code.

Although the scripting engine supports some generalized functions to produce output, it is not enough to support the logging of complex sensors. Their primary purpose instead is to assist in the debugging of user programs. Currently, when sensors or variables must be logged, they are written to the glider's sensor array. The vehicle will then record the values of the array within each control cycle, usually every four seconds. The existing system is also limited to being only able to record floating point values. Having proper support for data logging in the engine is essential since it may be required by future applications.

Thus far, the glider scripting engine has only been tested on the Shoebox simulator and on the glider itself while still on the bench. A Shoebox simulator contains the essential glider electronics needed to perform glider simulations, and is named after the shoe box sized container that the electronics are housed in. We hope to perform additional experiments both in and out of the water. Out of water tests before the sea trials will continue to improve the robustness of the infrastructure. Initial sea trials will require a buoy to be attached to the glider

via a rope until is determined safe enough to be set free. First, simple dive segment manipulation tasks will be tested, followed by short thermocline tracking missions.

When the core of the scripting engine has independently completed its open ocean trials, the data and control interactions between the engine and the SBC need to be tested and deployed. We hope that this remote execution of code will enable considerable energy savings by instructing the SBC to sleep until an event is triggered by the engine. For example, the scripting engine may use a limited amount of sensory input to detect an algae plume. When detected, it may decide to power up the SBC as well as more advanced sensors to sample at a higher data rate and process the data in real-time. To quantify the energy savings, we plan to measure the power consumption of components of the vehicle, including the SBC, using the power measurement infrastructure we have developed as part of our previous work [12].

Finally, we would like to add additional safety precautions to the system. This may be accomplished in several ways, such as a preprocessor that performs safety checks before any of the `GLOC` code is ever executed, or by a compiler. Not only can a compiler make it easier to program for the engine, by compiling a higher level language into `GLOC`, it can alleviate some concerns regarding the safety of some aspects of the code by only allowing safe programs to compile. An optimizing compiler that reduces code size or memory usage is also in the planning stage.

ACKNOWLEDGMENTS

This research has been partially funded by NSF awards CSR-CSI #0720836 and MRI #0821607. We would also like to thank David Aragon, Tina Haskins, Chip Haldeman, Oscar Schofield and Scott Glenn from the Institute of Marine and Coastal Sciences at Rutgers University for their continued support.

REFERENCES

[1] Teledyne Webb Research, "Slocum glider," Falmouth, MA, http//www.webbresearch.com/slocum.htm.

[2] O. Schofield, J. Kohut, D. Aragon, L. Creed, J. Graver, C. Haldeman, J. Kerfoot, H. Roarty, C. Jones, D. Webb, and S. Glenn, "Slocum gliders: Robust and ready," *J. Field Robotics*, vol. 24, no. 6, pp. 473–485, 2007.

[3] H. Woithe and U. Kremer, "A programming architecture for smart autonomous underwater vehicles," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2009. IROS 2009.*, October 2009.

[4] Persistor Instruments Inc., "Cf1 computer system," Marstons Mills, MA, http://www.persistor.com.

[5] R. Brooks, "A robust layered control system for a mobile robot," vol. RA-2, no. 1, March 1986, pp. 14–23.

[6] J. Bellingham and J. Leonard, "Task configuration with layered control," in *IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, CA, May 1994.

[7] K. Cooper and L. Torczon, *Engineering a Compiler*. San Francisco, CA: Morgan Kaufmann Publishers (Impring of Elsevier Science), 2008.

[8] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Francisco, CA: Morgan Kaufmann Publishers (Impring of Elsevier Science), 2007.

[9] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley, 2007.

[10] S. Petillo, A. Balasuriya, and H. Schmidt, "Autonomous adaptive environmental assessment and feature tracking via autonomous underwater vehicles," in *IEEE Oceans 2010 Conference*, Syndey, Australia, May 2010.

[11] M. R. Benjamin, H. Schmidt, P. M. Newman, and J. J. Leonard, "Nested autonomy for unmanned marine vehicles with moos-ivp," *Journal of Field Robotics*, vol. 27, no. 6, pp. 834–875, 2010.

[12] H. C. Woithe, I. Chigirev, D. Aragon, M. Iqbal, Y. Shames, S. Glenn, O. Schofield, I. Seskar, and U. Kremer, "Slocum glider energy measurement and simulation infrastructure," in *IEEE Oceans 2010 Conference*, Syndey, Australia, May 2010.

[13] H. Woithe and U. Kremer, "An interactive slocum glider flight simulator," in *IEEE Oceans 2010 Conference*, Seattle, USA, September 2010.