Graphical State-Space Programmability as a Natural Interface for Robotic Control

Junaed Sattar, Anqi Xu, Gregory Dudek, and Gabriel Charette

***Abstract*— We present an interface for guiding and controlling mobile and underwater robots that combines aspects of graphical trajectory specification and state-based programming. This work is motivated by common tasks executed by our underwater vehicles, although we illustrate a mode of interaction that is applicable to mobile robotics in general. The key aspect of our approach is to provide an intuitive linkage between the graphical visualization of regions of interest in the environment, and activities relevant to these regions. In addition to introducing this novel programming paradigm, we also describe the associated system architecture developed onboard our amphibious robot. We then present a user interaction study that illustrates the benefits in usability of our graphical interface, compared to conventionally established programming techniques.**

I. Introduction

This paper describes an approach and software architecture for the programming and control of underwater and terrestrial robots. Our presented technique encompasses what we believe to be a novel variation of conventional mechanisms for specifying robot tasks. We examine this using a concrete implementation and an illustrative user study, and have also tested the approach in the field. Our primary focus is to provide a graphical tool that allow robot programmers and designers to generate plans of moderate complexity that combine target trajectories and a set of activities and procedures to be followed. In particular, plans for mobile robots often entail location-specific tasks, and thus require both procedural as well as geometric (or other positional) information. For example, one might seek to have a robot circumnavigate a set of objects in the environment, collect data in certain distinct locations, and remain alert for further instructions while the plan is being executed. One of the canonical tasks for our underwater robotic vehicle, shown in Fig. 1, is to take pictures and video clips while surveying underwater coral reef structures.

Standard task specification approaches permit plans with actions conditioned upon position or other state variables, but they are often specified in a non-intuitive structure that makes visualization of the relationship between state values and corresponding actions difficult to appreciate. This paper introduces an alternative methodology and an associated visual mechanism for specifying these types of plans in a practical and natural manner. Our objective is to provide a general approach to assist in robot programming and task specification.

The authors are with the School of Computer Science, McGill University, 3480 University Street, Montréal, QC, Canada H3A 2A7 {junaed,anqixu,dudek,gchare}@cim.mcgill.ca

Fig. 1. Our second generation amphibious Aqua robot exploring a populated reef environment.

Task-based robot programming often involves taking specific actions in certain locations or moving to places where events are expected to occur. Existing programming environments reflect these task specifications through three established interfaces: (1) a programmatic interface for writing procedural or functional code in a standard computer language, (2) a Graphical User Interface (GUI) for specifying trajectories or for direct tele-operation, and (3) a graphical representation of the program flow (such as an execution flowchart editor). Each of these approaches has its particular domain of application, and although they are sometimes used in combination, unfortunately these implementations often correspond to graphical front-ends for fundamentally textual programming activities. Specifying geographically-dependent tasks using existing programming environemnts is accomplished by using mechanisms which amount to case-based code execution, or, in some cases, proof-theoretic conditions.

In contrast to the aformentioned approaches, this paper proposes an alternative scheme which explicitly ties code execution to regions in space using a graphical interface. By allowing users to visually indicate regions in the environment (or more generally, in the robot's state space) and by associating these coordinates with blocks of code, positionally-dependent execution becomes very easy to specify, inspect and debug. For example, to take a picture at a particular location, one simply has to attach a picture-taking script to a target region of interest within a spatial map of the robot's surroundings. Likewise, when an amphibious vehicle

transitions from water onto land, the change in gait behaviour can be triggered based on values of location or depth. More generally, by explicitly visualizing the connection between spatial layout and position-dependent activites, our programming paradigm assists with the separation of concerns in the context of software development[1].

A secondary objective of our graphical programming paradigm is to provide a simple scripting interface as a software wrapper to low-level control functions. This requirement helps in minimizing the amount of code needed to specify high-level actions associated to particular locations. Our implementation also provides a network-based access protocol to this scripting interface, which allows for platform-independent remote robot control.

We extended the system architecture of our amphibious robot to include the proposed graphical state-space control scheme. This addition is particularly useful because specifying trajectories and tasks for a robot with 6 degrees of freedom is tedious and error-prone using existing interfaces such as locomotive-based control and tele-operation methods. A common phenomenon in robot programming, especially as related to sensor-based action planning, is the coupling of sensor processing, pose estimation and control activities. This activity is intrinsically cross-cutting, which has been demonstrated to be an impediment to userstandable and maintainable code [5]. Our graphical scheme has been successfully tested in real-world marine environments on our amphibious vehicle. In particular, we used our interface to both collect sensor data and trigger condition- and state-dependent task execution.

Finally, we have conducted a simplified human interaction study to compare our proposed graphical approach to conventional textual program development. Our results illustrate the various quantitative advantages of using the graphical state-space programming paradigm for specifying plans composed of location-dependent tasks.

II. Related Work

Many mobile robots (and in particular exotic variants such as underwater and flying vehicles) are programmed directly using hardware libraries and low-level Application Programming Interfaces (API). The software foundation for our own amphibious vehicle, as well as for many other state-of-the-art robotic systems, is built on top of the RoboDevel and RHexLib libraries. Unfortunately, programmers working with these libraries directly must first write a slew of basic functionalities (such as object detection and avoidance) before they can develop more sophisticated behaviour. Our proposed methodology automatically handles these mechanisms using different modules in the background, so that users can swiftly develop high-level and complex behaviours.

The Player/Stage/Gazebo robot development package [4] also provides commonly-used basic behavioural functions. The Player server provides a standardized interface to a wide range of robot sensors and actuators over a network socket. This server is often accompanied by the Stage and Gazebo applications, which are visual environments for 2-D and 3-D robot simulators, respectively.

As an established successor to the Player library, the Robot Operating System (ROS) software suite [3] provides a framework for message passing, interaction scheduling and code interconnection. This development environment is also targeted at building mechanisms and abstraction tools to facilitate the task of robot programming, but with a much larger scope and more amorphous goal than the approach presented by this paper. The core visualizaton and graphical control mechanisms provided by Player/Stage/Gazebo and by ROS are a superset of what we seek to provide, although they do not have high-level scripting API and lack the ability to graphically bind code execution to regions in state-space.

The Microsoft Robotics Developer Studio (RDS) [6] (formerly known as Microsoft Robotics Studio) is an Integrated Development Environment (IDE) for robotics applications. This extensive software package includes a hardware abstraction layer, a module-based communication and interaction protocol, a visual programming interface for enthusists, and a visual simulation environment. One of the most impressive features of RDS is its extensive ability of operate with different hardware platforms, rivaled only by a handful of other software suites such as Player and ROS. This package includes both a visual programming interface and a traditional object-oriented lower-level interface. Perhaps as a direct result of its ambitious range of functions, the desire to provide fine-grained control, and the generalized application domains it targets, this rich yet complicated API can have a very high learning overhead. In contrast, our proposed scheme favours a simpler (although possibly less general) API, which allows users to swiftly write code fragments at a high abstraction level.

The Subsumption architecture [1] is based on decomposing intelligent behaviour into multiple modules, each with different goals. By organizing these modules into a prioritized and layered framework, robotic systems can continously adapt their behaviours by choosing the module that is most appropriate for its immediate neighbouring environment. This approach has commonalities with our dataflow architecture and does explitly deal with system state, although it does not relate actions to well-defined state values necessarily.

The Saphira control system [7] is a reactive robot architecture composed of integrated routines for sensor interpretation, map building and navigation. These routines all operate on a shared representation of the robot's environment based on occupancy grids. The emphasis on this geometric model of the world is shared by the control methodology presented by this paper.

Several robot development environments include graphical interfaces for programming execution and data flow. These interfaces are designed as learning tools for amateur robot enthusiasts, although they also nicely accomodate to prototyping and debugging purposes. For example, the

[1] "Separation of concerns is a well-established principle in Software Engineering. Nevertheless, the failure to separate concerns effectively has been identified as a continuing cause of the ongoing software crisis."[13]

Visual Programming Language (VPL) [10] is a component of the Microsoft RDS environment that represent variables, robot commands and other programming constructs as visual blocks in a graphical interface. By connecting different blocks together, programmers can quickly build and deploy applications on robots while minimizing the amount of typing required. Similarly, the Lego Mindstorms NXT-G software embodies these constructs as different graphical icons, and thus provides a simplified interface for building basic robot control applications. Our approach also aims to reduce programming overhead in rapid prototyping situations, although we favour a more expressive hybrid graphical/textual representation instead.

The Human-Robotics Interaction (HRI) literature features many different non-conventional interaction mechanisms. The RoboChat framework [2] maps robot actions and programming constructs onto different fiducial markers. By showing these visual symbols in a particular sequence, one can program robots to accomplish complex tasks without writing a single line of code. The simplicity of encapsulating both low- and high-level robot behaviours alike using fiducial markers inspired our proposed approach to adopt a simple scripting API.

Gesture-based interfaces have been developed using both implicit and explicit communication mechanisms. Several authors have considered specialized gestural behaviors [8] or strokes on a touch screen to control basic robot navigation. Skubic *et al.* have examined the combination of several types of human interface components, with special emphasis on speech, to express spatial relationships and spatial navigation tasks [12]. These natural interaction methods can be easily integrated into the presented path planning methodology to improve its usability.

III. Graphical State-Space Programming Paradigm

A fundamental aspect of our approach is the ability to program robot behaviour using a high-level scripting language that provides access to a wide range of functions. It is also important that this scripting system has a degree or *transparency*, to allow access to functions implemented by the lower-level API. Using this substrate, we propose to combine the existing three types of robot programming methods. In particular, we allow the user to define a path in 2-Dimensional (or higher dimensional) space by specifying a sequence of waypoints, akin to graphical control or teleoperation. The main novelty of our system is the ability to attach executable code fragments (i.e. textual programs) to regions of space or segments of the robot trajectory. The execution of each code fragment occurs only when the robot arrives at or enters the associated waypoint or region, although execution can continue (as separate threads) for an indeterminate interval. Thus, the relationship of these scripts is graphically represented by waypoints and their connections, similar to visual programming. This combination of spatial layout, preferred trajectory and code fragments is referred as a "State-Space Code Template."

By using a high-level language to specify code fragments, users can quickly attach action or behaviour routines while minimizing the amount of coding required and without obscuring the graphical interaction. The embedded code can access primitives to load (or compute) a new trajectory, and thus conditionally-dependent trajectory planning can be readily specified and visualized using our approach. Furthermore, by allowing code fragments to load new templates during execution, we can achieve a programming mechanism that allows code to be developed in layers (e.g. as contingency plans).

In principle, the regions associated with code fragments can be mutually exclusive or they can be overlapping, which can lead to execution time performance costs and difficulty in visualization. In practice, the limitations in human visualization suggest that these regions often remain fairly simple, thus the intersection checking algorithm has a low computational cost. Anecdotally, we attached code fragments mainly to points or extremely large regions in our experiments.

A State-Space Code Template or a collection of interrelated templates is built off-line by a robot programmer. These plans are then downloaded onto the vehicle for execution. Unlike conventional compilation-based programming schemes, our approach allows the library of templates to be expanded during execution. Therefore, this methodology is highly expandable and versatile.

IV. Implementation

We have implemented graphical state-space control on our family of marine vehicles, which we collectively named as Aqua. The Aqua class of vehicles is based on a hexapod body design, equipped with 3 cameras and inertial sensors, and uses six flippers to generate thrust underwater, as shown in Fig. 2. The vehicle has an aluminum body and is ballasted to be neutrally buoyant in the water. When equipped with suitable hybrid flippers, Aqua obtains limited amphibious capabilities and can both walk as well as swim. The vehicle weighs under 20 kg and operates using two general-purpose computers. Aqua is operated either via remote access over a high-bandwidth fiber-optic tether, using a gestural interface allowing a human operator to specify high-level commands, or in a completely autonomous visual servoing setting. In all of these modes, it is commonplace to conduct trials involving an ordered sequence of experiments. Some experiments need to be repeated based on a function of external observations, internal state parameters, and the current location of the vehicle.

A typical task in the marine biology context is to visit a sequence of waypoints and collect data (e.g. record pictures or videos) at interesting locations. In some situations, the observations made might alter the subsequent activities to be conducted or the path to be followed. The Aqua family of robots have been deployed in regions as far south as the equator (in the Caribbean Sea) and as far north as the high Arctic (79° 26" North latitude) to execute this type of data

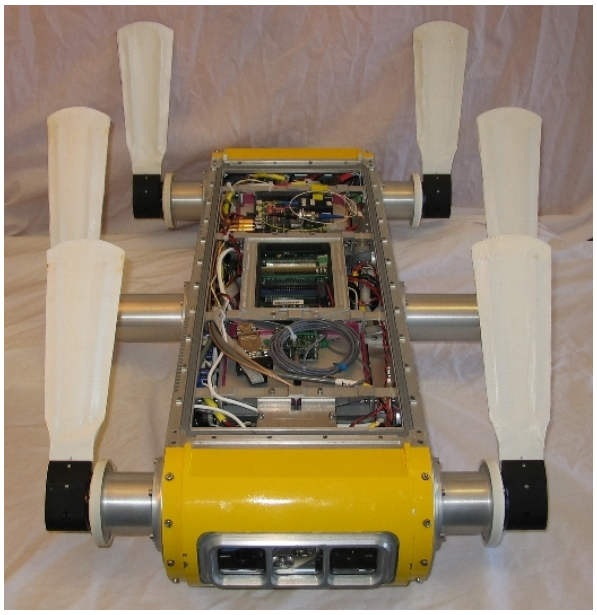

Fig. 2. The Aqua robot has 6 legs, 3 cameras (2 in front & 1 in back), and 2 on-board processors (stacked together in the center of the body).

collection tasks. For this piece of work we deployed the vehicle in a lake in Canada.

A. On-Board System Architecture

In previous control schemes, Aqua is programmed and controlled using a complex API that communicates between two Operating Systems (OS) running on two distinct processors on-board the robot. The *Control Stack* processor works with robot hardware drivers, amd is responsible for gait control and motion generation algorithms, which affect the synchronous motion model of the robot's legs. This low-level controller is written in C++ and is based on the RoboDevel [11] software suite. Due to strict real-time requirements, the control software operates on the QNX real-time operating system.

The second processor inside the Aqua robot focuses on processing sensory data, primarily originating from the on-board cameras, and hence is appropriately called the *Vision Stack*. The sensor-processing code is written in C++ and executes on top of a custom-tailored Linux OS. Since this software suite implements vision-guided motion and behaviour control of the robot, it inherently provides accessors via a private network connection (using the User Datagram Protocol, or UDP) to the low-level hardware controllers running on-board the *Control Stack*.

Although this control framework is highly optimized, expanding its functionality requires the use of a very complex API. To address this drawback, we developed an abstract API wrapper over the existing architecture. This *RoboControl Server* module was written using the Python scripting language, and provides a client-server infrastructure. This solution allows programmers to write arbitrarily-complex scripts that control the robot and receive feedback from it,

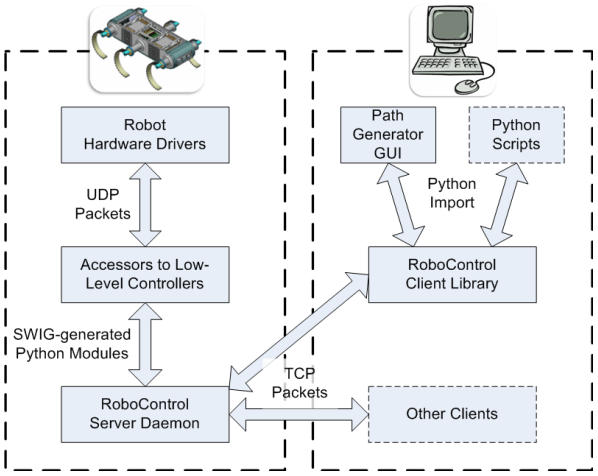

Fig. 3. Block diagram of current system architecture for the Aqua robot.

without needing to modify the existing control and vision code base. Fig. 3 depicts this hiearchy on-board the robot, as well as off-board abstraction layers implementing our proposed state-space programming paradigm.

The existing API within the Vision Stack was wrapped using an open-source C++-to-Python code generator called SWIG, which stands for Simplified Wrapper Interface Generator. We chose the Python scripting language because of its extended collection of standard libraries, its terse programming syntax, and its extended cross-platform compatibilities. To achieve (both hardware and software) platform independence, the *RoboControl Server* provides a network interface (via the Transmission Control Protocol, or TCP) to the wrapper modules. This is accomplished by translating ASCII text received on the listening port into appropriate function calls.

B. Off-Board Components

As mentioned previously, any application connected to the robot's network can control the Aqua robot by sending and receiving TCP string packets to and from the *RoboControl Server*. In particular, we developed a Python module called *RoboControl Client* which is used by our state-space contorl GUI. This library inverts the operation of the *RoboControl Server*, since it translates the network communication packets back into Python functions. The *RoboControl Client* provides an easy-to-use and platform-dependent interface, which for example allows users operating on Windows, OSX, or embedded platforms to control Aqua using a Python prompt.

We implemented the state-space programming paradigm as a Python-based GUI using the *RoboControl Client* library. This interface allows users to compose 3-Dimensional trajectories for the robot by specifying a sequence of waypoints, as shown in Fig. 4. The novelty of this GUI lies in its ability to associate Python scripts with each waypoint, as illustrated in Fig. 5, which will be executed when the robot passes through the corresponding locations. In addition, the user can also define a *global* script to be executed at every waypoint.

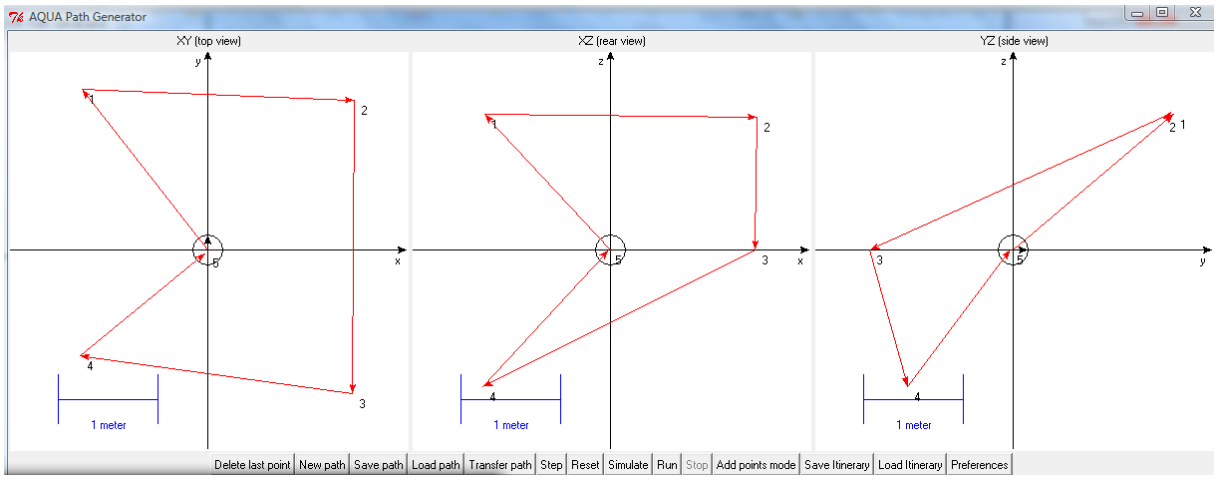

Fig. 4. A 3-D path generated using the state-space control GUI.

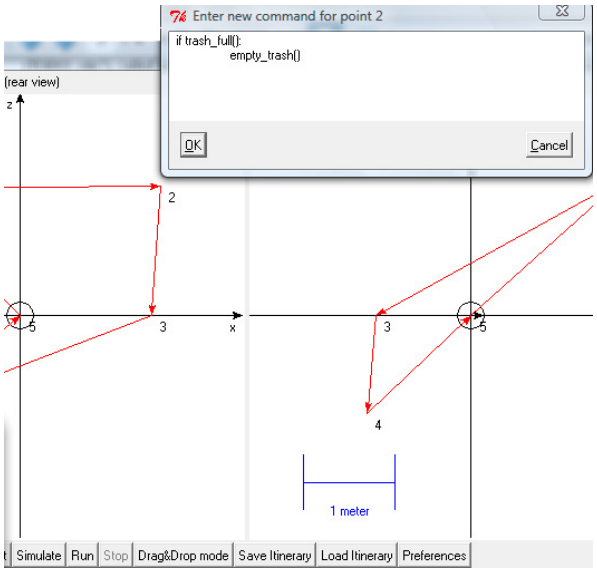

Fig. 5. A close-up of the GUI showing the attachment of a code block to waypoint 2.

When a path is executed using the real robot, a local planning algorithm continously regulates Aqua's motion to ensure that it follows the specified path. This planner can be tuned by adjusting its parameters such as the robot's speed and directional radii of curvature using the GUI, in order to reflect the robot's true behaviours as accurately as possible. The local path planner is complemented by software servos on-board the *Control Stack*, which compensate for unexpected forces applied onto the robot, such as underwater currents.

In addition to controlling real-life robots, the GUI also includes a stand-alone simulator. In this mode, the robot's position is estimated using a predefined motion model and is updated in real-time. Any other function calls or status requests using the *RoboControl* API is automatically deferred to simulator components, which are either specified by the user or as default stubs. This simulation mode provides a powerful tool for debugging complex paths before they are executed on the real robot.

V. Empirical Validation

We conducted a preliminary user performance study within a controlled and simulated environment to evaluate the usability of our proposed state-space programming paradigm. In particular, we compared the elapsed time required to specify a path using our GUI and to write up a conventional script. The primary emphasis is focused on allocating tasks to locations, and while navigation itself had to be specified the motion model was simplified to make this aspect as simple as possible.

We also conducted field tests with our amphibious robot using the graphical state-space interface. In particular, we used our GUI to program the robot to accomplish various tasks while mosing along a specified underwater trajectories. We validated these trials using qualitative and subjective performance measures.

A. Study Setup

In the controlled study, participants were asked to schedule a path for a maintenance robot through an amusement park. The layout of this ficticious environment is represented using a grid-based map with eight specified target waypoints, denoted by one or more icons indicating the required tasks at each waypoint, as illustrated in Fig. 6. We used a selection of five different tasks to diversify the process and to prevent users from memorizing paths prior to each recorded attempt. To perform a task at a specified location the user must invoke a function call and attach it to the appropriate location; for example, the *empty_trash()* function was to be called whenever the robot moved over a garbage can icon. After planning a path through all locations in any desired order, the robot had to return to its starting position.

When using our state-space programming GUI, subjects assigned movement and action tasks respectively by clicking on the top-down view and by writing function calls in

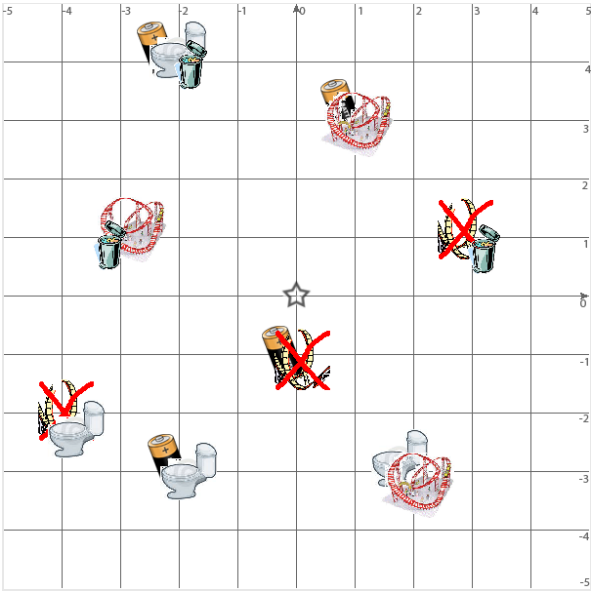

Fig. 6. Map of the environment, provided to participants of the user study.

the appropriate pop-up dialogs. Contrarily, the conventional scripting approach required users to make a single sequence of function calls in plain text, each corresponding to either addressing tasks, moving the robot forward by a particular distance, or rotating in-place for 90° in either direction. This simplified movement model was chosen to reduce users' cognitive load during the experiment.

After a brief tutorial on how to use our graphical interface, each subject was asked to program paths twice using the GUI and twice by writing conventional scripts in an alternating fashion. We used the elapsed time for each attempt as the evaluation metric for our study. On the other hand, although we observed all errors made during the sessions, we chose not to use it as a quantitative metric.

B. Study Results

Five participants were recruited for this study, including one female graduate student, three male graduate students and a male adjunct professor, all belonging to McGill University's School of Computer Science. All subjects have varying degrees of prior programming knowledge, and although nobody had used our GUI beforehand, some had experience using similar graphical robot interfaces.

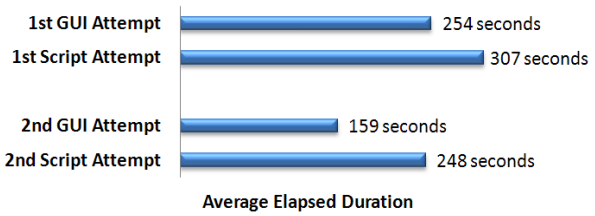

Fig. 7. Average elapsed durations for programming a path using our state-space control GUI and by writing a conventional script. Smaller values are preferred.

The results illustrated in Fig. 7 indicate that using the GUI lead to faster completion times on average compared to writing code. This performance gap is especially remarkable for the first GUI attempt, since all participants were learning how to use the interface and thus took more time to complete the task. The reduced performance for the scripting method can be attributed both to users having to plan movements in the robots local coordinate frame, and by the fact that users must keep track of the robots current location and orientation.

We also observed that the improvement in speed for the two GUI attempts is more prominent than for the scripting approach. Since all participants had prior programming experience, the small performance increase in the second scripting attempt was most likely the result of familiarity with the path due to repetition. In contrast, subjects were able to achieve up to a two-fold performance increase in their respective GUI attempts, suggesting that our graphical interface is a natural and intuitive way for programming these types of tasks.

All participants were able to correctly program paths using the GUI while making a negligible amount of typographical errors. On the other hand, multiple users incorrectly moved the robot through the environment after forgetting its previous orientation. Intuitively, it is simpler to provide locations to a path planning algorithm (as in our graphical state-space programming paradigm) rather than manually making low-level function calls to move the robot.

Interestingly, some participants used copy-paste, find-replace and other editing features to minimize the amount of typing required for writing conventional scripts. We are presently investigating ways of incorporating these and other programming practices to make our graphical interface more efficient and programmer-friendly.

Although this human interaction study has clearly illustrated the advantages of using our proposed state-space programming paradigm, we require more elaborate and controlled studies to better assess the quantitative benefits of this new approach and to ameliorate our graphical interface.

C. Robot Field Trial

To validate the usage of our implementation in the field, we deployed the Aqua robot in a large open-water lake. Although the results of this trial can only be reported in a qualitiative form, it nevertheless allowed us to assess and verify the utility of our approach within a field deployment context. We validated both the on-board robot control using our Python API, and the use of location-specific code execution (including state-conditional execution branch and on-the-fly path loading). Thus, the latter session demonstrated the ability to re-program the robot's path and operating parameters in the field using our graphical interface.

VI. Conclusion

This paper described a general approach and a corresponding software tool for the graphical state-space programming of underwater or otherwise mobile robotic vehicles. The control methodology is based on the ability to associate high-level executable code to locations in physical space or in

state-space. This method makes the execution of condition-dependent tasks more intuitive and more natural compared to existing programming interfaces. Moreover, by explicitly visualizing connections between states and actions, our environment enhances the design and debugging processes not only in terms of speed, but also in terms of reliability and comprehensibility. These benefits are consistent with observations made regarding the utility of graphical visualization tools for generic software engineering tasks [9]. Additionally, the visualization of both trajectories and position-dependent tasks assists with the standard software engineering objective of separation of concerns [13].

We have incorporated our graphical state-space control method into the software architecture of our amphibious vehicle. Our graphical interface provides not only a visualization of position-dependent activities, but also encompasses a high-level interface for many robot functions. Based on informal experience collected during field trials as well as a controlled user study, we observe that this interface substantially surpasses the performance of existing programming and control methods in terms of robustness, simplicity, and speed. Moreover, it is anecdotally observed to be a natural and pleasing development environment for creating multi-step plans (with built-in contingency) to accomplish complex tasks.

The software tool and approach we have developed can also be readily extended to include other state variables beyond spatial coordinates, although these are generally domain-specific. In the case of our underwater vehicle, useful dimensions include battery voltage and the average level of local illumination. Actions dependent on the values of these parameters can either be programmed as waypoints (if these dimensions are visualized) or as a plain conditional statement in the globally-executed script. Performing more general state-based programming using a mixture of graphical and textual styles remains a fertile topic for further examination.

We are currently investigating ways of improving the usability of our GUI, for example by facilitating common programming practices such as macros, by overlaying the 2-D planer maps with snapshots of the robot's environment, and by allowing users to define non-linear paths from one waypoint to another. Additionally, we are developing a more powerful local planner with collision-avoidance and other built-in reactive behaviours, so that users can concentrate solely on high-level interactions. Finally, we plan to extend our programming paradigm to other natural user interfaces, for example using an augmented reality setup.

VII. ACKNOWLEDGMENTS

The authors kindly acknowledge the contribution of the participants in our human interaction study. We also gratefully appreciate the financial support of the National Science and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[2] Gregory Dudek, Junaed Sattar, and Anqi Xu. A visual language for robot control and programming: A human-interface study. In *IEEE International Conference on Robotics and Automation (ICRA07)*, pages 2507–2513, Rome, Lazio, Italy, April 2007.

[3] Willow Garage. Ros. http://www.ros.org. Accessed: 09/15/09.

[4] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. *11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[5] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[6] Kyle Johns and Trevor Taylor. *Professional Microsoft Robotics Developer Studio*. Wrox Programmer to Programmer. Wrox, 2008.

[7] Kurt Konolige, Karen Myers, Enrique Ruspini, and Alessandro Saffiotti. The saphira architecture: A design for autonomy. *Experimental and Theoretical Artificial Intelligence*, 9:215–235, 1997.

[8] D. Kortenkamp, E. Huber, and P. Bonasso. Recognizing and interpreting gestures on a mobile robot. In *13th National Conference on Artifical Intelligence*, 1996.

[9] Peng Li and Eric Wohlstadter. View-based maintenance of graphical user interfaces. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 156–167, New York, NY, USA, 2008. ACM.

[10] Microsoft Robotics. Vpl introduction. http://msdn.microsoft.com/en-ca/library/bb483088.aspx. Accessed: 09/15/09.

[11] Uluc Saranli and Eric Kavins. Object oriented state machines. *Embedded Systems Programming*, May 2002.

[12] M. Skubic, D. Perzanowski, S. Blisard, A. Schultz, W. Adams, M. Bugajska, and D. Brock. Spatial language for human-robot dialogs. *IEEE Transactions on Systems, Man and Cybernetics, Part C*, 34(2):154–167, May 2004.

[13] Stanley M. Sutton, Jr. and Isabelle Rouvellou. Modeling of software concerns in cosmos. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 127–133, New York, NY, USA, 2002. ACM.