Graphical State Space Programming: A Visual Programming Paradigm for Robot Task Specification

Jimmy Li and Anqi Xu and Gregory Dudek

***Abstract*— We describe a framework that combines a software development paradigm, a software visualization technique, and a tool for robot programming. This infrastructure is called "Graphical State Space Programming" (GSSP), and allows robot application programs to be decomposed and visualized within state-dependent views. Our approach simplifies and expedites the programming process for robot routines and behaviors, and we examine the performance improvement that ensues through a set of controlled user studies. The usability and effectiveness of GSSP are also illustrated using a field demonstration with an aerial robotic vehicle.**

I. INTRODUCTION

This paper presents a software framework that facilitates the programming of typical execution plans for mobile robots by making geometric primitives explicit while using an aspect-oriented representation and visualization of the codebase. Our tool incorporates both a textual programming interface and a graphical flow-chart editor, allowing the user to specify robot routines using a combination of the *procedural* programming style (i.e. "What sequence of actions should the robot use to accomplish a specific task?") and the *declarative* programming style (i.e. "How should the robot react to a specific event or an anomaly?"). We explore various aspects and issues related to this programming methodology, some of which have been investigated previously [1]. In addition, we analyze this framework using a concrete implementation, multiple field tests, and a user study.

The primary goal of our framework is to allow programmers to efficiently generate plans that address a sequence of tasks while monitoring various constraints on the state of the robot and of the environment. These robot plans often encompass a desired path through the environment, a set of activities to be carried out at specific locations or within certain regions, and various other behaviors and failsafe mechanisms to deal with unexpected events. We model all of these three components as routines that are triggered when certain conditions are satisfied.

As a concrete example that illustrates different components in a plan, we often want our unmanned aerial vehicle (UAV), seen in Fig. 1, to take pictures of specific events and locations while travelling through a particular trajectory. We also want to ensure that the vehicle remains within an operational range and want it to constantly monitor factors that might affect its normal operation, such as low battery levels and strong wind conditions. For each of these exception states, the robot should either attempt to resolve the anomaly, or it should return back to home base immediately. All of these regular activities and failsafe mechanisms can be implemented as conditionally-triggered routines.

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

The authors gratefully appreciate the financial support of the National Science and Engineering Research Council (NSERC) of Canada.

Fig. 1. Our robotic platform is a commercial fixed-wing UAV with an on-board autopilot microprocessor and a gimbal-mounted camera.

Common robot programming interfaces include:

- a text-based programming environment for writing code in a standard computer language
- a graphical editor for specifying the flow of execution and the transfer of data
- a graphical user interface (GUI) for depicting the desired motion path and for issuing location-specific commands over a map of the environment

When using either of the first two types of interfaces, we can employ conditional constructs, such as *if-else* statements or graphical flow branch elements, to trigger individual robot routines. We can also define an order of execution by concatenating multiple conditional triggers together. Unfortunately, it is often difficult to embed failsafe behaviors into the normal ordering of activities, since these reactive mechanisms can be triggered at any time during execution. Brute-force attempts to do so may result in hard-to-find logic flaws in the execution flow.

In contrast, graphical control environments display the motion path and the sequence of location-specific actions over a map of the environment, which allows operators to easily identify and fix errors in the ordering of activities. Most of these systems however lack generic programming mechanisms for specifying branching paths and failsafe behaviors, and therefore can be used to implement only a limited subclass of robot plans.

Because the drawbacks of these programming environments are complementary to each other, we propose to merge their programming styles together into our Graphical State Space Programming (GSSP) paradigm. Through a GUI, GSSP allows users to define waypoints on a map of the world and to indicate regional constraints (i.e. *regions*) in the state space. Separately, programmers can write procedural code using a standard computer language to implement individual robot routines and behaviors. After attaching a block of code either to a waypoint or to a region, GSSP can then use this geometric information to determine when the code should be executed. Thus, GSSP preserves the expressiveness of textual programming and handles geometric constraints explicitly. In addition, since so many robot behaviors are tied to specific states or locations, GSSP provides a useful graphical visualization of the constraint space.

GSSP also integrates implementations of core functional modules (i.e. sensor processing, pose estimation, and path planning algorithms) in its back end. This prevents users from re-implementing the same basic functionalities in every plan. In addition, users can configure GSSP to incorporate specific motion controllers (e.g. [2], [3]), which would then systematically change the robot's behavior across all plans. By using this abstraction layer for plans and by decoupling geometric constraints (e.g. waypoints and regions) from robot routines, GSSP employs an aspect-oriented approach for specifying plans, which increases modularity by allowing the separation of cross-cutting concerns [4]. This in turn has been shown to directly result in a more understandable and maintainable codebase [5].

In prior work we showed that the visualization of waypoint-based programming improves productivity [1]. In this paper, we extend that work substantially by developing a richer vocabulary for geometric constraints while retaining simple visualizations, and describing a robust conflict resolution mechanism for managing concurrent robot behaviors. In addition, we examine the system's usability and performance through a set of user performance studies and through a large-scale field study.

II. GRAPHICAL STATE SPACE PROGRAMMING

The Graphical State Space Programming (GSSP) paradigm is used to specify execution plans for mobile robots. GSSP employs different programming approaches to address the two types of routines found in typical plans – a *procedural*-style ordering of location-specific tasks (e.g. "Go to this set of coordinates and take pictures of each location."), and a *declarative*-style grouping of reactive behaviors (e.g. "When your battery is low, return home immediately."). GSSP represents location-specific tasks as a sequence of waypoints, where each waypoint may be associated with a set of actions that are executed when the robot arrives at the corresponding location. In contrast, GSSP implements reactive behaviors as a set of actions that are triggered when the state (of the robot and of the environment) lies within certain regional constraints, which we refer to as *regions*. On a basic level, waypoints and regions are both geometric analogs to Boolean combinations of inequalities found within conditional (i.e. *if-else*) clauses. In reality however, since the robot might need to carry out waypoint-based routines and region-based behaviors concurrently, GSSP supplements these conditions with additional criteria to establish a priority scheme for handling concurrent execution.

GSSP generates execution plans using a state space representation, which decouples the codebase for plans from low-level, robot-specific hardware instructions. The *state space* is the space spanning all parameters relevant to the robot's programming, which are commonly called *state variables*. State variables comprise of both parameters for the robot, such as its horizontal position and its battery level, and parameters of the environment, such as the temperature and the time of day. State space programming affects the robot's behavior by issuing a *change of state* to the robot controller, which maps this request into input parameters for its specific hardware *resources*. For example, to move a wheeled vehicle, GSSP first computes the vector between the robot's current position and the desired coordinates. When the robot controller receives this *change of state*, it then carries out the request by adjusting the wheel velocities to steer the vehicle towards the destination. This state space representation allows the same plans to be executed on-board different hardware platforms, and is not coupled to motion planners or controllers.

A. Waypoint-Based Programming

A common structure found in the majority of plans for mobile robots is a sequential execution flow alternating between where the vehicle should move to next and which actions to take at that destination. Although programmers can readily implement this sequence of location-specific actions by writing procedural code, GSSP provides the necessary infrastructure both to minimize the amount of programming required from the user, and to decouple common control logic from each plan's codebase.

GSSP allows the user to visually drop and connect sets of waypoints on a map of the robot's state space, as shown in Fig. 2. Waypoints can be specified only for *writable* state variables, which are parameters that the robot's actuators can affect. A waypoint for one or two state variables can be visualized as a point on either a 1-D grid bar or a 2-D map. One of the most important maps for terrestrial and aerial vehicles is the top-down view of the environment, which depicts the x and y state variables. Another useful map that can be displayed in GSSP is the combined configuration space of all the joints of a robotic arm.

GSSP can couple different state variables together, which enforces waypoints to be specified only within their joint space. For example, it is often desirable to couple the x and y state variables for outdoor wheeled robots to prevent the user from creating a waypoint in y without providing the accompanying x values. On the other hand, this state coupling might not be as useful for wall-climbing robots.

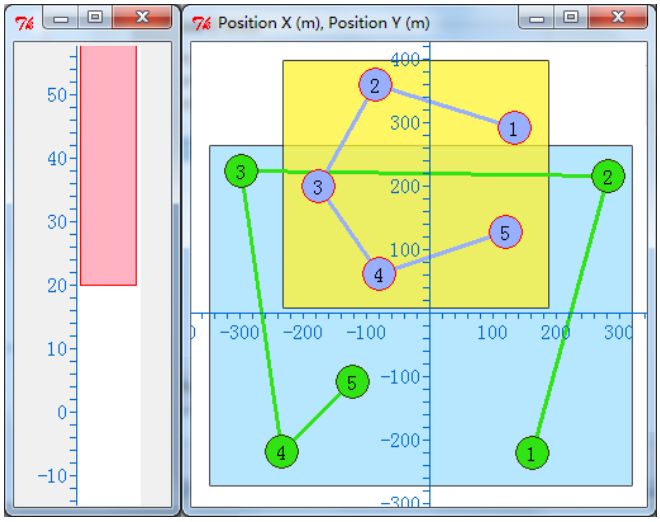


Fig. 2. Our GSSP implementation can visualize a state variable as a 1-D grid bar, and a pair of state variables as a 2-D grid map. Although regions can be defined on any state variables, waypoints can be specified only for *writable* state variables, i.e. those that the robot's actuators can affect.

Once a set of waypoints is created, the user can attach procedural code, written in a standard computer language, to any individual waypoint. Each block of code indicates the actions that the robot should execute when it arrives at the associated location. The user can explicitly initiate a waypoint set by writing a special function call and attaching it to either a region or waypoint. For example, to merge two separate paths, the user can trigger one waypoint set through a function call attached to a waypoint in the other set.

B. Region-Based Programming

Although the execution flow is relatively simple when programming waypoint-based actions procedurally, the logic can become drastically more complex when declarative-style reactive behaviors are added to the robot's programming. Building reactive routines to cope with unexpected events and state anomalies is an inherently declarative task – these behaviors describe *what* the robot should do when something does not go according to the original plan, but not *how* these actions should be integrated into the program ordering.

1) Triggering Regions: GSSP allows the user to define triggering conditions for reactive robot behaviors by drawing rectangular regions in the state space, as illustrated by Fig. 2. A region depicts a set of constraints applied to one or more state variables. A constraint can either take the form as an equality (e.g. $x = 10\,\text{m}$), a one-sided inequality (e.g. $battery < 25\,\%$), or, most frequently, a two-sided inequality (e.g. $0\,\text{m} < y \leq 50\,\text{m}$). Unlike waypoints, which can be specified for writable state variables only, the user can define regional constraints on any variables or parameters. As a concrete example, the user can prevent an aerial vehicle from losing stability in a windy environment by writing code to reduce its altitude and attaching that code to a triggering region of the form $windSpeed > 20\,\text{m/s}$.

Since high-dimensional regions only depict *intersections* of constraints, GSSP complements this by allowing the user to combine multiple constraints using arbitrary Boolean logic. A *Boolean region* is defined as the composite of multiple regions via their intersection (i.e. *AND*), their union (i.e. *OR*), or their negation (i.e. *NOT*). Aside from being used to combine constraints on multiple parameters, Boolean regions can also depict complex geometric shapes. For example, if a terrestrial robot is on patrol duty around a rectangular building, the operator can construct a rectangular ring-shaped area containing the patrol route and instruct the robot to take pictures every so often when it is within this area. Although both (simple) regions and Boolean regions can be used to specify multi-dimensional constraints, the former interface provides a useful geometric visualization, whereas the latter is syntactically more expressive.

GSSP provides a unified scheme for managing region-based behaviors. In particular, when a triggering region is satisfied, GSSP will attempt to initialize an instance of the attached code block. This scheme also forbids having more than one instance of each code block at any time during execution. This simplifies the conceptual model of our system by avoiding concerns related to thread safety.

2) Properties of Regions: By default, GSSP will spawn a new instance of a code block *immediately* if its triggering region is satisfied and if there are no existing instances for this code block. The user can specify a repetition count to limit the total number of times that the code block can be instantiated, and the user can also provide a time interval to delay the execution timing between consecutive code instances.

When writing routines to address anomalous states and unexpected events, we often want to stop the recovery process when the state returns back to normal. For example, if a terrestrial robot moves out of an operational area, one way to bring it back is to define a negated region outside the operational range and attach code that triggers a waypoint located inside the range. This might lead to an unexpected result however, because once the waypoint is initiated, GSSP will ensure that the robot reaches the destination before resuming other activities. To address this issue, GSSP defines an optional *bounding region* property for each code block. During execution, when a bounding region ceases to be satisfied, GSSP will terminate that corresponding code instance.

3) Concurrent Behaviors: Arguably the biggest advantage of using region-based programming is the notion of concurrency – by defining multiple region-specific code blocks, the user can request the robot to perform different actions at the same time. For example, the user can instruct an UAV that is flying through a set of waypoints to take pictures repeatedly within a certain area, by inserting a corresponding region and then attaching the camera request to that region. In contrast, to program the same task by writing procedural code, the user must either implement a separate picture-taking thread or manually issue the same camera command at multiple positions in the execution flow. This simple yet typical example illustrates that declarative programming can be conveniently used to add behaviors and functionalities to existing execution plans.

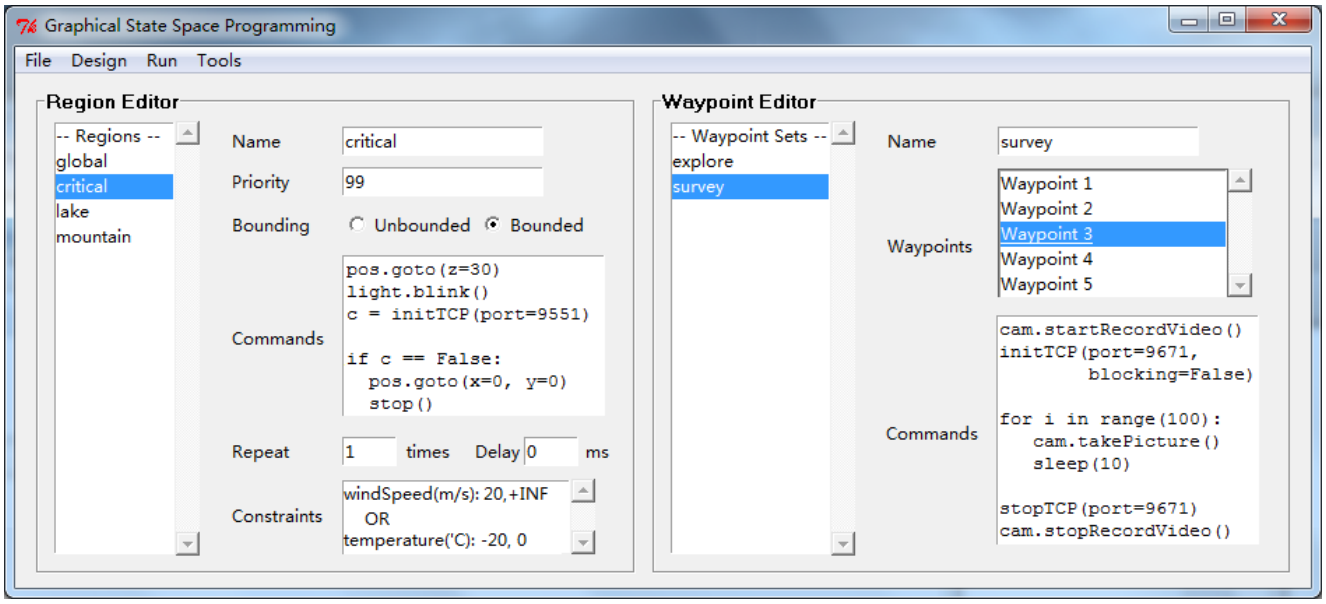


Fig. 3. The main GUI window of our GSSP implementation is used to attach blocks of code, written in a standard computer language, either to a regional constraint on the state space, or to an entry within a set of waypoints. The user can also configure the priority value and other parameters of each region. Please refer to our video accompaniment for a demonstration of our GUI.

A critical issue in concurrent programming is the presence of resource-related conflicts. For example, the user can define two overlapping regions that request the robot to move to different locations. This creates a conflict in the execution because both code instances are attempting to control the same *resource*. This type of multi-process resource management is a well studied concept in operating systems design [6], and GSSP resolves these conflicts using one of the most popular schemes available. First, each triggering region (and thus each code block) is assigned a priority value. Whenever a code instance requests a state change on an available resource, GSSP allocates that resource to the code instance until the state change has been fulfilled. Now consider the case of a conflict by assuming that code instance A currently owns a resource R and code instance B requests a state change on the same resource R. If B has equal or higher priority than A, then GSSP will pause A and grant R to B. Otherwise, if B has lower priority than A, then GSSP will pause B. In both cases, the blocked code instance will be resumed as soon as the resource R becomes available.

This simple yet powerful conflict resolution scheme is well suited for our robot application programming domain. For example, given two sets of waypoints corresponding to a global path through the environment and a local path used to explore a certain region, the user can combine these two trajectories by attaching code that initiates the local path to one of the waypoints in the global path. Since the local waypoint set is triggered by the global waypoint set, the former inherits the same priority value as the latter. When the local waypoint set is initiated during execution, GSSP will pause the code instance running the global waypoint set, let the robot explore the region of interest by following the local trajectory, and then resume the global path.

GSSP allows the user to establish an order of importance among different reactive robot behaviors by changing the priority rating for each corresponding triggering region. This priority rating does not have an effect on concurrent execution unless multiple code blocks with different priority values request the same resource. In this case, our conflict resolution scheme correctly allocates the resource to the code instance with the higher priority and pauses all other conflicting code instances with lower priority. Thus, our priority scheme can be used to ensure that anomalous states are addressed in order of urgency, and that normal activities are resumed once the anomalies have been resolved.

As a concrete example, the user can instruct a robot that is low on battery to return back to home base, by creating a high-priority region of the form $battery < 10\,\%$ and attaching code to initiate a single waypoint located at home base. Although GSSP will carry out this failsafe behavior properly, it will not prevent other concurrent code blocks (as in the picture-taking example) from further depleting the battery. To resolve this issue, the user can make a special function call in the code block that will disable and prevent all code instances with lower priority from executing. Although this feature should not be used carelessly, it can assist in the successful and timely recovery from a dire emergency by disabling non-essential robot behaviors.

III. IMPLEMENTATION

We implemented a graphical editor (as seen in Fig. 3) and the back end infrastructure for our GSSP paradigm. We used the Python computer language both for our implementation as well as for writing procedural code in GSSP. Python is well suited for our application because of its terse syntax, because of its versatile platform compatibility, and because it is supported by many popular robot middleware packages. Furthermore, since the Python interpreter only evaluates code during runtime, this allows the user to modify the plan during execution.

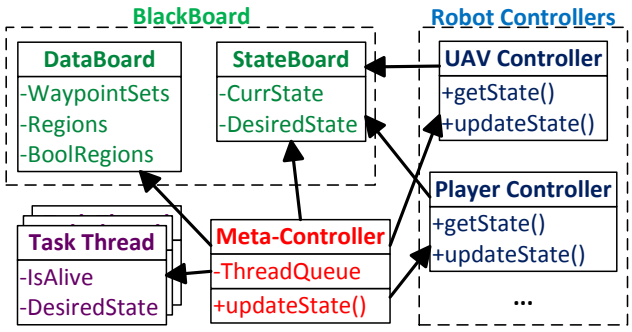


Fig. 4. Condensed class diagram of our GSSP implementation.

As illustrated by the class diagram in Fig. 4, our system uses a blackboard model [7] for storing and accessing both the data in plans as well as the current state of the world. The meta-controller is responsible for continuously ensuring the proper execution of the plan, which includes evaluating regions, managing code block instances (i.e. as task threads), and resolving resource conflicts. Since code blocks can trigger waypoint sets, which can then trigger more code blocks, new task threads inherit the priority value of their parent thread. After collecting and processing state change requests on different resources from all executing threads, the meta-controller then issues a combined state change to the robot-specific controller, which then addresses this request using low-level hardware commands.

A. Waypoints and Regions

We defined waypoints and regions using exact numeric inequalities rather than using fuzzy logic to establish an intuitive conceptual model. To prevent numerical instabilities however, we allow the user to specify a proximity distance for waypoints. Thus, waypoint-based code blocks are equivalent to region-based code blocks with small spherical regions.

In our implementation, a code block's bounding region can only be either identical to its triggering region or disabled. In the former situation, GSSP will terminate a running code instance when its associated region fails to be satisfied. Although this approach is somewhat restricted, it prevents scenarios where the triggering region is satisfied but the bounding region is not satisfied, which could result in an infinite loop of initialization followed by immediate termination of the code block instance.

GSSP automatically attaches a *global region* (i.e. a region without any constraints) to every plan, in which the user can define setup code. By adjusting its frequency and priority parameters, additional global regions can also be used to specify idle or default behaviors when no other code instances are running. For example, by attaching code to a low priority global region that triggers a waypoint located at home base, the user can ensure that the robot will always return home after it has completed all other activities.

B. Application Programming Interface

In addition to writing generic Python statements in code blocks, users can employ special variables and functions defined by GSSP to query state variables, and to create or modify waypoint coordinates and region parameters. In addition, the user can issue a state change to any writable state variable. Although by default user-defined variables are limited by scope within their individual code blocks, these variables can be explicitly promoted into a shared or global namespace.

Because GSSP provides a robot-independent programming environment, it often cannot replicate all the available functionalities of the target robotic platform. We address this deficiency by enabling *transparency*, meaning that we give the user access to the robot-specific API when writing code in GSSP. Unfortunately, because some APIs are available only in a compiled format, our GSSP system does not make any assumptions on how these low-level functions are implemented. As a result, GSSP does not provide resource management, conflict resolution, or other features for these low-level function calls. Without careful planning, using transparency carelessly might result in unexpected execution. For example, if the user adjusts the wheel velocities through a low-level function call, this might either deviate the robot away from the trajectory specified within GSSP, or this command might be instantly overwritten by a requested state change. Transparency can greatly improve the expressiveness of robot plans, but the user must be responsible for ensuring that these low-level function calls do not conflict or negatively affect the meta-controller's operations.

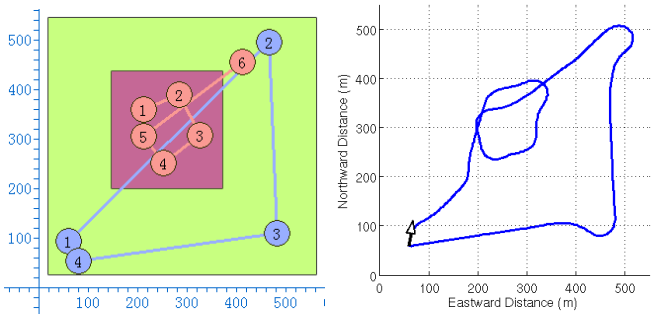
Our implementation includes a remote interface that allows external applications to interact with the meta-controller. In particular, client instances can request values and issue state changes by sending text messages through a network socket. This allows programmers to write interfaces using any programming language that supports network sockets, and connect their applications to our GSSP framework. Using a GSSP code block, the user can launch these clients either synchronously (i.e. the meta-controller will halt execution until the client sends a terminal command) or asynchronously (i.e. the meta-controller will continue executing code instances, and has the ability to pause and forcefully terminate the client instance). Using this remote interface, we were able to easily write a client that can steer the robot using the accelerometer of a smart-phone device.

IV. EMPIRICAL VALIDATION

We integrated our GSSP system with the control software for an unmanned aerial vehicle (UAV). We conducted field trials to assess the usability and efficiency of our graphical programming paradigm within a field deployment setting. We also conducted an elaborate user study in a controlled environment to quantitatively measure the performance difference between GSSP and procedural text-based programming.

A. Field Trial

Our UAV, shown in Fig. 1, is a rigid body fixed-wing aircraft commercially available from Procerus® Technologies. Its 1 meter wingspan is built using expanded polypropylene foam, which is useful for absorbing the collision impact



(a) (b)

Fig. 5. (a) An execution plan that evaluates the priority system, and (b) the resulting flight trajectory for an UAV. The triangle-shaped waypoint set is attached to the larger region (with high priority) whereas the circle-shaped set is connected to the smaller region (with low priority).

during touchdown. A brushless electric motor powered by lithium polymer batteries can drive the plane at average ground speeds of 14 m/s and for durations of up to 30 minutes. An on-board autopilot micro-processor receives instructions from the ground control software wirelessly. The autopilot is connected to various sensors, including an accelerometer, a gyroscope, a pitot tube (i.e. pressure and wind speed sensor), and a GPS device. The UAV can operate in many modes ranging from joystick-based manual control to fully autonomous waypoint-based navigation.

We ran our Python GSSP software within a Linux environment on-board a 1.66 GHz dual core notebook computer. Our software communicates with the UAV through the proprietary ground control software.

Fig. 5(a) depicts a large and low priority region attached to a triangle-shaped waypoint set, and a smaller region with higher priority attached to a circle-shaped waypoint set. The recorded flight path in Fig. 5(b) shows that the UAV started travelling through the triangular waypoint set, but then immediately switched to the circular trajectory upon entering the high priority region. After successfully completing the circular trajectory, the vehicle resumed its previous course, and finally completed the triangular waypoint set.

We also executed another plan that allowed the human operator to control the plane using arrow keys through the remote interface. To ensure that the robot does not fly too far away from home base however, our plan contained a high priority negative region that restricted the operational range by forcing it to go towards home back. During flight, as soon as the operator steered the vehicle outside of the operational bounds, GSSP disabled the remote interface and turned the robot around towards home base. The user regained control of the plane when it returned within the specified boundaries. In all of our executed plans including the aforementioned two, we instructed the UAV to continuously take pictures using its on-board camera while carrying out other activities.

Our field tests confirmed the successful operation of all the major aspects of GSSP, including region priority, resource-related conflict resolution, concurrent behaviors, and external control using the remote API. We were also able to demonstrate that GSSP can be used to quickly modify plans and generate new plans during field trials.

B. User Study Setup

We conducted a controlled user study to evaluate and compare the performance between GSSP and text-based procedural programming when specifying typical robot plans comprising of both location-specific tasks and reactive failsafe behaviors. In a controlled environment, we asked the participants to implement five robot tasks first using GSSP, and then by writing pure Python code. These tasks consist of following a set of waypoints, carrying out several location-specific actions, and three failsafe behaviors in response to different anomalies. We then asked the participants to implement a different plan with a similar set of five other tasks, this time only using GSSP.

Prior to the programming sessions, we provided a 15 min training session in which we explained the concepts behind GSSP and demonstrated our GUI. We also provided a condensed Python documentation of a robot library, as well as sample code outlining basic Python structures and syntax.

We recruited 16 male and female participants ranging from 19 to 52 years old for this user study, each with varying degrees of prior robot control and general programming knowledge. We categorize the participants as follows based on their level of prior experience with robot programming:

- *Regular Users*: 11 individuals without prior experience with programming robots, and with minimal formal training in writing code in general;
- *Expert Users*: 4 individuals with prior experience programming robots by writing text-based procedural code;
- *Developer*: the developer of our GSSP implementation.

As in other user studies, results from the "developer" category of users do not exemplify typical performance, but rather serve as a baseline used to compare with other results.

To quantitatively assess each participant's performance when using GSSP and when writing pure Python code, we employed the following three metrics:

- *Elapsed Time*: the time interval between when the user begins programming the plan, until when the user declares that all five tasks have been addressed;
- *Typographical Errors*: individual keywords, waypoints or regions that are specified incorrectly, but adjusting these elements would result in correct execution;
- *Task Errors*: after correcting all typographical errors, the number of tasks that were not implemented correctly.

We argue that the last metric provides an assessment on how well each user understands the concepts of GSSP and of textual robot programming.

C. User Study Results

We observe that all participants required noticeably more time (Fig. 6) and committed far more typos (Fig. 7(a)) when writing pure Python code compared to using GSSP. Arguably the dominant contributing factor to this performance gap is the fact that we provided the meta-controller infrastructure in GSSP, whereas users had to explicitly address geometric constraints, concurrent execution, and priority schemes during the text-based programming session. This resulted in

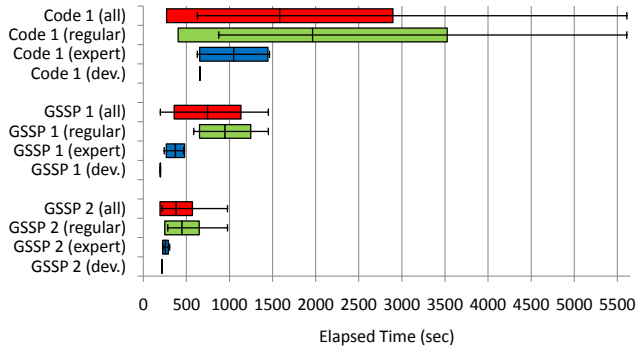
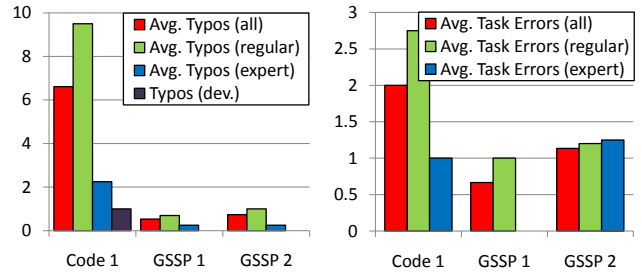


Fig. 6. Elapsed time for the textual programming session (Code 1) and for the two GSSP programming sessions (GSSP 1 and GSSP 2). Beams indicate $\pm 1\sigma$ from mean; error bars indicate min. and max. values.



(a) (b)

Fig. 7. Average number of typographical errors (a) and average number of task errors (b) present for the procedural code session (Code 1) and for the two GSSP plans (GSSP 1 and GSSP 2). Note: the developer did not make any task errors during all three sessions.

drastically different codebase sizes, which explains the gaps in elapsed time and in the number of typographical errors.

In addition to manually implementing features of the meta-controller, another challenge for users during the textual programming session is to embed the three reactive failsafe behaviors into the execution ordering. The participants used different approaches to emulate reactive behaviors by writing procedural code, some of which were more successful than others. As shown in Fig. 7(b), both the regular users and the expert users made significantly more task errors on average during their textual programming session.

Upon closer inspection of the elapsed time results, we observe that even the first GSSP session is characterized by a two-fold speed improvement over the textual programming session on average, whereas the second GSSP session further widens the gap with an average four-fold speed improvement over the text-based session. This is particularly striking since GSSP is a completely novel interaction paradigm for all users except the developer, while all users have some familiarity with the creation of textual descriptions of process flow (even if outside the robot programming context). These performance gaps are also present in the variance of the dataset, suggesting that not only did the participants individually exhibit comparable speed improvements between the three sessions, but the consistency of the elapsed time across participants are ordered with a similar proportionality.

One expert user achieved comparable results to that of the developer for all three metrics and during all three sessions. In addition, although the standard deviation beams in Fig. 6 indicate that the elapsed times for expert users varied greatly during the textual programming session and remarkably less so during the first GSSP session, all expert users completed the second GSSP session almost as fast as the developer. These results suggest that users with prior programming experience are able to quickly adapt to GSSP and make effective use of its various features, arguably because the concepts presented by GSSP are readily predictable by programmers. Thus, these results support our belief that GSSP offers an intuitive interface for programmers.

Regular users showed the most pronounced improvements between the textual programming and GSSP sessions. In fact, the span of elapsed times for expert users writing Python code is worse than the span of elapsed times for regular users during their first GSSP session. This illustrates that people with minimal programming experience can specify robot tasks using GSSP with comparable speeds to veteran robot programmers writing textual code, which suggests that the concepts behind GSSP are accessible by a broad audience.

Both the average number of typos and of task errors are noticeably smaller for the two GSSP sessions when compared to the text-based programming session, as shown in Fig. 7. On the other hand, the results indicate a slight increase in the average number of task errors when comparing the first GSSP session to the second one, for both types of users. Anecdotally, we observed that all participants were exhausted after the Python programming session, and thus carried out the second GSSP session in a hurried an annoyed mood, resulting in more errors. This suggests that textual programming interfaces can be more taxing psychologically than our GSSP system when programming robot plans.

Our experimental results indicate that GSSP is noticeably better suited for programming the class of robot actions and behaviors used in our tests, as compared to writing procedural code alone. The extent to which these results generalize to all robot programming, and to large-scale programming, is a question for future research.

V. CONCLUSION

We have described a software programming paradigm called Graphical State Space Programming (GSSP) to specify execution plans for mobile robots. Using our software implementation, we can specify an ensemble of location-specific and state-specific actions, an execution ordering through these actions, and reactive robot behaviors for addressing anomalies. While our examples are largely related to physical locations as the key states of interest, our framework can be used to describe general robot plans.

Using the GSSP methodology has several distinct advantages over conventional programming: it provides visualization for both state-dependent actions and constraint regions, it facilitates the arbitrary Boolean composition of multiple constraints, and it provides a mechanism for prioritization and concurrency control. Equally important from a software engineering standpoint, GSSP allows for a degree of "separation of concerns" by segregating different code blocks.

We evaluated the usability of the system using both a controlled user study, and a field trial with a robot aircraft. GSSP appears to greatly improve the efficiency of code development for the typical robot plans that we have examined. Despite these results, it remains to be seen how well GSSP will scale for large and complex robot plans.

In future work we are examining extensions to our conflict resolution mechanism that are more suited to the nature of robot programming. We are also evaluating alternative methods for specifying priority to can better cope with the scalability of robot plans. Finally, we plan to conduct more elaborate experiments to further characterize the quantitative nature of the performance gain of GSSP over various traditional robot programming approaches.

VI. RELATED WORK

Text-based systems are among the most popular methods for programming robot behaviors. Numerous research groups and corporations have developed libraries and middleware using standard computer languages to provide a common application programming interface (API) for different classes of robots [8], [9]. Prominent examples of robot abstraction systems using the procedural programming style include Player/Stage/Gazebo [10], the Robot Operating System (ROS) [11], and Microsoft's Robotics Developer Studio (RDS) [12]. Both GSSP and text-based programming environments provide abstraction layers for specifying general robot behaviors in a hardware-independent manner, although GSSP further increases the modularity of the codebase by explicitly decoupling geometric constraints.

Kanayama and Wu developed a text-based programming approach [13] that is designed for programming motion trajectories and location-specific tasks. Their Motion Description Language (MDL) establishes a standardized, procedural-style method for describing motion geometry. This text-based approach for programming motion paths nicely complements the visual interface for specifying waypoints in GSSP.

Another class of text-based methods (e.g. [14], [15], [16]) uses the declarative programming style to specify behaviors that the robot should exhibit in reaction to different states of the environment. Because the order of execution is not specified, these systems can program general behaviors such as wall following using minimal amount of code [8]. The region-based programming nature of GSSP is a graphical analog of these declarative text-based methods.

Various robot abstraction systems include graphical programming interfaces that visualize the execution and data flow. Systems such as Microsoft's Visual Programming Language (VPL) [17] and the Lego Mindstorm NXT-G software [18] visually represent variables, robot commands and other programming constructs as interconnected blocks to depict a flowchart of the codebase. GSSP similarly provides a visualization of the execution flow, although it also embeds geometric data by overlaying the information on top of a map of the environment.

Graphical interfaces for specifying waypoints and issuing commands can be found in a number of robot control systems (e.g. [19], [20]). Because most of these GUIs do not require the user to write code, they allow people without any programming experience to control robots. Unfortunately, this comes at the sacrifice of useful programming constructs such as conditional triggers and arbitrary path loops. GSSP addresses this deficiency by merging GUI-based control with text-based programming into a single comprehensive framework.

Fuzzy logic also uses state-space models to explicitly describe controller functionality. GSSP, however, places emphasis on the graphical depiction of state space constraints and on the separation of code (i.e. *separation of concerns*) to enhance programmer efficiency. In contrast, fuzzy logic systems focus on the use of continuous functions of the state to identify the level of activity of various control parameters.

REFERENCES

[1] J. Sattar, A. Xu, G. Dudek, and G. Charette, "Graphical state-space programmability as a natural interface for robotic control," in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA 2010)*, vol. 3, Anchorage, AK, USA, May 2010, pp. 4609–4614.

[2] P. Giguere and G. Dudek, "Clustering sensor data for terrain identification using a windowless algorithm," in *Proc. of Robotics: Science and Systems IV*, Zurich, Switzerland, June 2008.

[3] A. Xu and G. Dudek, "A vision-based boundary following framework for aerial vehicles," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2010)*, Taipei, Taiwan, October 2010.

[4] J. S. M. Sutton and I. Rouvellou, "Modeling of software concerns in cosmos," in *Proc. of the 1st Int. Conf. on Aspect-Oriented Software Development (AOSD '02)*. NY, USA: ACM, 2002, pp. 127–133.

[5] I. Jacobson and P.-W. Ng, *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley Professional, 2004.

[6] A. Silberschatz, G. Gagne, and P. B. Galvin, *Operating System Concepts, Seventh Edition*. John Wiley & Sons, 2005.

[7] S. C. Shapiro, Ed., *Encyclopedia of Artificial Intelligence, Second Edition*. New York, NY, USA: Wiley-Interscience, 1992, vol. 1.

[8] G. Biggs and B. MacDonald, "A survey of robot programming systems," in *Proc. of the Australasian Conference on Robotics and Automation (ACRA 2003)*, Brisbane, Australia, December 2003.

[9] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for robotics: A survey," in *Proc. of the 2008 IEEE Conference on Robotics, Automation and Mechatronics*, September 2008, pp. 736–742.

[10] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," *11th International Conference on Advanced Robotics*, pp. 317–323, 2003.

[11] "Willow Garage - ROS," http://www.ros.org. Accessed: 10/09/10.

[12] K. Johns and T. Taylor, *Professional Microsoft Robotics Developer Studio*, ser. Wrox Programmer to Programmer. Wrox, 2008.

[13] Y. Kanayama and C. Wu, "It's time to make mobile robots programmable," in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA 2000)*, 2000, pp. 329–334.

[14] R. A. Brooks, "Intelligence without representation," *Artificial Intelligence*, vol. 47, pp. 139–159, 1991.

[15] L. Huang and P. Hudak, "Dance: A declarative language for the control of humanoid robots," Yale University, Tech. Rep. YALEU/DCS/RR-1253, August 2003.

[16] G. Biggs and B. MacDonald, "Implementing a reactive semantics using openrtm-aist," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2005)*, October 2010, pp. 994–999.

[17] "Microsoft Robotics - VPL Intro." http://msdn.microsoft.com/en-ca/library/bb483088.aspx. Accessed: 10/09/10.

[18] J. F. Kelly, *LEGO MINDSTORMS NXT-G Programming Guide*. Apress, 2007.

[19] T. Kramer, R. Laird, M. Dinh, C. Barngrover, J. Cruickshanks, and G. Gilbreath, "Firre joint battlespace command and control system for manned and unmanned assets (jbc2s)," in *SPIE Unmanned Systems Techology VIII*, Orlando, FL, USA, April 2006.

[20] "Procerus Technologies - Ground Control Station - Virtual Cockpit 2.6 & Commbox 1.1," http://www.procerusuav.com/Downloads/DataSheets/Virtual_Cockpit_2.6.pdf. Accessed: 10/09/10.