A Control Software Architecture for Autonomous Unmanned Vehicles inspired in Generic Components

Alberto Ortiz, Francisco Bonnin-Pascual, Emilio Garcia-Fidalgo and Joan P. Beltran

***Abstract*— This paper describes GLOC3, a general software control architecture intended for unmanned vehicles. Applying a criterion of maximizing software maintainability and reusability, as well as fulfilling the monitoring and logging needs that arise from any experimental platform, this architecture is based on a single minimalist generic component. Besides, this software infrastructure is intended to make easier both the specification of experiments and missions at the different development stages. The case of the control architecture of an Unmanned Aerial Vehicle, to be employed as part of the robot fleet of the MINOAS project, is considered by way of illustration.**

***Index Terms*— Autonomous Unmanned Vehicle, Unmanned Vehicle System, Control Software Architecture**

I. INTRODUCTION

Unmanned Vehicle Systems (UVS), in its many different autonomous/untethered forms as *Unmanned Aerial Vehicles* (UAV), *Autonomous Underwater Vehicles* (AUV), *Autonomous Surface Vehicles* (ASV) or *Unmanned Ground Vehicles* (UGV), and also as the non-autonomous/tethered *Remotely Operated Vehicles* (ROV) or *Remotely Piloted Vehicles* (RPV), have become more and more prevalent over the last decade, being involved in an exponentially increasing number of either land, marine/submarine and aerial applications. Irrespective of the environment, UVS have been proposed to be used for a broad range of tasks including security, monitoring, inspection, and military operations (intelligence, reconaissance, surveillance and target acquisition), just to name but a few, and new uses are being identified every day. Just as examples of the impact UVS are expected to have in the near future, the AUV and ROV market reports by Douglas-Westwood forecast that 1,144 AUVs will be required over the next decade in the most likely scenario, with a total market value of $2.3 billion for AUVs and $3.2 billion for workclass ROVs [1], [2], while Teal Group's 2010 market study estimates that UAV spending will more than double over the next decade from current worldwide UAV expenditures of $4.9 billion annually to $11.5 billion, totaling just over $80 billion in the next ten years [3].

As the use of *Autonomous Unmanned Vehicle Systems* (AUVS) continues growing, as forecasted above, *Control Software Architectures* (CSA) will become an even more critical aspect within the development process of those robotic systems. In effect, the CSA constitutes the framework where the required functionalities are implemented, ranging from the lowest level tasks —e.g. a control law— to the highest level tasks —e.g. scene 3D reconstruction—, so that, using an analogy, the CSA can be regarded as the *backbone* of the robot. The election of a proper CSA is not as critical for validating isolated functionalities, but, when a complete system is concerned, the inherent complexity requires from a careful election of system components and how they are interfaced.

Moreover, since AUVS come in a wide variety of configurations differing in size, sensor suites, processing power and communication patterns, not to mention the differences in their locomotion capabilities, this diversity has naturally led to a great deal of one-of-a-kind control software, and, thus, most research and development for robotic platforms is based on proprietarily designed software architectures invented from scratch, where implementations are tied to specific robot hardware. However, underlying this variety there is a significant portion of robot functionality that is common to a large number of robotic systems and different application domains: UAV, AUV, ASV and UGV need all sensor sampling and pre-processing, sensor data processing and fusion —many times for very similar sensor suites across platforms—, self-localization, and perhaps map building, motion planning and control (including obstacle avoidance and path planning), etc., with similar solutions for platforms sharing the same number of degrees of freedom. Therein lies a large opportunity to reduce cost, complexity and risk. Furthermore, these architectures can now be implemented using standard compliant *Components Off-The-Shelf* (COTS) software to provide stable, mature solutions.

The choice of a flexible and easily extendable CSA, as well as code reuse and transfer between platforms, is crucial to avoid spending more time and effort than strictly necessary in every single project. Even during early stages of development, when experimentation takes most part of the time, a CSA that makes easier not only the incorporation of new functionalities at low cost, but also mission specification and monitoring, is undoubtedly of great value to shorten prototyping times and so meet the UVS market requirements. It becomes as well fruitful during platform exploitation —including for research use— when it is a goal to apply a criterion of greatest efficiency and, accordingly, widen the scope of executable missions for the same AUVS.

After the great difficulties we have encountered in the development, maintenance and upgrade of the control system of unmanned platforms, this paper presents GLOC3, *Generic*

This work is partially supported by projects SCP8-GA-2009-233715 (EU FP7 MINOAS) and AAEE 0108/09 (Conselleria d'Innovacio, Interior i Justicia, Govern Balear), and FEDER funding.

All authors are with the Department of Mathematics and Computer Science, University of Balearic Islands, Cra. Valldemossa, km 7.5, 07122 Palma de Mallorca, Spain; email: alberto.ortiz@uib.es, emilio.garcia@uib.es, xisco.bonnin@uib.es, joanpau.beltran@uib.cat

LOosely-Coupled Component-based Control software architecture, our latest effort on developing robotic software architectures. One of the primary goals while devising GLOC3 has been the design of a network-transparent CSA general enough to account for virtually any kind of autonomous vehicle and mission. Easy maintenance and software reusability have also been of utmost importance, and, to this end, GLOC3 is organized around a generic component fitted with usually needed functions —i.e. data exchange, error handling, status notifying, etc.— and a command interface through which the component receives orders, input data and provides results. In this way, only the strictly intended functionality for the module must be specified (i.e. estimate vehicle motion by means of the fusion of laser and vision data using a particle filter), and, once defined, it can be reutilized without restriction because of the loose coupling between components which the command interface provides.

The rest of the paper is organized as follows: Section II revises the concepts involved in CSA and existing approaches; Section III gives an overall view of GLOC3 and the main components that it defines; Sections IV and V describe the generic structure of GLOC3's components; Section VI presents the case of the control software for a UAV; finally, Section VII concludes the paper.

II. CONTROL SOFTWARE ARCHITECTURES FOR ROBOTS

By definition, an AUVS is both unmanned (i.e. there is no man in the loop to continuously guide the robot so that the system must make its own decisions) and untethered (i.e. there is no power supply nor communication towards the vehicle), or, in other words, an AUVS acquires the necessary information from the environment, using the adequate sensors, process it as necessary to decide how to act, and then executes the appropriate actions by means of the available effectors in order to achieve the set of goals corresponding to a specific mission. Although part of these functions can be solved by classical control theory (which, thanks to the solid mathematical foundation underneath, is able to provide most times optimum solutions), overall AUVS control needs to be addressed through complementary methods, such as try to mimic somehow human-like reasoning.

Getting inspiration from this general model, the spectrum of control methodologies that have been developed and applied to control real vehicles is ample. However, it is widely recognized that they can be generally grouped as: *deliberative/hierarchical*, which feature explicit reasoning/planning on symbolic representations as well as on accurate and complete world models; *reactive/behaviour-based*, characterized by a tight coupling of sensing to action through the so-called behaviours, sort of rules that almost directly map sensor data to actions; and *hybrid*, halfway between the deliberative and the reactive paradigms, exemplified by three-layered architectures mixing reactive and deliberative components, where the former handles low-level control issues requiring fast response time, while the latter is responsible for high-level issues on a longer time scale (see [4], [5], [6], [7], among others).

Although a control methodology is an essential piece of a complete robotic system, it is not enough with adopting one: it is still necessary to decide how the control software is going to be specified and implemented. It is true that many times the control methodology has, to a great extent, determined the software structure, or, in other words, the framework for defining the control software has been devised around a certain control methodology. Because of this, it is usual the confusion between the CSA and the control methodology, although, clearly, they are quite a different thing.

Although it is really difficult to evaluate quantitatively the design of a CSA, there are several acknowledged qualitative criteria describing how a well-developed architecture should be, namely predictability, reactivity, robustness, modularity, extendibility, generality and standardization [6], as well as reduced footprint and end-to-end turnaround time [8]. So far, a number of platforms for robot software development trying to satisfy some or all of these goals have been proposed. Among them, Player/Stage [9] and, very recently, ROS [10] have become very popular, although other relevant efforts in this line that are also worth mentioning are Orocos [11], CoRoBa [12], MCA [13], Miro/SORA [14], [15], Marie [16] and JAUS/OpenJAUS [17], [18]. Many of them additionally provide middleware functionality that simplifies software distribution over different physical agents, as well as libraries of components to simplify prototypes building.

Two recent European initiatives have tried to bring order to this existing software for robot programming: RoSta, a Coordination Action funded under the European Union's Sixth Framework Programme (FP6)[1], and its continuation as the FP7 BRICS (Best Practice in Robotics)[2] project, whose prime objective is to structure and formalize the robot development process itself and to provide tools, models, and functional libraries. The *Joint Architecture for Unmanned Systems* (JAUS) was a similar initiative formerly sponsored by the *Office of Naval Research* (ONR) of the United States Department of Defense, which finally migrated from the JAUS Working Group, composed of individuals from the government, industry and academia, to the *Society of Automotive Engineers* (SAE), Aerospace Division, Avionics Systems Division; the AS4, *Unmanned Systems Technical Committee*, now maintains and advances the set of standards.

Adopting the categorization of RoSta, which distinguishes between (1) *middleware and integration frameworks*, (2) *control architectures*, and (3) *development toolkits*, the work that is presented in this paper is intended to be independent of the middleware to be finally used, and therefore lies in-between the two first RoSta categories.

[1] http://www.robot-standards.eu/index.php?id=2
http://wiki.robot-standards.org/index.php/Main_Page
[2] http://www.best-of-robotics.org/

III. A GENERIC LOOSELY-COUPLED COMPONENT-BASED CSA

A. Overall View

GLOC3 has been devised around a typical configuration for the whole robotic system consisting of one or more *base stations* and one or more *platforms* (i.e. vehicles) that communicate with the base station(s). On the one hand, platforms are assumed to be able to respond to requests from the base stations for executing either individual *actions* or partially ordered sets of actions, which will be referred to as *tasks*. A platform can execute an action depending on its *capabilities*, i.e. perform a certain motion primitive, perceive surrounding obstacles, estimate the platform motion, etc. A *mission* can in turn be expressed as the execution of a partially ordered set of tasks. By definition, thus, both tasks and mission definitions can include the parallel execution of actions.

On the other hand, the communication links between platforms and base stations are intended to carry:

(1) *status information* and *sensor data* from the platform(s) to the base station(s), so that platforms' health and sensor data are remotely available at all times; and
(2) *commands*, from the base station(s) to the platform(s), requesting the execution of tasks, through *high-level commands*, or actions, through *low-level commands*.

This framework is general enough so as to cover almost any configuration. Besides, it permits dealing particularly well with the typical setups of underwater and aerial applications. In effect, in the first case, it is usual to have a surface station which, while the AUV is on the surface, uploads mission specification files to the AUV and downloads the data gathered while the vehicle was submerged. In the case of aerial applications, a ground station is typically supporting the vehicle, for data visualization purposes and even for processing sensor data off-board due to the payload restrictions of these platforms, which particularly limit the available processing power onboard.

B. Main Components of GLOC3

The execution unit in GLOC3 is the component, similarly to CoRoBa [12], OROCOS [11] and MARIE [16]. In accordance with the above-depicted setup, a total of five different types of generic components are distinguished within GLOC3:

- *sensor samplers* (SS), which implement the interface with the corresponding physical sensor and sample it at the requested rate;
- *data processors* (DP), which process the available sensor data and produce the elaborate estimations about the surrounding environment required by the current mission;
- *controllers* (CC), which, from the data produced by the data processors, generate commands to the vehicle actuators, so that the platform can progress towards achieving the mission goals;

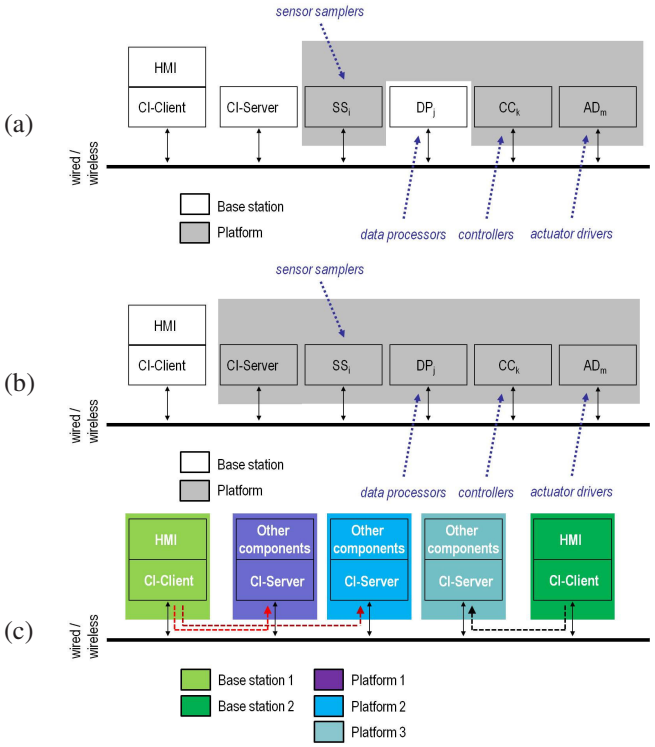

Fig. 1. Examples of setups: (a) UAV case, (b) AUV case, (c) several base stations (2) and platforms (3).

- *actuator drivers* (AD), which implement the interfaces with the physical actuators; and
- *command interfaces* (CI), which take care of the communication between a base station (client side) and a platform (server side).

Clearly, the first four types of component represent the typical elements of a control system as considered by the classical control theory. Consequently, they form a chain along which information flows unidirectionally, from sensor samplers to actuators. Regarding the command interfaces, they not only encapsulate the functionality required to support the distribution of the control software, but also are intended to make easier the interaction with the platform(s). To this end, they are devised as a finite state machine that interprets and responds to a set of well defined commands that directly refer to the vehicle capabilities, i.e. the sort of tasks it is able to carry out.

The information flow between the different components is supposed to be in the form of messages. The kind of messaging provided by a subscribe/publish mechanism — such as the one provided by e.g. ROS [10]— would be particularly adequate. However, GLOC3 is intended to be as independent as possible of a specific middleware, so that no election is performed in this regard. Nevertheless, since the interaction with a specific middleware will be unavoidable once chosen, this interaction is *hidden* within the generic component that is described in Section IV, in accordance with the maintenance simplification goal that has been outlined above.

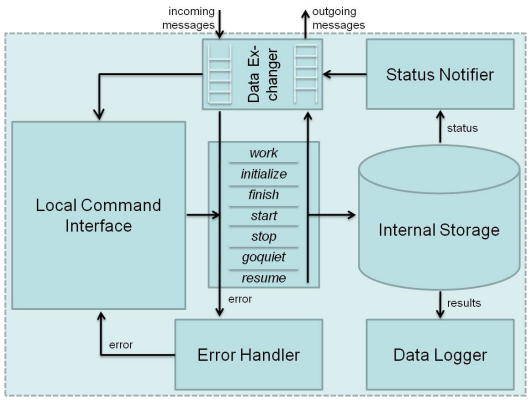

Fig. 2. Structure of the generic component of GLOC3.

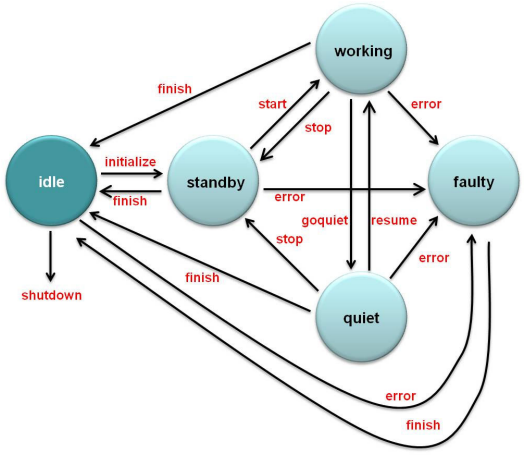

Fig. 3. Finite state machine implemented in the LCI of every generic component of GLOC3: states in blue, command messages in red.

To finish, the distribution of the control software is intended to be at the component level. In this regard, figures 1(a) and (b) describe how the control software components could be distributed among the available processors for, respectively, the UAV and AUV cases. Moreover, figure 1(c) illustrates the case of several base stations and platforms interacting all together and sharing the computational load, if necessary.

IV. THE GENERIC COMPONENT

Every component of GLOC3 follows a common structure which accounts for the generic functionality that every component is likely to make use of in one way or another. In this way, the specification and maintenance of usually needed functions is simplified. Specifically, every component is endowed with (see Fig. 2):

- A *local command interface* (LCI) which makes the component provide the functionality requested by the platform CI through the corresponding commands. To this end, the LCI is implemented as the finite state machine depicted in Fig. 3 and Table I. As can be observed, the component can be in one of five states: *idle*, *standby*, *working*, *quiet* and *faulty*. The transition from one state to another, after the reception of the corresponding command, entails the execution of, using C++ terminology, the appropriate method:

  initialize() — configures the component to the settings specified;
  finish() — resets the component and makes it wait for another configuration;
  start() — makes the component provide its functionality, reporting the results produced, if any;
  stop() — the component stops providing the intended functionality;
  goquiet() — the component continues working but no results are being reported;
  resume() — the component resumes the publication of its results.

  While in the *working* or *quiet* states, every execution cycle the LCI calls the *work*() method, which implements one cycle of the component specific functionality.

TABLE I
LOCAL COMMAND INTERFACE STATES

States	*idle*	Initial state of the component.
	standby	Component properly initialized or re-initialized.
	working	Component operating and publishing results.
	quiet	Component operating but not publishing results.
	faulty	An error that prevents the component from starting/operating has occurred.

  Observe that, by means of the *initialize*, *finish*, *start* and *stop* commands, the base stations can enable/disable any component and have total control about the system configuration.
- A *data logger*, which, if enabled, logs selected items from the component internal state at the end of every cycle.
- An *error handler*, which makes the component transit to a fault state if a malfunction is detected and keeps track of errors.
- A *status notifier*, which periodically reports the state of the component to the rest of the system.
- A *data exchanger*, which implements the messaging functionality and, thus, maps it to the specific middleware functions. It also handles incoming and outgoing message queues.

Table II enumerates the different kind of messages that can be used for interacting with the generic component. Apart from the messages oriented to make the component transit from one state to another, the message catalogue also includes *status* messages, for periodic health status reporting, and *function* messages, reserved for interacting with the component while it is in the *working*/*quite* states, and, therefore, they depend on the particular functionality the component implements.

As can be observed, GLOC3 components are fitted with minimum generic functionality. Some previous works in this regard, however, have tried to provide the component with as much functionality as possible, what leads to strange situations when the particular functionality of a given component

TABLE II
MESSAGES

Command messages	*initialize*	Makes the component initialize and adopt the configuration specified.
	finish	Disables the component.
	start	The component starts providing the intended functionality.
	stop	The component stops providing any functionality.
	goquiet	The component continues providing service but does not report results.
	resume	The component resumes reporting results.
	shutdown	The component disconnects from the rest of the system. Particularly happens during system shutdown.
Function message		To be used to interact with the component while it is providing its functionality.
Status message		Periodic reporting about the health of the component.

only requires a small fraction of all the generic functionality. In our opinion, this is the case of the *autonomic element* described in [19]. At the other end of the spectrum lies MCA [13], whose generic component is not that generic but oriented towards controller-type components [20].

V. COMMAND INTERFACES

Command interfaces (CI) are not considered different components, but are components whose functionality is to take care of the interaction between base stations and platforms (i.e. this is what they do while in the *working* state). Consequently, they follow the structure of generic components and are endowed with their generic functionality as any other component of GLOC3. This in particular entails that a CI integrates an LCI.

As indicated in Fig. 1, a CI acts as a client when runs on a base station, and as a server when executes on the platform side. In the former case, the CI interprets mission specification files, accordingly sends command messages to the platforms and, finally, also receives status messages from the platforms. In the latter case, the CI executes the commands received from the clients, accordingly sends commands to the components that are under its control and receives status messages from them, which are finally summarized and sent to the CI client.

To finish, observe also in Fig. 1 that CI clients are intended to be attached to Human-Machine Interaction (HMI) software for platform data visualization and interaction.

VI. EXAMPLE OF APPLICATION: A CONTROL ARCHITECTURE FOR A UAV

The blocks diagram of Fig. 5(left) corresponds to the control architecture and experimental setup involving two UAVs that are currently used to develop an aerial platform for visual inspection purposes in the context of the EU-funded FP7 project MINOAS[3]. Both vehicles are fitted with a front-looking stereo vision system, a ground-looking camera, a height sensor and an Inertial Measuring Unit; one of them also carries a laser scanner. Regarding the computational power onboard, apart from two ARM7 controllers that are in charge of platforms low-level control, one of the vehicles is fitted with a Gumstix Overo Fire board while the other carries an Intel Atom processor. The vehicles are the Hummingbird and Pelican self-stabilized platforms from Ascending Technologies. Fig. 4 shows one of these vehicles.

[3]http://www.minoasproject.eu

Fig. 4. One of the UAVs to be used in project MINOAS.

Essentially, the motion capabilities of the MINOAS flying robot comprise *attain a list of waypoints* and automatic *take-off* and *landing*. To this end, for localization purposes, the platform implements a navigation strategy based on two visual odometers using a front-looking stereo vision system and a ground-looking camera, as well as a laser scan matching-based odometer. The robot pose estimation produced is finally complemented with the estimation provided by an external optical follower able to track a vehicle fitted with a high-brightness LED. This redundant strategy is intended to provide robust positioning information able to tolerate the failure of any of the positioning subsystems in case of the vehicle getting out of the line of sight of the optical tracker or in case the vehicle motion cannot be estimated from the available sensor data.

Fig. 5(right) shows the control architecture for the whole system expressed in terms of the generic components of GLOC3. By way of illustration, Fig. 6 shows simulation results of a visual inspection mission for one of the experimental platforms: the left and middle screenshots come from the simulation/visualization tools employed (*Gazebo* and *rviz* over ROS), while the right plot compares the path followed by the vehicle (solid blue line) against the path requested as a list of waypoints (dotted red line).

VII. CONCLUSIONS

This paper has described GLOC3, a general control software architecture intended for unmanned vehicles. Applying a criterion of maximizing software maintainability and reusability, GLOC3 is based on a single minimalist generic component, providing basic but essential functionality, which allows dealing with the complexity of applications involving autonomous platforms. The loosely-coupled nature of this component allows for their reutilization without restriction.

REFERENCES

[1] Douglas-Westwood, "The World AUV Market Report 2010-2019," 2009.
[2] ——, "The World ROV Market Report 2010-2014," 2009.
[3] Teal Group, "UAV Manufacturers Market Overview," 2009.
[4] A. A. D. Medeiros, "A survey of control architectures for autonomous mobile robots," *Jour Brazilian Computer Society*, vol. 4, no. 3, 1998.

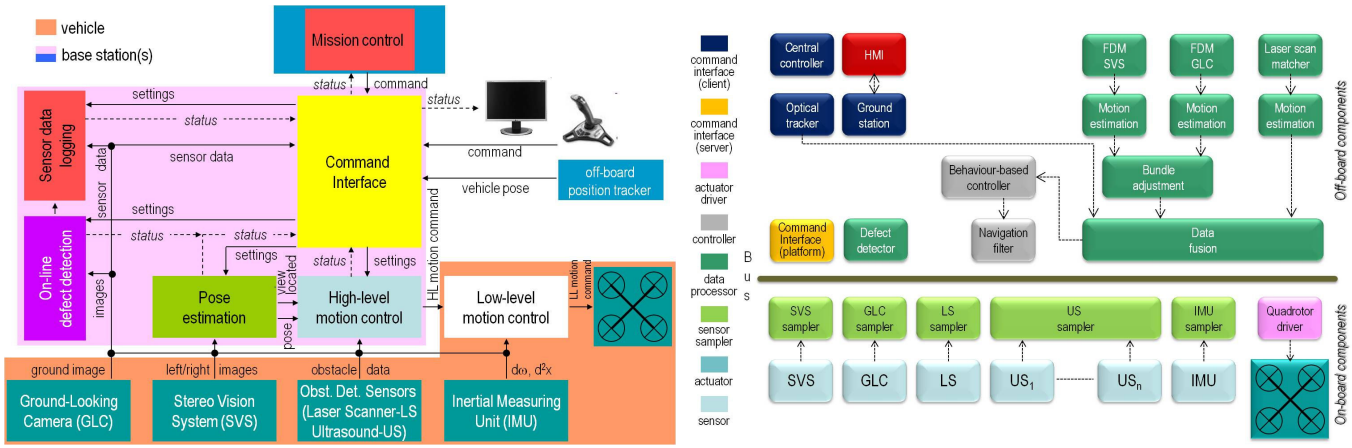

Fig. 5. (left) UAV HW/SW block diagram. (right) The UAV control architecture expressed by means of GLOC3 generic components. Although not explicitly indicated, all the components are connected to the communication bus. Dotted lines represent logical connections between components. (FDM stands for *feature detection and matching*.)

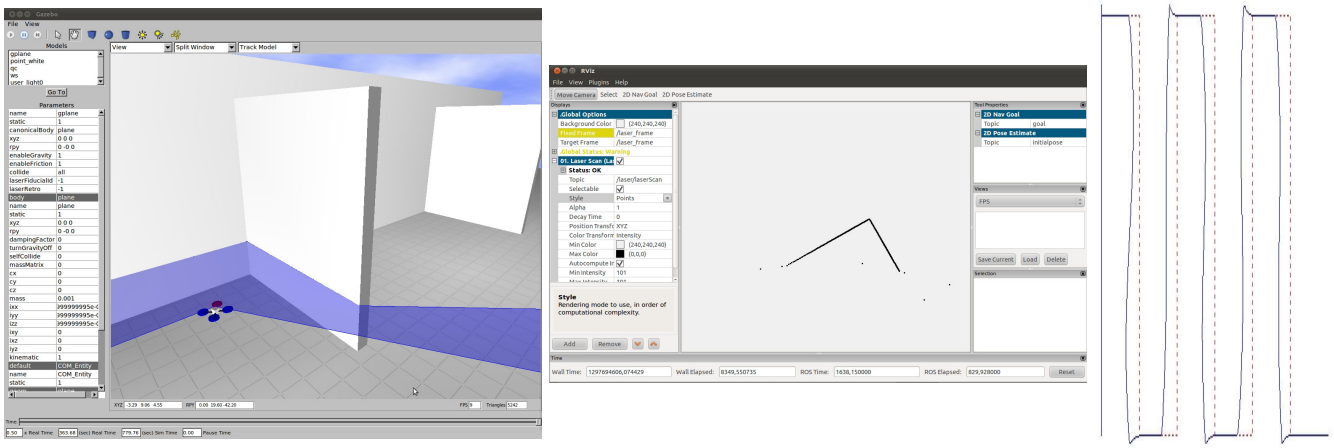

Fig. 6. Screenshots of a simulation of an exploration mission.

[5] È. Coste-Manière and R. G. Simmons, "Architecture, the Backbone of Robotic Systems," in *ICRA*, 2000, pp. 67–72.

[6] A. Orebäck and H. I. Christensen, "Evaluation of architectures for mobile robotics," *Autonomous Robots*, vol. 14, no. 1, pp. 33–49, 2003.

[7] R. C. Arkin, *Behavior-Based Robotics*. The MIT Press, 1998.

[8] Oracle, "Unmanned Vehicle Systems: Leveraging COTS Software, White Paper," 2007, http://www.oracle.com/us/industries/045962.pdf.

[9] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," in *ICAR*, 2003, pp. 317–323.

[10] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *Open-source Software Workshop (ICRA)*, 2009.

[11] H. Bruyninckx, "Open robot control software: the OROCOS project," in *ICRA*, 2001, pp. 2523–2528.

[12] E. Colon, H. Sahli, and Y. Baudoin, "CoRoBa, a Multi Mobile Robot Control and Simulation Framework," *Intl Jour Adv Rob Systems*, 2008.

[13] K. U. Scholl, J. Albiez, and G. Gassmann, "MCA - an expandable modular controller architecture," in *Real-Time Linux Workshop*, 2001.

[14] H. Utz, S. Sablatnög, S. Enderle, and G. K. Kraetzschmar, "Miro - middleware for mobile robot applications," *IEEE Trans Rob and Automation*, vol. 18, no. 4, pp. 493–497, 2002.

[15] L. Flckiger, V. To, and H. Utz, "Service-Oriented Robotic Architecture Supporting a Lunar Analog Test," in *Proc Intl Symp Artificial Intelligence, Robotics, and Automation in Space*, 2008.

[16] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud, "Robotic software integration using MARIE," *Intl Jour Adv Rob Systems*, vol. 3, no. 1, pp. 55–60, 2006.

[17] S. Rowe and C. R. Wagner, "An Introduction to the Joint Architecture for Unmanned Systems (JAUS)," 2008, http://www.openskies.net/papers/07F-SIW-089␣Introduction␣to␣JAUS.pdf.

[18] T. Galluzzo and D. Kent, "The OpenJAUS Approach To Designing And Implementing The New Sae JAUS Standards," in *AUVSI Unmanned Systems Conference*, 2010.

[19] C. Lin, X. Feng, Y. Li, and K. Liu, "Toward Generalized Architecture for Unmanned Underwater Vehicles," in *ICRA*, 2011.

[20] D. Doroftei, E. Colon, Y. Baudoin, and H. Sahli, "Development of a behaviour-based control and software architecture for a visually guided mine detection robot," *Eur Jour Autom Systems*, vol. 43, no. 3, pp. 295–314, 2009.