A RECONFIGURABLE MISSION CONTROL SYSTEM FOR UNDERWATER VEHICLES

Jorge Estrela Silva, Alfredo Martins*, Fernando Lobo Pereira

Faculdade de Engenharia da Universidade do Porto and
Instituto de Sistemas e Robótica
Rua dos Bragas, 4099 Porto Codex
Portugal

* Instituto Superior de Engenharia do Porto
Rua de S. Tomé 4200 Porto
Portugal

Abstract — **This paper describes the mission control software used in the LSTS/FEUP underwater vehicles. This software follows the guidelines of the generalized vehicle architecture [1], adapts the original idea to encompass the current application requirements and constitutes a first implementation.**

The work is focused on the design and implementation of an application that can be easily adapted to different vehicle configurations or even to different vehicles. One of the desired goals was to enhance software reusability and to establish a development environment that allows developers with a minimal knowledge of coding details to upgrade the application. To assist this purpose, a CASE tool, which provides modern software development techniques, was used.

A simulation environment was also developed whose purpose is to test the applications and to detect possible malfunctions before they occur during mission execution.

MOTIVATION

Porto University has been working on the ISURUS vehicle for two years. ISURUS is a REMUS vehicle created by the Woods Hole Oceanographic Institution (WHOI). Since the beginning we have been customizing the vehicle to fulfill our particular needs and interests. We upgrade the vehicle in order to minimize power consumption, increase data logging capacity, install new oceanographic sensors and increase vehicle operational capabilities. A new navigation system was developed for this vehicle using *Kalman* filtering techniques [2]. Soon we felt the need to adapt the original software to accommodate the new features.

Furthermore, as the research interests of this group encompasses topics in advanced control theory, we aimed at a framework allowing the implementation of several control algorithms and test them in a real environment. For that purpose, the software should provide capabilities for testing several control laws during the same mission, with minimum human interaction. It should be possible to schedule the same control law with different parameters and log the vehicle performance.

The original WHOI control software [3] used on ISURUS was targeted to DOS. It was decided to abandon this line of development since in a medium term it would be impractical to guarantee the static scheduling provided by the WHOI software. Using DOS would imply the need to divide carefully our longer tasks in chunks of processing, in order to provide pseudo multi-tasking. The multi-tasking would have to be explicitly coded leading to software harder to understand. We would be dealing with several tasks with different priorities and scheduling rates. Furthermore, it is desirable to have parallel development of several tasks what would be difficult and more error prone in such an environment.

Developing a multi-tasking OS or adapting DOS to that purpose was out of question given the time vs. usability/reliability trade-off and given the great variety of available products on the market. For those reasons it was decided to choose a commercial multi-tasking real time operating system.

Furthermore, since new related projects were started, demand for new software applications would justify the decision made.

Since the whole software would be rewritten, it was decided to specify an architecture that would promote software reutilization. We identified the processes, data flows, system events, communications, reactions to special events and maneuvers that would be common to a great majority of vehicles (as hardware independent as possible). The software and underlying control laws was organized in several layers and modules.

We adopted a commercial tool (TEJA) [4] to support the development of a first instance of the software, mainly on the simulation layer. This tool allowed us to graphically model the dynamic behavior of the application using the hybrid state machine paradigm. The interaction (data flow and event triggering) between different tasks is also graphically modeled by using the OMT methodology. In the end, it generates C++ code that must be built to the target OS.

ARCHITECTURE

The application organization follows a hierarchical layered model, with well defined interfaces and access points. This organization allows the development team to rapidly locate the desired changes or upgrades and to assign those tasks.

Several agents, running concurrently, are defined at each layer. Each agent will manage a specific subsystem encompassing the relevant information. This approach ensures functional separation, thus reducing the side effects of future upgrades.

Libraries of device drivers for sensors and actuators and algorithms for control, guidance and fault management will be created as new situations arise. These algorithms can be developed for a specific vehicle configuration or can be designed in such a way that its behavior is a function of the vehicle model parameters, thus permitting code reusability. The TEJA CASE tool allows the developer to graphically choose the set of components that will constitute the application for each vehicle configuration.

The architecture is intended to be operating system independent. However, the implementation depends on the capabilities of the operating system. We assumed that any operating system we will use will have multi-tasking capabilities. The remaining desired features (scheduling, priorities, message passing, communications, service identification via names) are encapsulated in classes whose implementation can be adapted to different operating systems. The application tasks rely on those classes instead of direct system calls. The overhead of this methodology was shown to be negligible on the ISURUS 486DX computer.

1 – Abstraction Layer

In the first layer, we can distinguish two different sublevels:

- The tasks which deal with interrupts and low level routines (device drivers). Normally the OS shall provide these modules: drivers for the serial ports, network adapter, disk drive and console, and access to memory mapped devices. These tasks, as also the bulk of the OS kernel, have the highest priority. A communications stack like TCP/IP or some real time protocol is also needed (on ISURUS we used a TCP/IP stack for PPP connections).
- The tasks provide the high-level protocol to interface with the specified device: knowledge of messages formats and timings, message parsing, processing and handling, information gathering and fault tolerance.

We adopted a distributed and hierarchical fault-tolerance scheme. For a given abstraction level, the identities ensures robustness by interpreting lower term fault detection data and either by undertaking the pertinent error recovery or by forwarding it to a higher abstraction level. For instance, occasionally, the vehicle serial ports stop responding. The task associated with each serial port has the duty of checking the time since the last character was received. The knowledge required to infer whether the serial port is dead or not is embedded in that task, since that is intrinsically related to the device protocol.

Every task on the system can log their messages. Besides the normal data gathering performed by the oceanographic sensors, the log file also behaves as the vehicle "black box". To log all the available information concerning vehicle status (temperature, voltage, power consumption, etc.), whenever possible, has proved, in the course of field missions, to be very useful. We have already discovered and modeled a conflict between two devices only by inspecting the log files. For this reason we think AUVs should be provided with enough data storage capability (several hundreds of MB). For ROVs, the information can be logged via data link on a remote computer.

Data communication between the first and second layers is done via shared memory. The application processes use a library whose aim is to provide abstraction of the set of system devices. Layer 1 processes broadcast their data calling the functions provided by this library.

By using this approach, when changing or adding an instrument, we only have to create a new process. There is no need of rebuilding the remaining software.

2 - Functional Layer

The components of this layer are virtual sensors and groups of functions that basically provide the motion and navigation operators. This layer controls redundancy and supports fault-tolerance procedures so that alternative modes of operation are available for dynamic reconfiguration.

Layer 2 processes send commands by invoking a function which puts the data on the shared memory and signals the target process. The details of process identification are hidden by this library, which behaves like a device database. For instance, on layer 2, the navigation process can ask "give me attitude data" or "give me range to a position" with no priori knowledge of the sensors installed. On the other hand, they can also query the set of installed devices and test whether a given device (which can be identified by its name) is present.

Positioning takes place at the function layer. A positioning algorithm must be implemented for each set of installed sensors. Normally, during one mission, each vehicle uses only one navigation algorithm, with several modes, that will try to extract the best result from the installed sensors. However, it can of interest to select different algorithms in order to compare results during a given mission. Eventually, two modes (for instance, due a total failure of a special sensor) might be so different that both algorithms have to be specified separately.

Another block of the functional layer is the virtual world. Its function is to map the real world objects and important phenomena. It can be used by the coordination level in order to provide mission re-planing due to detected obstacles. In a multi-vehicle environment, provided communication with the vehicle takes place, this virtual world can be updated with data collected from each vehicle. We're also planning to estimate sound speed along a given segment based on distributed data.

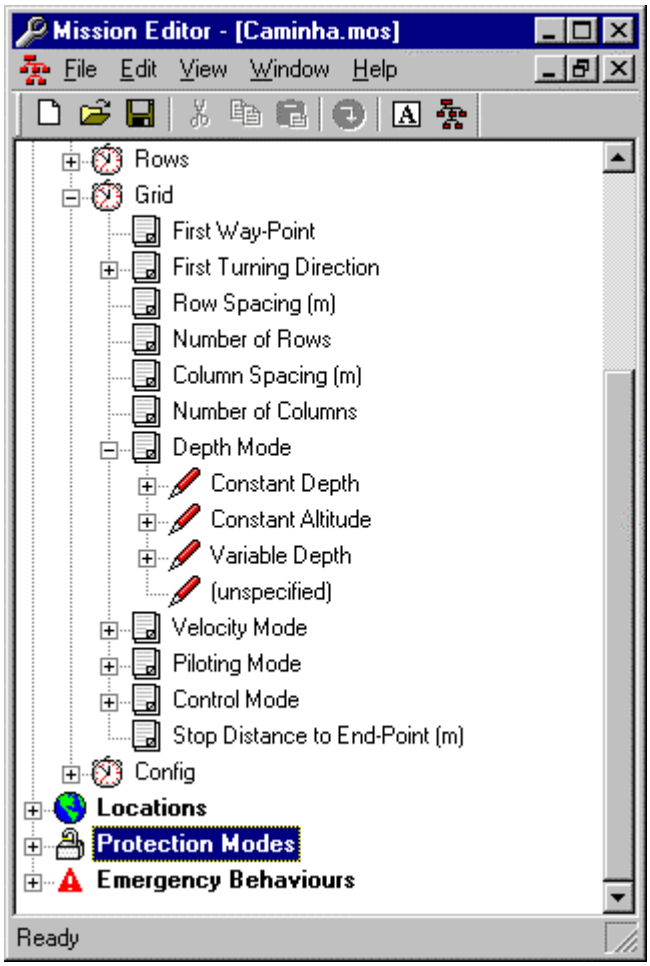

Figure 1 - The template for the Grid maneuver in the mission editor.

3 – Coordination Layer

The function of the coordination layer is to accomplish correct mission execution and tolerance to unexpected events. Tasks are scheduled according to vehicle state and received events, having in view the mission correct execution. A mission is organized in a set of several tasks picked from the functional layer ("Objectives" on the WHOI nomenclature or "behaviours") which are loaded either from a file at vehicle power-on or dynamically via a remote data link. A behaviour can be an atomic command (like triggering a device) or a maneuver. A maneuver has hooks for trajectory generation algorithms, guidance algorithms and motion control. The desired algorithms can be chosen from the available set on the functional layer. The choice is made via mission file.

Other tasks can be made active asynchronously, mainly due to fault detection or user interaction. Besides event signaling, the coordination layer is fed by the functional layer with additional data and conditions that can be scheduled for verification (for instance, "no deeper than …", "keep vehicle in a given geographic zone"). When the condition is violated an event is triggered. The response of the coordination layer to each event is defined in the mission file and can be set different for each individual mission stage. Those responses are implemented by recruiting resources from the functional. A small set of emergency behaviors is specially hard-coded in order to guarantee the ultimate robustness.

Whenever a faulty situation is detected, the current mission is either paused or aborted, depending on the seriousness of the situation, followed by mission or vehicle recovery.

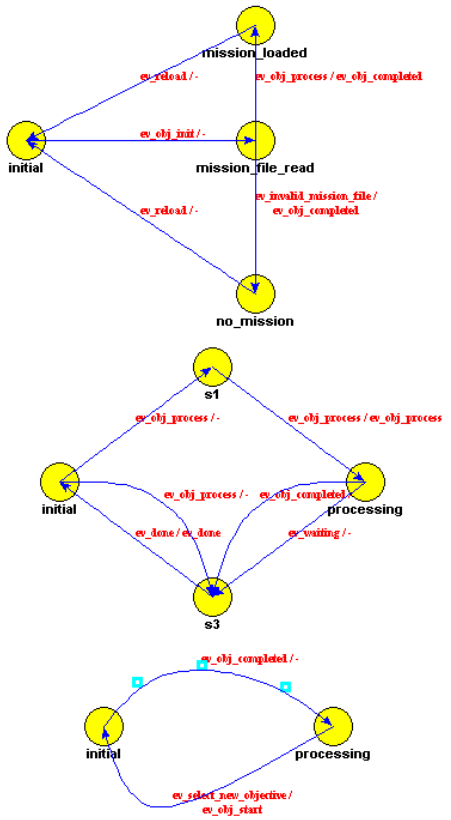

Figure 2 – State diagrams of the mission coordinator

IMPLEMENTATION

As stated previously, the described architecture was followed in the implementation of the software for the ISURUS AUV.

After a market survey, we chose QNX to be the operating system. QNX is a real-time, extensible POSIX-certified OS with a small microkernel and a set of optional cooperating

processes. This architecture allows us to scale QNX to the particular needs of the vehicle. Although, the memory requirements were much bigger than we expected.

Although its deficiencies on thread support (multitasking is UNIX-like with processes running on separated addressing space), QNX provided most of the required function: process prioritisation, FIFO scheduling, message passing system, traditional Unix IPC mechanisms, standard real-time timer functions (up to 0.5 ms precision) and total access to the underlying hardware.

The most natural way of designing application on QNX is using the blocking-mode paradigm. The processes stay blocked until receiving a message or a proxy. A proxy is a kind of signal that can be attached to special events (timers, data arrival, etc). The message passing system used by QNX is very powerful, providing inherently mutual exclusion: when a process A sends a message, it stays blocked (if we do not want the processes to block we must use a message queue) until the destination process B issues a receive call. When a process issues a receive call, it also stays blocked until it has some pending message. When process B receives the message, process A remains blocked until process B replies. The send-receive-reply sequence can be seen as invoking a function which is resident on another process.

On the abstraction layer, we implemented processes to deal with each of the following components:

- Compass and inclinometers
- Altimeter
- CTD
- Acoustic system
- Motors
- AD, counters, and parallel interfaces.

This layer also include the data logger process. The priority of the data logger process is client driven. We do not want a higher priority to block because it is dependent of the execution of a lower priority process. The same behaviour could not be achieved even if we set the data logger with the highest priority. In that case, a process could be preempted by the data logger which eventually could be doing processing to a lower priority process.

Although, formally, the functional layer and the coordinator layer are distinct, in the current implementation the simplicity of the later suggested a tight coupling between them. Thus we decided to keep them in the same process.

SIMULATION

When writing new modules or adding new maneuvers it is essential to have a straightforward way of performing some tests “on the bench”. For testing purposes, the first software layer can be replaced by a simulation layer that mimics the normal operating mode of the vehicle. This architecture allows the developer to test new tasks with no need to build the usual test stubs. Furthermore, the new task will run integrated with the remaining application, providing an environment as similar as possible to the real operating mode of the vehicle. This way it is easy to find unexpected interactions with the remaining tasks.

The simulation layer is composed by the task which performs the simulation calculations and several other tasks which model the physical devices. The physical devices are divided into sensors and actuators. This tasks interact with the above layer using the same interface (functions that access a shared memory segment for data communication, sending messages to the data logger or triggering events using the event manager) and with the simulation task through local variables. We modeled the actuators dynamics and power consumption and added noise to sensor measurements. One interesting achievement was the modeling of the acoustic system: the navigation task was tested in a virtual environment by simulating the presence of two transponders (lockout times and delays were accounted for), currents and constant sound speed. The behavior of the vehicle was also simulated with the implemented PID controllers and guidance algorithms.

The model uses the decoupled equations of motion from Fossen [5]. The parameters were obtained from water tests. The integration of these equations is made at a fixed frequency using a simple integration technique [4].

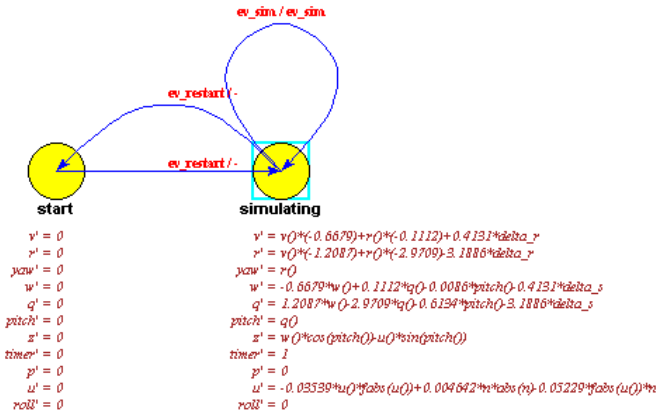

Figure 3 – Dynamic model of the ISURUS vehicle

The main constraint of this implementation is to keep the capability of running the simulation process on the target (the vehicle computer) keeping real-time operation. This was achieved because the vehicle models, although realistic, were kept simple as also the integration algorithm. It is not possible to have simulations running faster (or slower) than physical time but the system was not intended to do so anyway.

The model equations used were here strongly customized to the ISURUS vehicle but our goal is to implement a 6DOF model having the parameters read from some configuration file. Obviously, with a different vehicle we will have different physical devices whose models can be easily implemented according to this architecture.

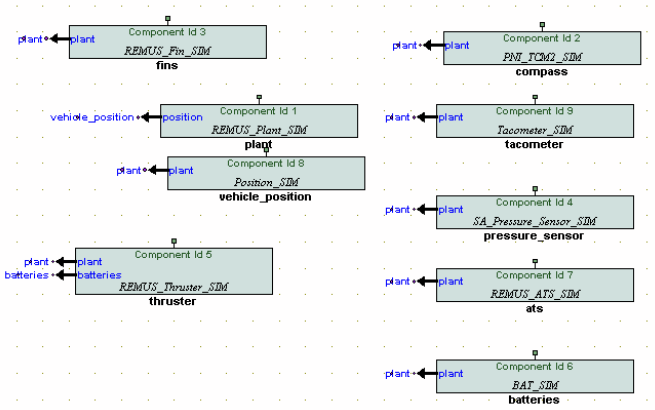

Figure 4 – Block Diagram of the Simulation Layer for the ISURUS AUV

MISSIONS

The vehicle has already executed several missions using the new software. Since the beginning, the only software difficulty we experimented was related with disk consistency. QNX file-system is not very tolerant to system reboots.

We use a Point-to-Point Protocol (PPP) connection to download the mission objectives to the vehicle or upload the log files. This approach allows us to use standard applications (FTP, telnet) to perform the routine operations. It is even possible to change the code and rebuild it on the vehicle in execution time, although we only used this feature a very few times in test missions.

Missions have been specified with the help of an application (Figure 1) with graphical interface that simplifies the whole process. The mouse is used to insert the desired mission tasks. The user must only fill the presented input boxes with the desired parameters values. In the end, the application generates the mission file (which can also be manually edited). This is a improvement when compared to writing text files.

CONCLUSIONS AND FUTURE WORK

The control architecture has proved to be useful on helping to point out the most important aspects of the software organization and establishing a solid framework. The relevant modules for the current application were identified and implemented.

The current implementation allowed us to improve our REMUS vehicle by upgrading the navigation subsystem, adding new maneuvers and actions and changing vehicle operation from a end-user perspective. We consider it to be a major improvement to the original WHOI software. The combination of the QNX operating system and our application revealed satisfactory.

We are concluding several lateral applications (mission editor and applications for data warehousing, visualization and retrieval) which will improve our operational capacity.

In a near future, this framework will be used in related projects, as also great part of the implemented software.

REFERENCES

[1] J. Borges de Sousa and F. Lobo Pereira. A Generalized Vehicle Control Architecture for Multiple Autonomous Vehicles. In *Proceedings of the IEEE AUV'96 Conference*, Monterey, CA, USA, June 1996, 223-30.

[2] Aníbal Matos, Nuno Cruz, Alfredo Martins and Fernando Lobo Pereira. Development and Implementation of a Low-cost LBL Navigation System for an AUV. To apper in the *Proceedings of the MTS/IEEE Oceans'99 Conference*, Seatle, WA, USA, September 1999.

[3] B. Allen, R. Stokey, T. Austin, N. Forrester, R. Goldsborough, C. Alt. REMUS: a small, low cost UV; System description, field trials and performance results. In *Proceedings of MTS/IEEE Oceans 97 Conference*, Halifax, Canada, October 1997.

[4] A. Deshpande and Joao Borges de Sousa. "Real-time Multi-agent Coordination using DIADEM: Applications to Automobile and Submarine Control." *1997 IEEE Conference on Systems, Man and Cybernetics*, Orlando, Florida, October 1997,.

[5] T. Fossen, Guidance and control ocean vehicles, New York, John Wiley & Sons Hall, 1994.