PATTERN ORIENTED DESIGN OF THE CONVENIENT HIERARCHICAL AUTONOMOUS STATE MACHINE FORMALISM FOR AUV HIGH LEVEL CONTROL

Benoit Dupire
Department of Ocean and System Engineering
Florida Atlantic University
bdupire@seatech.fau.edu

Andres Folleco
Department of Ocean and System Engineering
Florida Atlantic University
afolleco@seatech.fau.edu

Samuel M. Smith
Brigham Young University
Provo Utah
21271 Waycross Drive Boca Raton Florida 33428
smithsm@samuelsmith.org

Abstract

The complexity of AUVs and of their control systems and associated software is ever increasing, due to continuous demand for more advanced features and capabilities. There is a need for broader, more general use of these vehicles, along with demand for longer, more complicated missions. FAU's AUV are presently limited by the current lack of maturity of task management architectures that can be placed on-board. Several desirable additions have been identified, such as being more capable in the autonomy underwater, performing conditional execution of tasks, providing error diagnosis and recovery capabilities.

The Morpheus, the latest generation of FAU's AUV was designed in this frame of mind, as well as being as modular as possible. The software must reflect these improvements: it should be as dynamic as possible, must adapt to the different payloads and missions, allow for complex behaviors and be dynamically reconfigurable by plugging in some new behaviors at run-time. The software for the AUV must support the following features amongst others: event driven more complex single vehicle missions, failure detection and handling, multiple cooperative missions. This paper describes the object-oriented design of the new high-level control architecture [2] developed for the Morpheus, referred to as Convenient Hierarchical Autonomous State Machine (CHASM) architecture, based on the hierarchical state machine formalism [1] [2] [5] [10]. The new design allows the control system of the AUV to perform conditional execution of high level-tasks, while reacting to the environment it moves in and being sensitive to the AUV's health. Python, a 4th generation language, was chosen to implement the distributed object-oriented software. The system is modeled as a set of concurrent CHASMs, communicating through dynamic shared memory and can be quickly reconfigurated to accommodate a variety of sensor payloads. It supports effortless integration of multiple modules with time and ordering constraints that cooperate or compete with each other. Object-oriented analysis and the use of design patterns were consistently applied during

the development. UML (unified modeling language provides an excellent framework for requirement capture and analysis. Its features are used throughout this project to explain the different steps of the design.

Introduction

The Morpheus, the latest generation of AUV developed by the Advance Marine Systems Lab in the Ocean and System Engineering department at FAU, has been designed to be as modular as possible [17] [18]. Its length is variable, depending on how many payloads have been added. On board sensors can be activated at different times during the mission, providing an extensive range of applications over a wide range of spatial and temporal scale. It has been designed to serve as a testing platform for many possible kinds of payloads. The software for the Morpheus must reflect this improvement: it should be as dynamic as possible, must adapt to the different missions, to the different payloads and allow for complex behaviors.

Due to the large variety of critical tasks an AUV must perform, a robust multilevel software framework is therefore essential. The AUV's framework is responsible for assembling its sensing, planning and control components into a coherent system. This framework constitutes the high-level control architecture of the soft real-time software. It also refers to the interactions between these components that provide the on-board intelligence required by the vehicle to autonomously perform its mission. A good software architecture for a given vehicle enables it to accomplish its mission, protects it from damage and eventually allows for the pursuit of elements of the mission under abnormal or unexpected conditions. To achieve this aim, we must find the most efficient way to organize the different functional components of a such system.

Requirements

We need a component-based software architecture for the high-level control of the AUV which has to provide advantages such as software reuse, flexibility, expendability, improved maintainability, and on-the-fly software reconfiguration. The Morpheus software is presently limited by the current lack of maturity of task-management architectures that can be placed on-board. There is a need for more sophisticated mission plans. We would want to add several features to the actual software of the Morpheus:

- Conditional execution of tasks during missions
- Failure Detection and Handling
- Exception handling
- Event driven, more complex single vehicle missions
- Multiple vehicle cooperative missions

The development of AUVs and supporting software tools is a complex process. Software modules are often added during the developmental stage for various reasons such as supporting a new sensor or implementing a new algorithm. It is important for the overall software architecture to be able to support these additions in a smooth manner so that the additions are localized and warrant minimal or no changes to the rest of the system. In addition, the architecture should support effortless integration of multiple modules with time and ordering constraints that cooperate or compete with each other. In other words, the software architecture should be scalable.

Objectives

The objective is to design a more powerful framework for command and control. This framework is based on the formalism developed by Anouk Girard in [1][5]. This new framework is referred to as the CHASM framework, for Convenient Hierarchical Autonomous State Machine.

The term Convenient Hierarchical Autonomous State Machine is used to describe a single hierarchical state machine (in which states are hierarchically organized) with additional features as described in [2]. Each CHASM represent a particular kind of high-level controller which is assigned a given task (for example the control of the speed, or the depth of the AUV). CHASMs are built dynamically from some text streams. Each CHASM is under the responsibility of a manager, which runs it on a cyclic basis. Each manager runs a separate process and Inter-Process Communication (IPC) is done through a dynamic shared memory (SM) (Figure 1). In particular, the Navigator manager has to be entirely rewritten as a CHASM created from a mission plan whose syntax remains accessible to the non-expert users.

Each state of a CHASM represents a mode of the system and is associated with some actions (or behaviors) which are components that have to be

fired. We transit from state to state based on the progression of the AUV.
The CHASM framework is therefore made of generic components, i.e. components that are both hardware and application independent. Behaviors are application dependant but can be configured for different types of hardware. The CHASM framework supports the plug in of reusable behavior objects that could implement different portions of the command and control logic (i.e the plug-in of new components that can be triggered by the CHASM).

Most of the different modules of the actual software will be redesigned and implemented in Python, using the CHASM formalism. We have to interface the new high-level software architecture with the other processes which remain untouched, or modified (logger, monitor, etc.). These interactions define the interface of our system with the current AUV software. As a matter of fact, the interactions between processes are only done through shared memory, which acts as a conduit for the exchange of the data between the various processes of the system. The different CHASM processes therefore have to

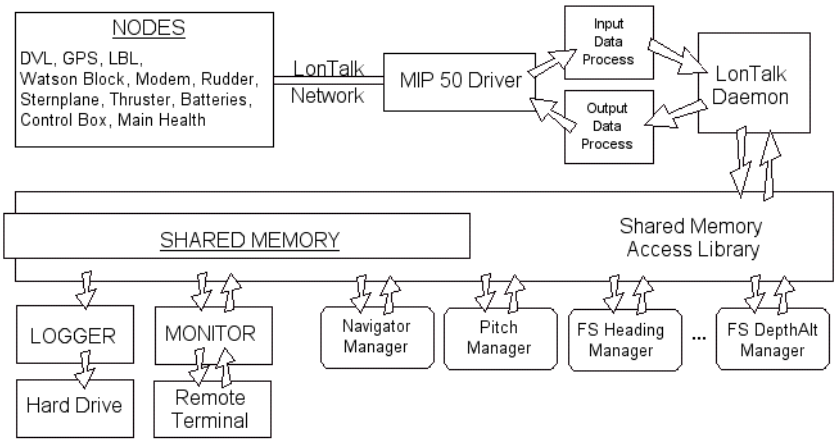

Figure 1: Communication Model of the Morpheus AUV:
Each manager is implemented as a Convenient Hierarchical Autonomous State Machine (CHASM)

The CHASM architecture is designed using the object-oriented paradigm, and hence special attention must be taken to ensure the observability and traceability of the system.
As previously mentioned, the chosen language for the implementation is Python, a fourth generation language and consequently some interfaces (API) and some wrappers have to be implemented to access the different specific QNX functions, especially the access to shared memory. Because AUVs have relatively slow dynamic time constants soft real time operation is typically sufficient.

Use of Shared Memory.

The software of the Morpheus AUV consists, like his OEX predecessor, of several processes cooperating together to accomplish the mission goal. The Operating system they run on is QNX 4.25 and shared memory was chosen as the way to do Inter Process Communication, because it is both simple and efficient (figure 1).

interface with the Shared memory so that they can communicate with the rest of the system.

Upgrability through Modular Design

Instead of developing a solution with built-in behaviors, some work has to be done to provide a solution that offers a modular design to add, remove or modify control features, one came up with a modular solution that can be assimilated to what is commonly called ‘plug-in’. That is, modules that can be easily inserted into an application to offer new features. The design of the architecture has to be developed so that new functionalities can be added and included into the software without modifying the code or recompiling it. Moreover, there should be no need to stop the mission and the software to update or add a new behavior and there should be no additional installation to do in order to use the plug-ins which are downloaded through RF and initialized automatically by the software if they are used in the mission plan for a CHASM

manager (i.e. only when needed). This should constitute a very efficient way to extend the capabilities of a CHASM and is also greatly benefit future developments. Indeed, future developers will only need to know how to design a new behavior - a new plug-in - to extend the application without knowing anything else about the CHASM architecture.

Pattern-Oriented Analysis

Designing a modular, flexible, and dynamic software is hard. The use of the object-oriented paradigm is therefore a prime choice for our system. It helps find pertinent objects, factor them into classes at the right granularity, and establish key relationships between them. Our design has to be specific but must also address future problems and requirements. To ensure reusability, modularity, and ease of use, our design has to be enhanced by the use of some principles, and good solutions that have worked for others in the past. These solutions are recurring patterns of classes. The combination of these patterns provides a powerful solution to our requirements. We have to take full advantage of the tools that object-oriented analysis and design methods offer to design the architecture of the software of the AUV. The paradigm is well suited to implement subsystem frameworks, which we would like to specify (for each of the manager in the system) and to reuse them for the development of similar subsystems.

UML (unified modeling language) has received much attention when it comes to modeling of requirements. It provides an excellent framework for requirement capture and analysis. Its features are used throughout this paper to explain the different steps of the design.

Requirements Analysis

The purpose of the requirement analysis is to capture and analyze the problem domain and the requirements on the system to be built. The feature of the embedded system consists in making the different objects of the system communicate between each other. Each manager (controller) is viewed as a black box and only the objects visible on the system boundary and outside the system are modeled. The goal is first to identify the active objects (also called actors). This is done from the analysis of the last paragraphs. The actors are the exterior elements in the system.

Our goal is to design the high level architecture of the FAU Software and thus, we define our system as the set of managers that manage the AUV. The different actors are consequently the actuators, the sensors. The managers get the data

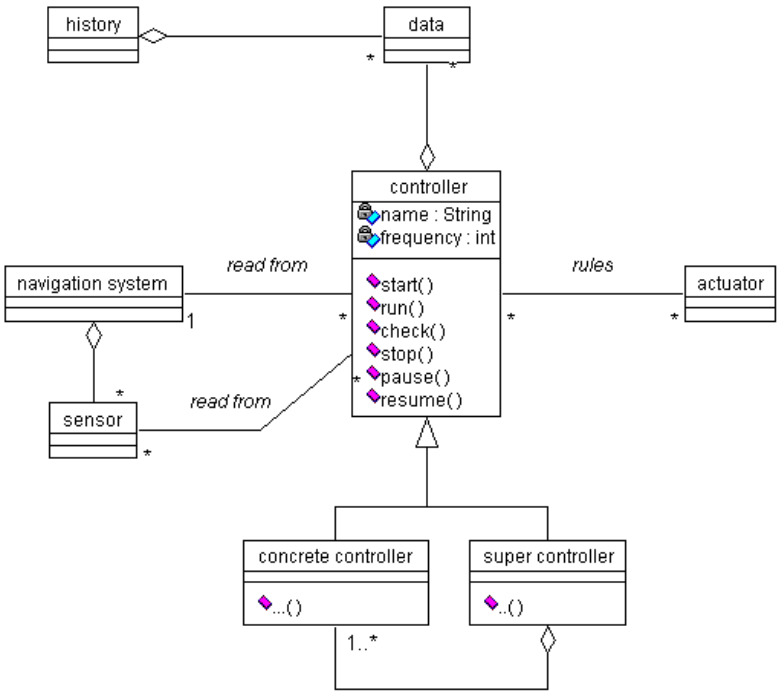

Figure 2: OOA of the control Architecture of the Morpheus AUV.

from the sensors, do some processing and then write the results back to the actuators, in order to satisfy the pre-loaded mission plan. Each manager (controller) is viewed as a black box and only the objects visible on the system boundary and outside the system are modeled. The use of the shared memory is transparent at this level, because we do not care about the implementation details at this level.
The class diagram (figure 2) shows the interaction between the controllers and the different agents exterior to the system. This abstraction simplifies the object behavior design for the presentation. The abstract model of the system is very generic, easy to understand, and supports the development of software for real-time autonomous robotics systems.
Our design deals with several controllers (CHASMs) running concurrently. We are then able to assign one of these controllers the role of initializing and orchestrating the other ones. It becomes a super controller which is able to control the other ones with different commands (ex: stop, start, resume, ..)
To keep this design as powerful and flexible as possible, we allow for several super managers (or controllers) running concurrently, each of them managing their own set of regular controllers. The instances of the controller are commonly called managers in the AUV terminology.

System Analysis

The purpose of the system analysis phase is to describe the architecture of the controller and to identify the classes that are needed to implement the required functionality. The controller design we propose is based on the CHASM Formalism. The focus is on designing a generic architecture for the different managers. Each manager then tailors this generic architecture so that it fulfills its specific needs. Most of the objects can be easily identified in the CHASM formalism.
A CHASM consists of a tree structure of states, with one or several root states at the top. Actions are associated to these states. An entity has to manage this set of states and to trigger the transition and the different actions associated to the states in the current active path, on a regular basis. The CHASM manager is this entity and ia responsible of initializing the CHASM, managing the CHASM tree, and keeping track of the active state path. Indeed, a CHASM manager has to run the CHASM structure at a regular frequency to ensure the good execution of the mission.
Consequently, the different objects of the system are the CHASM manager, the parser (or Loader) which loads the CHASM, the actions, the states, the transitions, etc... The other objects are the actors defined in the previous section.

Management of the CHASM

The responsability of the 'CHASM manager' is to run the CHASM at regular intervals. This functionality can be decoupled into two separate tasks: the CHASM Management itself and the scheduling. Some operations will be responsible for testing transition conditions and taking transitions, and some other operations will be responsible for scheduling the first operations. In other words, we can make the architecture more flexible and modular by having two separate objects each one focused on one task.
The first object is the *CHASM manage*r: it's the entity which is responsible for managing the states, the transitions, etc. This CHASM manager have to be triggered on a regular basis by the other object, named *CHASMSchedule*r, whose role is to schedule the CHASM manager (Figure 3).
The *CHASMScheduler* schedules the *CHASM manager* at a certain rate but the scheduling depends also from the commands coming from the super-CHASM (ex: stop, run, pause, resume). Some controllers will run at 8 Hz, and 16 Hz is the fastest rate at which a manager is run in the vehicle, which in today's world of real time systems is very slow. The system is soft real time, because the AUV has relatively slow dynamics and motion, and thus we can tolerate occasional missed deadlines and jitter.

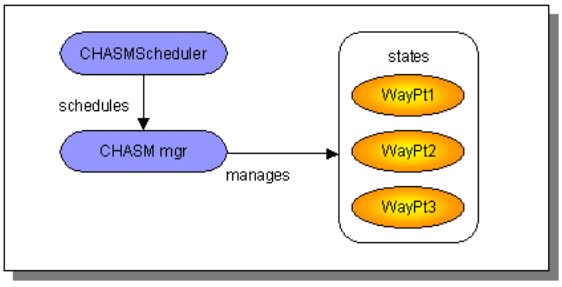

Figure 3: CHASM Management decoupling

The *CHASM mgr* can be seen as a delegate. The *CHASMScheduler* delegates the CHASM management to the *CHASM-manage*r. It can also be said that the *CHASMScheduler* class use the

behavior of the *CHASM Manager* by keeping a *CHASM-manager* instance variable and delegating CHASM manager-specific behavior to it. New functionality is obtained by assembling or composing objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called "black-box reuse", because no internal details of objects are visible. Objects appear only as 'black-boxes' (as shown in figure 3)
Object composition is defined dynamically at run-time through objects acquiring references to other objects. Because objects are accessed solely through their interfaces, we don't break encapsulation. That won't be the case if we had only one big class. Because an object's implementation will be written in terms of object interfaces, there are substantially fewer implementation dependency. Object composition has another effect on system design: it keeps each class encapsulated and focused on one tasks. The system therefore remains more flexible and modular, and not to have a unmanageable 'Manager' class which both manages the scheduling and the CHASM. We compose behaviors: in fact, the CHASMScheduler could schedule something completely different, it would not know anything about it. Delegation is a good design choice because it simplifies the design.

State data structure.

One approach one might use for implementing a State Machine using the Object Oriented Formalism is to instantiate a single State object for all the mission. The state (i.e. the instance variables) of this State object then varies when a transition is made from one state to another one.
When a transition is taken, the CHASM manager reads from a database (or from a file or whatever) the characteristics of the state we have just made the transition to, and initializes the state object with these values. Another approach is to instantiate a new State object at run-time, each time the system makes a transition to another state. These approaches are not doable here because we are dealing with a Hierarchical State Machine, so it is much more convenient to generate all the states at once, and then to transit from one to another.
They are several reasons to this:
• As we deal with a Hierarchical Finite State Machine, not just one state is active at any one time but a path of states. We have to check the transitions of the state with a higher level of priority first.
• There are special relations between the states: parent/child, default next, default child, etc.
• We do not want to access a file or a database every time we make a transition, it's time-consuming.
We want to have all the objects loaded at init-time. All the states are generated at init-time by the Loader. The role of the Loader is to parse the mission plan and to generate all the objects of the CHASM architecture.

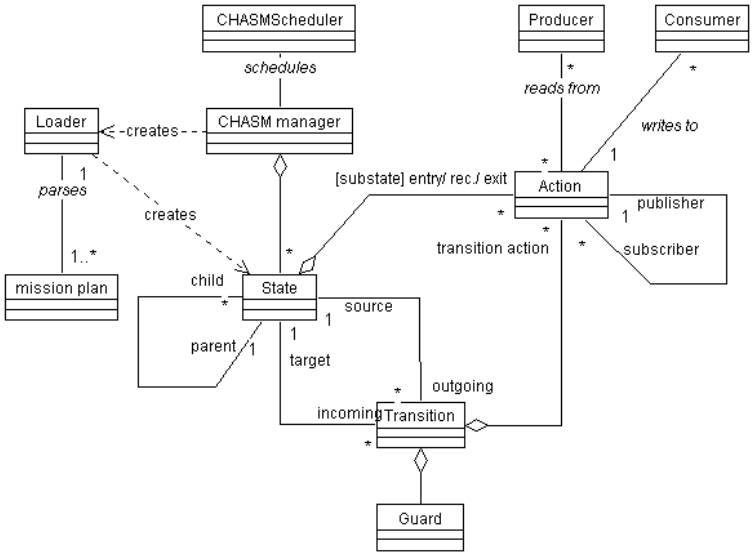

Figure 4: OOA of the CHASM architecture

Analysis Class Diagram

This stage of the design captures the relations between the different objects: The CHASM manager manages the CHASM data structure (a forest of state objects) and hence acts as a container for the different states of the system. A state may have several substates to allow for the hierarchy property, or can be a leaf state. Only certain states are active in the CHASM at one given moment. They belong to the current active path, which starts from one root node and ends with a leaf state. Some actions are associated to the states, and they are run as long as the system is in this state. There are different types of actions with a different time-model, so all the actions associated to the current active states are not fired every time the CHASM manager is fired. The class diagram in figure 4 describes the static relation of the different classes at this early stage. *Producer* represents any entity we can have data from, mostly sensors, but also the INS (Inertial Navigation System). The role of the INS is to merge many sensor data and to synthesize different parameters to provide the position and the speed of the vehicle. This is done at the low-level in our system, and data are emitted by one of the node of the LON-Talk network. *Consumer* represents any entity we can send data to, mostly actuators, but also low-level controllers which are connected to the LON-Talk network. The Actions get the data from the sensors, from the INS system, or from the results of other actions and do some processing, before publishing the results, either to some consumers or to other actions. The class diagram allows us to make appear the 'one writer-many readers rule'.

Design of the CHASM

The objective is now to take advantage of the design patterns. Design patterns are solutions to well-know problems that occur again and again. A design pattern names, abstracts and identifies the key aspects of a common design structure that make it useful for creating object-ori-ented designs that are flexible, elegant and ultimately reusable.

Use of the Command Pattern

In the context of a finite state machine, states and transitions objects have to fire a list of behaviors to undertake some actions. The CHASM application is a generic application (a tool kit) which issues commands without knowing anything about the operation being requested. This problem could be solved with the Command Pattern [8]. This is illustrated in Figure 5. As a matter of fact, we would like the CHASM to be able to fire any type of actions, without knowing anything about it. We must avoid hard-coded requests, to make it easier to change the way a request gets satisfied both at compile-time and at run-time. The Command Pattern allows us to encapsulate a request as an object, thereby letting us parameterize the states with actions.

The key to this pattern is to declare an interface for executing operations. This approach allows us to decouple the state from the operations it has to perform. We don't end up with a tightly coupled monolithic implementation, where the action is an operation of the State object.

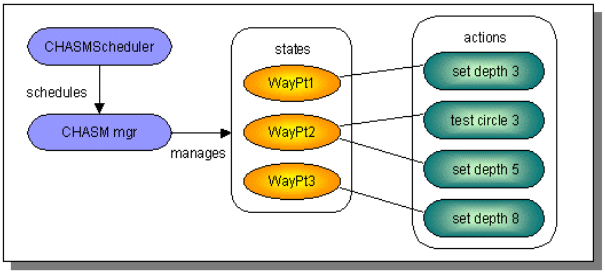

Figure 5: Action Management using the command Pattern

Combining the Command Pattern with the Flyweight pattern.

However, some problems comes up with this solution in our application. The drawback of such a design is its cost. We may end up with hundreds of Action objects, which will consume lots of memory and may incur unacceptable run-time overhead. As shown in Figure 5, actions can only differ by a single parameter (set depth 5 and set depth 8) but they have a lot in common (same implicit natures, resources to acquire operations). We don't want to define two separate classes for these two actions. We would like to parameterize it with this argument, which is the very light non-common data between these two objects. This is the key idea of the Flyweight pattern. We therefore combine the Flyweight pattern to the Command pattern. A Flyweight is a shared object that can be used in multiple contexts simultaneously. It acts as an independent object in each context. Flyweights cannot make assumptions about the context in which they operate. The key concept here is the distinction between intrinsic and extrinsic state.

Intrinsic state is stored in the flyweight. It consist of information that's independent of the

flyweight context, thereby making it shareable. Extrinsic state depends on and varies with the flyweight context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it. In our example, the application can create an object for each behavior, i.e. for the 'set depth' behavior. Each flyweight stores a behavior, but the parameter for this behavior (the context) is different each time the behavior is used, and is therefore stored or determined from a different object.

However, in our problem the State object still does not have to know what it fires, so we end up with 3 objects: the *State* object, the *Action* object and a new one called *Action-Handler*. The *Action* object is triggered by the *State* at run-time, but this time it only contains the extrinsic state of the behavior, and a reference to the Action-Handler. The *Action-Handler* is the object that captures the intrinsic state of the Action. The *Action* object is therefore now a representation of the real action, which is undertaken by the *Action-Handle*r. An *Action* object is now more like an action request. Instead of having what was represented in Figure 5, we now have what is represented in Figure 6. The flyweight representing a particular behavior does not need to store its parameters.

The *Action* object supplies this context-dependant information. Contrary to the Flyweight pattern we don't end up with a number of objects far less than in the previous approach because we store extrinsic states in separate objects. However, these objects are small: it's just some light weight objects. The number of heavy weight object (those containing the code and most of the data, resources, etc. to know how to perform the operation) is far less. Consequently, our approach is scalable. We can have longer missions for the AUV: it will just add light-weight objects, which is not costly. This approach allows us to parameterize an object by an action to perform, but also allows us to parameterize the action to perform in itself.

Another consequence introduced by the Flyweight pattern is that it introduces run-time costs associated with transferring the extrinsic state. It adds one level of indirection. This cost is however not significant if the Action objects have a direct reference to the Action Handler. As actions are decoupled from the Action-Handlers, another consequence is that the action request is independent of its processing. There is a decoupling between the CHASM toolkit and the request it triggers and also a decoupling between the request and its actual implementation.

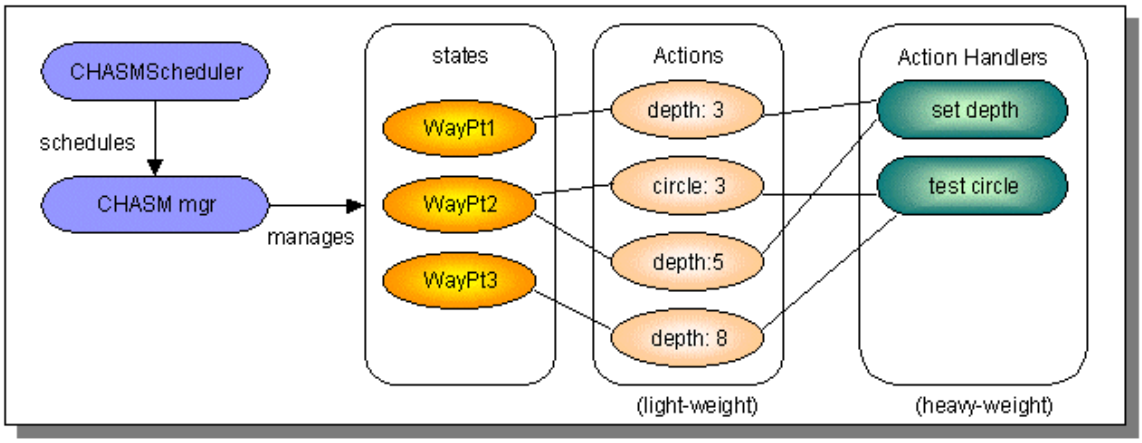

Figure 6: Combination of the Flyweight Pattern with the State Pattern

Action Handlers management.

Moreover, it is quite likely that all possible Action-Handlers won't be used during the mission, so we don't need to create all the Action Handlers. We just want to instantiate them at the beginning of the mission, if they are used. To do this, we can, like in the Flyweight pattern, use a *Factory* that creates and manages the flyweight object. Also, it ensures that flyweight are shared properly. When a client request an Action Handler, the Flyweight factory supplies an existing instance or creates one if none exist. In our example, the client is the Loader: when it must create a 'set depth 10' action, it creates a light-weight Action object containing the depth 10, and stores with it a reference to the 'set

depth' behavior. Consequently, this reference has to be requested first.

Dynamic Reconfiguration and Upgradability

Registration of new Action-Handlers.

In the previous approach, the Action Handler is bound with one or more Action objects that share it. If we want to add a Action Handler, we have to modify the program and recompile everything. We would like something truly modular, where it is possible to add new states, new action handlers on the fly and independently. We want to be able to reconfigure the objects dynamically

Participants

• *State* objects act as invokers: they have a list of actions that are to be fired at some points in the application. This list of actions is represented with an aggregation.

• *Action* objects are requests representing commands to be processed. Each *Action* object encapsulates its name and some parameters to be passed to the *Action-Handler* to perform the request. These parameters are the extrinsic state of the *Action-Handler*s.

• *Action-Handler* specifies an interface consisting of hook methods for the Action objects and for the Factory object. These methods represent the set of operations available

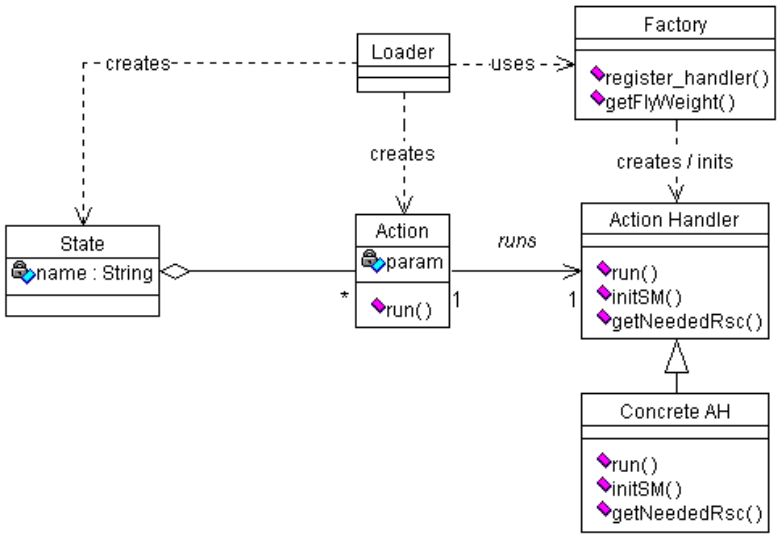

Figure 7: Class Diagram of the Command - Flyweight patterns combination

and to add new services, without recompiling anything.

To dynamically add Action-Handlers to the application, we can use the Factory object introduced in the last paragraph. When a new Action-Handler has to be added dynamically, it is implemented as a separate module, which has to be placed in a specific directory. The idea here is that it has to register itself with the factory. The factory then can instantiate and initialize it. The key concept here is to define an interface for the Action Handler, on one hand to allow the factory to initialize the Action Handler, and on the other hand, to allow the Action object to execute them. The resulting class diagram is given in figure 7.

to process application specific Action requests that may occur.

• *Concrete Action-Handlers* specialize the Action Handler and implement a specific operation that the application offers. Each *Concrete Action-Handler* is associated with some commands that identify the operation to perform within the application. In particular, *concrete Action-Handlers* implement the hook methods responsible for processing the requests. Application developers are thus responsible for implementing the concrete Action-Handlers. These concrete Action-Handlers can be registered dynamically just by importing new modules at run-time. Concrete Action-Handlers then register themselves with the Factory.

• The *Factory* object creates and maintains Action Handlers. It ensures that the Action-

Handler objects are shared properly. When the Loader requests an Action Handler, the AH-Factory object supplies an existing instance or creates one, if none exist.

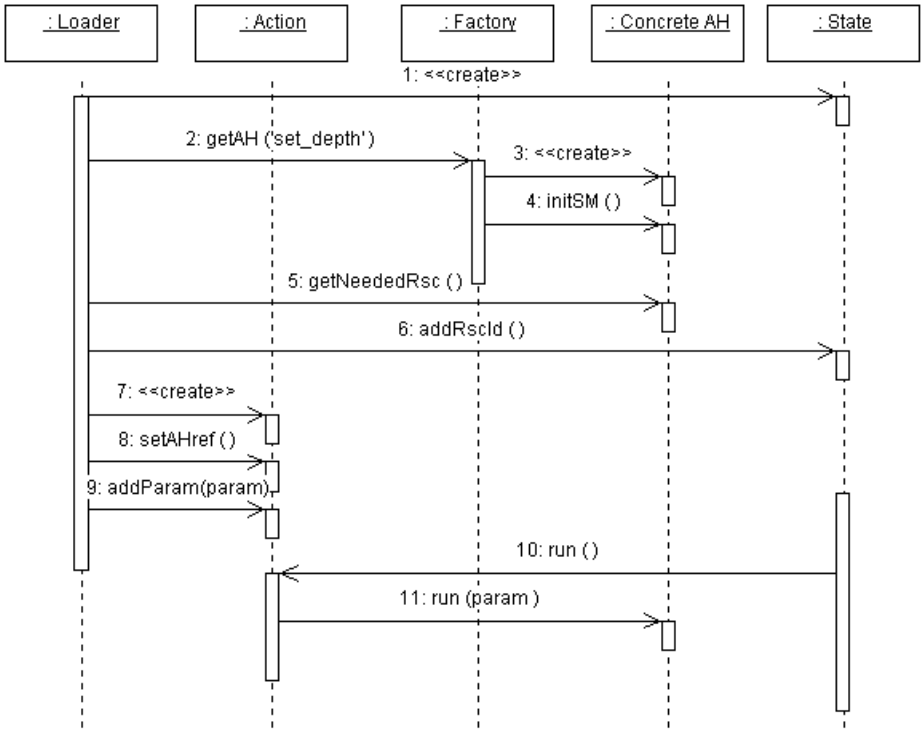

Figure 8: Sequence Diagram for the Initialization and the Execution of an Action

Implementation of the Factory using the Singleton pattern

Because we only need one AH-Factory in our system, to ensure the proper sharing of Action-Handler, we chose to implement the AH-Factory using the Singleton pattern [8]. Moreover, as in Python, classes, functions are first class objects, it is possible to register new Action Handler objects or new Action Handler classes. When the Loader requests an Action Handler, the AH Factory looks if there already is an existing instance of this AH object, then checks if it has a reference to this AH class, and create an instance of it if necessary. The registered Action Handler just has to provide the appropriate interface so that it can be used by the factory and by the actions handlers.

UML Sequence diagram

The sequence diagram for the execution of an Action Handler is given in figure 8. There are two phases. The first one is the initialization, when the loader creates and initializes the objects of the CHASM architecture. The needed shared memory variables for this Action-Handler must be loaded in shared memory (initSM), and we must store in the State object the resources the Action-Handler needs to own in order to run. We also pass to the Action object, a reference to the Action-Handler instantiated by the Factory. The second phase is at run-time, when the action is triggered. The State object fires the Action, which passes the extrinsic data of the operation to the Action-Handler and fires it.

Trade-off flexibility / speed

Although this design allows us to dynamically add some new Action-Handlers, it does not allow us to remove any of them, or to replace them. As a matter of fact, each Action object has a reference to its Action Handler. To solve this problem a solution would be to remove this reference and to add one more level of indirection between the Action and the Action-Handler object. The Action would not reference the Action-Handler explicitly but would have a key: each Action object would encapsulate the name of the Action-Handler to trigger. We would have an additional object, a ‘command

evaluator' which has to dispatch the request to the appropriate Action Handler with respect to the key. The 'command evaluator' is a kind of repository to store all the (key, Action- Handler reference) pairs, but this repository can now be changed at run-time if we decide to replace an Action Handler with another one, we just have to modify a pair. This approach however is costly: we have to do a look-up in this repository at run-time each time a action is triggered. Moreover, this feature is not essential to the AUV application: we are far more interested in the possibility of adding Action-Handlers at run-time than being able to remove or replace some of them.

triggered, it fires the associated test Action Handler passing it the parameters for the test and this one performs the test. The result is a boolean which is stored in the Action Handler object.
Step 2 of evaluating whether a transition should be made is to evaluate the transition condition. The transition condition is a boolean expression which can use combinations of several criteria. The Transition objects of our system must therefore have a mean to obtain the result of each criteria which appear in the Transition Condition. They must therefore have a reference to the appropriate Test Action Handlers. This is illustrated in figure 9. This design is very flexible and scalable because it allows to

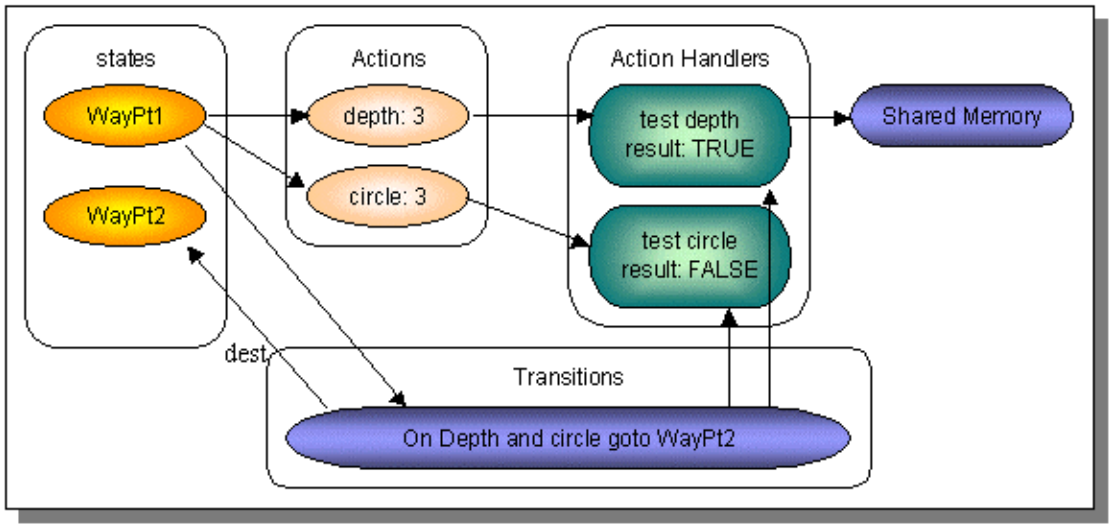

Figure 9: Object Diagram of the CHASM transition design

Transition Design.

Evaluating a transition in the CHASM formalism is done in two states.
Step 1 of evaluating whether a transition should be made consists in defining some test Actions and running them. They define some criteria and test some variables in shared memory to see if they satisfy these criteria. The Test Action objects contain the extrinsic data for the test, i.e. the criteria value, and the tolerance value, and the Test Action Handler is the object which performs the actual test and knows how to access shared memory to do it. There are a lot of different criteria that can be tested: pitch, altitude, etc. There are therefore as many Test Action Handler subclasses as there are criteria which can be tested. When a test Action is

decouple every functionality of the CHASM. Transition conditions can be composed of as many criteria to test as wanted.

Class Diagram of the CHASM data structure.

Taking now into account the design done for the analysis of the CHASM formalism, we end up with the class diagram presented in Figure 10. Are not shown on this figure: the CHASM management, details of the Loader, and the access to the Shared Memory, which are not treated in this paper. Only the main methods and instance variables are shown, not to overload the figure.

The CHASM scheduler

Design of the CHASM scheduler

The CHASMScheduler is the object used to schedule the CHASM manager. This latter one is triggered at regular interval when the CHASM is running. However, the CHASM unit can receive some commands from the super CHASM unit like 'pause', 'resume', 'stop', 'run'. The CHASMScheduler object has therefore to take this order in account before triggering the CHASM manager. The CHASM manager will be run at regular intervals if and only if the mode of the CHASM is 'running', in other words, if and only if the 'start' command was received. The CHASM unit can be in several mode: 'running', 'paused', 'initializing', 'idle', 'resuming'. As we deal with hierarchies of CHASMs, a CHASM is itself a state machine. When a CHASM unit receives requests from the super-CHASM, it responds differently depending on its current state. For example, the effect of the 'start' request depends on whether the CHASMScheduler is in its 'idle' state or its 'running' state. This is where the State pattern is useful.

The State Pattern describes how the CHASMScheduler can exhibit different behavior in each state. The key Idea in this pattern is to introduce an abstract class called CHASMState to represent the states of the CHASM. The

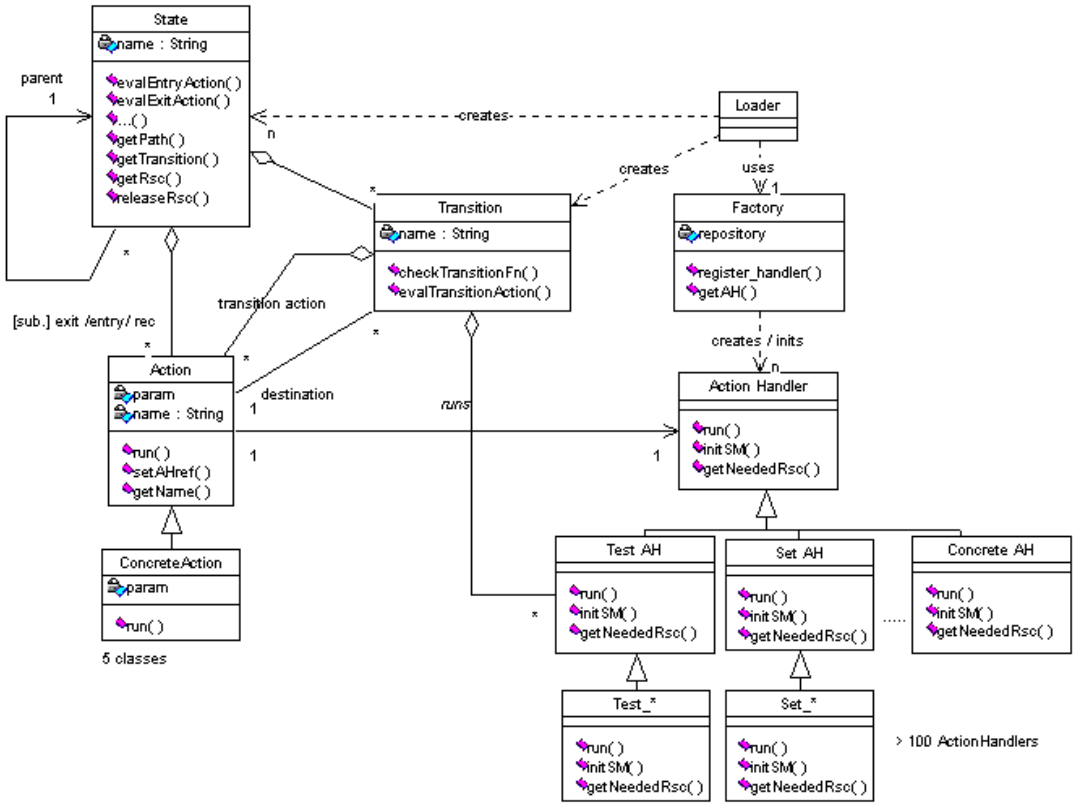

Figure 10: class Diagram of the Design of the CHASM data structure

CHASMState class declares an interface common to all classes that represent different operational states. Subclasses of the CHASMState implement state-specific behavior. For example, the classes CHASMIdle and CHASMRunning implement behavior particular to the Idle and Running states of the CHASMScheduler (see Figure 11).

The class *CHASMScheduler* maintains a state object (an instance of a subclass of *CHASMStat*e) that represents the current state of the CHASM. The class *CHASMScheduler* delegates all specific requests to this state object. *CHASMScheduler* uses its *CHASMState* subclass instance to perform operations particular to the state of the CHASM. Whenever the super-CHASM issues an order to change the state, the *CHASMScheduler* changes the *CHASMState*

object it uses. When the CHASM goes from 'idle' to 'run', for example, *CHASMScheduler* will replace its *CHASMIdle* instance with a *CHASMRunning* instance. The State pattern puts all behavior associated with a particular state into one object. Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses. It avoids to use large monolithic conditional statements in which case adding a new state could require changing several operations, which complicates maintenance. The State pattern avoids this problem but might introduce another because the pattern distributes behavior for different states across several State subclasses. This increases the number of classes and is less compact than a single class. Moreover, speed is also an issue. The system is slower when a subclass is replaced by another one, but this happens when the controller is started or stopped, i.e. once a while, so it is not a problem for our system.

Another consequence of the State pattern is that it makes state transitions explicit. When an object defines its current state solely in terms of internal values, its state transitions have no explicit representation. They only show up as assignments to some variables. Introducing separate objects for different states makes the transitions more explicit.

their successor state and when to make the transition. This requires adding an interface to the *CHASMScheduler* that lets *CHASMState* objects set the CHASMScheduler's current state explicitly. Decentralizing the transition logic in this way makes it easy to modify or extend the logic by defining new CHASMState subclasses. A disadvantage of decentralization is that one CHASMState subclass will have knowledge of at least one other, which introduces implementation dependencies between subclasses.

A common implementation trade-off worth considering is whether to create CHASMState objects only when they are needed and destroy them thereafter versus creating them ahead of time and never destroying them. In our system, CHASMState object do not store a lot of information and do not have some data of their own, so creating CHASMState objects that won't be used is not an overhead. Moreover, there are only five of them. Instantiation costs are paid once up-front (at init-time) and there are no destruction costs at all.

A solution is to make each CHASMState subclass a Singleton, using the Singleton pattern [8], and that each one uses the module namespace to reference to each other. In this way, we just use the module namespace to store all the different instances of the CHASMState objects.

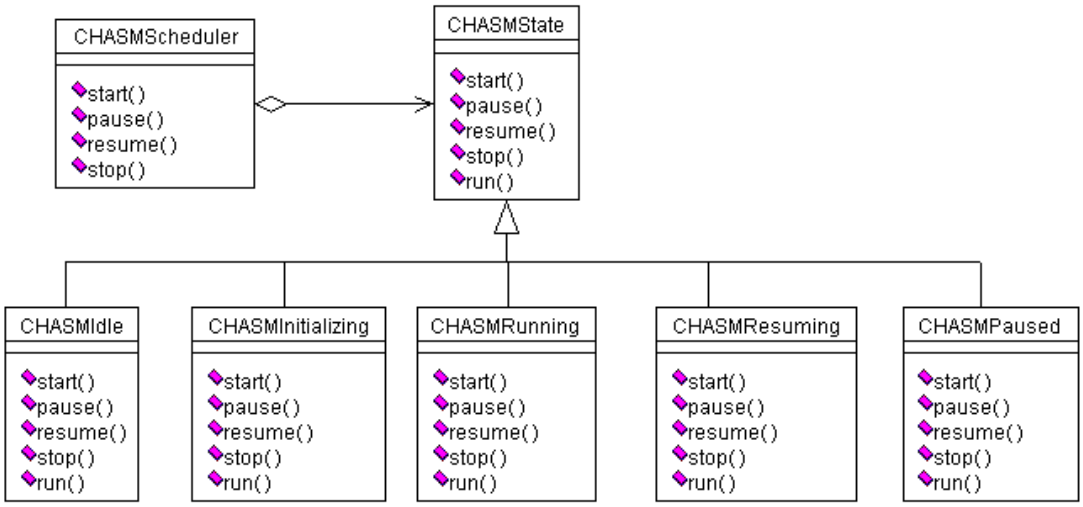

Figure 11: CHASM Meta Management using the State Pattern

Implementation of the CHASMScheduler

The question is now to know which class defines the criteria for state transitions. Only the *CHASMState* subclasses themselves can specify

References

[1] A. R. Girard, "A Convenient State Machine Formalism for High Level Control of Autonomous Underwater Vehicle", Thesis,

Florida Atlantic University, Boca Raton, May 1998.
[2] B. F. Dupire, "Pattern-Oriented Design of a Dynamically Reconfigurable Software Architecture for AUV High-Level Control", Thesis, Florida Atlantic University, Boca Raton, August 2001.
[3] H.Lin and S.Hsu, "Development and Simulation of Vertical Profiling Capability for FAU Autonomous Underwater Vehicle", Thesus, Florida Atlantic University, Boca Raton, Mars 1998.
[3] G. Van Rossum, F.L. Drake Jr., "Python Reference Manuel", Release 2.1, April 15, 2001.
"OEX Software Manual", May 1998, version 2.0, Florida Atlantic University, Advanced Marine System
[4] K. Ganesan, S. M. Smith, K. White, T. Flanigan, "A pragmatic Software Architecture for UUVs", IEEE symposium on Autonomous Underwater Technology, 1996.
[5] Girard, A.R., Smith, S.M., Ganesan K., "A Convenient Format for Discrete Event Control of Autonomous Underwater Vehicles", IARP Workshop on AUVs for shallow water and Coastal Environments, February 17-19, 1998, Lafayette, LA.
[6] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "Pattern Oriented Sotware Architecture: A system of patterns", 1996, Wiley & Sons, ISBN 0-471-95869-7
[7] D. Schmidt, M. Stal, H. Rohnert, F. Bushmann, "Pattern-Orientted Software Architecture, volume 2: Patterns for concurrent and networked Objects", 2000, Wiley & Sons, ISBN 0-471-60695-2.
[8] E. Gamma, R. Helm, R. Johnson, J.Vlissides, "Design Patterns", 1995, Addison-Wesley, ISBN 0-201-63361-2.
[9] P. Dhaussy, J. Champeau, R. Moitie and A. Pringent, "Object-oriented and Formal Methods for AUV development", proceedings of the MTS/IEEE Oceans 2000 conference, volume 1, pp73-78, September 11-14, 2000, Providence, Rhode Island.
[10] Y. Brave and M. Heymann, "Control of Discrete Event systems Modeled as Hierarchical State Machines", IEEE Transactions on Automatic Control, V38, N12, Dec. 1993, pp1803-1819.
[11] B. F. Dupire, A. Folleco, S. M. Smith, "A Python implementation for the High Level Control of an Autonomous Underwater Vehicle", proceedings of the 9th Python Conference, Long Beach, California, March 2001.
[12] J. Champeau, P. Dhaussy, and L. Latreille (ENSIETA, Brest, France), "Mission Control with the UML and SDL formalisms", proceedings of the MTS/IEEE Oceans 2000 conference, volume 3, pp 1639-1645, September 11-14, 2000, Providence, Rhode Island.
[13] D. B. Stewart, "Software Components for Real Time", Embedded Systems Programming Magazine, vol.13, December 2000, pp 100-138.
M. Steenstrup, M.A. Arbib, E.G. Manes, "Port Automata and the Algebra of Concurrent Processes", Journal of Computer and System Sciences, v.27, n.1, August 1983, pp 29-50.
[14] D.B. Stewart and G.A Arora, "Dynamically Reconfigurable Embedded Software, Does it make sense?", proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS' 96), Montreal, Canda, pp. 217- 220, October 1996.
[15] D. B. Stewart, R. A. Volpe, and P. K. Koshla, "Design of dynamically Reconfigurable real-time software using port-based objects", IEEE Transaction of Software Engineering, v.23, n.12, Dec.1997.
[16] Brian Foote and Joseph W. Yoder, "Evolution, Architecture and Metamorphosis", second Conference on Patterns Languages of Programs (PloP ' 95), Monticello, Illinois, September 1995, Pattern Languages of Program Design 2, edited by John M. Vlissides, James O. Coplien, and Norman L. Kerth, Addison-Wesley, 1996
[17] Smith, S.M., "Modular UUV Architectiures: Structure, Control and Power: Issues and Experiences". SOC UUV Showcase, Sept. 24-25, 1997, Southampton, UK.
[18] Smith, S.M., Dunn, R.T., "A new mini modular AUV", UT 98, 15-17 April, Tokyo, Japan.