Mission Specification in Underwater Robotics

Enrique Fernández-Perdomo, Jorge Cabrera-Gómez,
Antonio C. Domínguez-Brito and Daniel Hernández-Sosa

***Abstract*—This paper describes the utilization of software design patterns and plan-based mission specification in the definition of AUVs missions. Within this approach, a mission is described in terms of a set of task-oriented plans in order to simplify mission definition and favor reutilization of some aspects of a mission. Each plan organizes how and when basic tasks like measurement sampling, navigation or communication are to be carried out. The usage of design patterns for AUVs has been considered in order to ease system architecture design.**

***Index Terms*—Software Engineering, control architecture, underwater robotics, mission, framework.**

I. Introduction

DURING the last ten years, the adaptation of Software Engineering principles and methodologies to robotics has caught a lot attention in this field [8]. Nowadays, it is widely accepted that the final quality of a sensory-motor system, its cost of development and implementation, and its ease of usage, are highly conditioned by the software engineering methodologies and tools utilized. In this sense, concepts like design-for-reuse or the utilization of design patterns [10], components and programming frameworks [8] are now routinely used in the development of complex robotic systems.

Autonomous underwater vehicles (AUV) are valuable and highly sophisticated robotic devices and their programming is a complex task that demands an important effort from all the engineers, programmers and scientists involved. The complexity of these systems increases as new mission scenarios are proposed requiring —for example— larger periods of autonomy or coordination among multiple vehicles.

A key aspect of AUVs programming is that concerned with mission specification and control [7]. In particular, special attention must be paid to the number of intervening actors involved in the specification and development of a AUV mission: from system software architects and device integrators to scientists acting as final users of the vehicle.

Mission plans are to be written by scientists, who —probably— will not be experts in robotics, so it is important that a mission can be easily expressed in terms of basic activities of navigation (go to a point, achieve a depth, ...), measurement (take a measure with certain instrument), logging data, and communication. Due to the same reason, mission plans should be analyzed and validated automatically beforehand. Their expression must be concise and favor the reuse and adaptation of basic mission plans to new contexts. In particular, a clear separation between the software that is in charge of the AUV's control and navigation and the code that contains the mission specification is highly desirable [1]. In-situ changes of mission plans should be possible.

As commented previously, AUVs must exhibit large periods of true autonomy so they must be programmed to be reliable. The definition of procedures to detect anomalies or possible malfunction, and to recover from unexpected errors or exceptions is obliged. Most of the exception handling will be integrated within the vehicle control system and should be reusable between missions, but it should also be possible to redefine the treatment of certain failures in the context of a specific mission.

In this paper, the main elements of a software framework for programming AUVs will be presented. The section II will be devoted to present an approach for mission specification based on a set of plans, each of them conceived to cover the different facets of AUV's activities (communications, navigation, measurement, etc.). In section III, we describe the usage of design patterns within a component-based framework for building the software control architecture. Final sections include a review of other related works and a final summary.

II. Mission Specification

A mission is defined as the set of tasks a vehicle must perform. From most common AUV tasks analysis, a basic and proper mission specification framework can be outlined. It is possible then to evaluate qualitatively the features of a particular mission specification design depending on a series of parameters such as modularity, flexibility, monitoring, ease of definition and others, which are key features through mission life cycle stages.

A. Typical AUV tasks

In general, typical tasks included in almost every AUV mission can be orthogonally enumerated as:

1) *Measure sampling:* Sensory equipment is managed to sample different measures, which may be saved or transferred to surface station.
2) *Path following:* Given a set of waypoints, the vehicle tries to reach them in sequence. Therefore, it follows the path obeying some motion constraints.
3) *Area exploration:* The vehicle moves inside an area describing a particular pattern —e.g. *zig-zag*, concentric, spiral, etc.— to explore it.

All authors are members of the University Institute SIANI (Intelligent Systems and Numeric Applications in Engineering — Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería) at University of Las Palmas de Gran Canaria (ULPGC). E-mail: { efernandez, jcabrera, adominguez, dhernandez}@iusiani.ulpgc.es

This work has been partially supported by the following research projects: Project *PI2007/039* funded by the Autonomous Government of Canary Islands (Gobierno de Canarias — Consejería de Educación, Cultura y Deportes, Spain) with FEDER funding; and Project *TIN2008-06068* funded by the Ministerio de Ciencia e Investigación, Gobierno de España.

4) *Measure tracking:* The vehicle senses a particular measure and tries to track it using a predefined behavior.
5) *Communication:* Two main communication tasks are noticed:
   a) *Data/Mission transferring:* Send or receive single or multiple measure samples or internal system observable variables.
   b) *Coordination:* A network of vehicles communicate among them to achieve common goals.

Other vehicles like buoys and ships —e.g. ASVs (Autonomous Surface Vehicles)— may take advantage of some of these tasks, since AUVs capabilities cover most oceanographic missions. Additionally, these elementary tasks might be combined frequently.

B. Mission life cycle

The mission life cycle is composed of the different stages that take place during mission specification, from creation to execution finalization in the vehicle. Diagram in Fig. 1 represents the typical tasks involved in sequence.

1) *Creation:* Process of mission creation or edition. It is common to have a planning GUI (Graphical User Interface) to aid in the specification concerns. Independently of the availability of such a planning tool, an internal textual representation is mandatory to accomplish the next stages of the mission life cycle. Common representation approaches are DSLs (Domain Specific Languages) and Petri Nets coded with languages like Prolog or Lisp [1], [19], [20], [7].
2) *Validation:* It is possible to apply a validation process on a mission completely defined. This will usually be an automatic and transparent process embedded in a verification tool, which will manage the mission textual representation to detect errors or incompatibilities within mission elements.
   Depending on the mission design and representation, the validation techniques vary —e.g. Petri Nets formalism allows network correctness verification [16], [5], [7], a DSL designed with XML (eXtensible Markup Language) can be automatically validated if XML Schemata are provided, etc. Additionally, further semantic checks can be applied.
3) *Transference:* The former stages take place on a computing environment outside the vehicle. Once validated, the mission is transferred to the vehicle, which will save it. This is not trivial tough, because verbose mission representations may consume excessive bandwidth and power. Furthermore, mission modifications may be sent while mission execution as part of replanning.
4) *Execution:* Once the mission is saved in the vehicle, execution will begin after the starting order has been received. In general, we can identify three internal stages:
   a) *Load:* When vehicle's system is booted, mission must be loaded into memory using a data model that describes the mission in order to be interpreted and managed —e.g. Petri Nets are interpreted directly, DSL representations must be parsed and used to build a data model, etc.
   b) *Interpretation:* System is configured according with mission specification while it is interpreted. If any error or exception occurs, the system will log it and try to act accordingly.
   c) *Finalization:* Mission is ended when all specifications have been carried out. A deconfiguration process leaves the vehicle idle, i.e. releases resources, turns devices off, etc.
5) *Replay/Simulation:* Though not compulsory, a simulator may load information logged while mission execution and replay or recreate what happened, allowing offline mission analysis. Predicted or approximate environment conditions provided, simulation can be done before real mission too, allowing beforehand non-predictable incompatibilities detection.

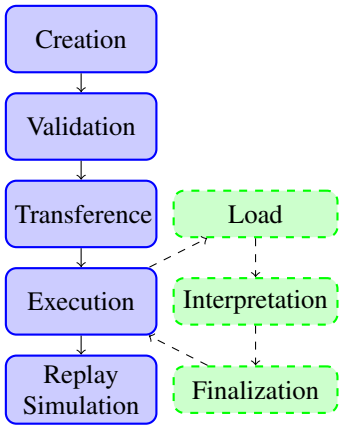


Figure 1: Mission life cycle stages

Despite mission specification modularity, there may still appear dependencies, so the validation stage is necessary to detect inconsistencies or incompatibilities. An incompatibility is an impediment to accomplish a task specified in the mission, usually caused by dependencies within mission elements, available vehicle equipment or vehicle's embedded system state. There are two types of incompatibilities:

1) *Predictable:* Incompatibility can be detected analyzing mission specification. A validation engine integrated in a planning tool usually achieves this task.
2) *Non-Predictable:* Incompatibility cannot be detected with a validation process. This kind of incompatibility arises during mission execution. Vehicle's embedded system is responsible for detecting them and acting accordingly —e.g. replanning, task priorization, exception throw.

C. Plan-based missions

As Fig. 2 shows, mission is split in several plans minimizing dependencies between them and enabling plan exchange among different missions. Anyway, integrity checks between plans can be tackled with a mission validation process.

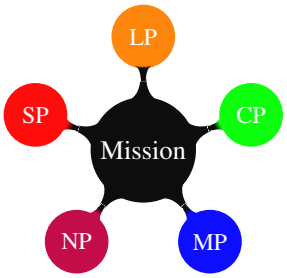


Figure 2: Mission Plans

Furthermore, it is desirable to have a high level mission specification independent of AUV control software.

A plan can be defined as a block that specifies a set of similar and related tasks from a particular field. Following plans are proposed to cover common AUV missions:

1) *Logging Plan (LP):* Logging system or *Black Box* configuration that indicates data logging tasks and stock management. Common *log-able* data are measures and system observable variables.
2) *Navigation Plan (NP):* Details navigation tasks that state where the vehicle must go and forbidden areas. Typical navigation missions are allowed, such as path following, area exploration and measure tracking, by means of actions with maneuver semantics.
3) *Communication Plan (CP):* Contains communication tasks, which are limited by vehicle's communication equipment. Mission validation allows premature detection of incompatibilities between CP and available equipment or other plans —e.g. NP may constraint communication equipment coverage and bandwidth.
4) *Measurement Plan (MP):* Specifies measurement tasks regarding data provided by available vehicle's equipment, with optional sensor selection.
5) *Supervision Plan (SP):* Declares tasks to carry out in case of execution fault. It fires any kind of action through commands or contingency plans.

Plan-based missions are basically inspired on sub-goals and tasks mission specifications, but additional features are considered:

1) Plans gather tasks from a particular typology. Modular specification allows plan reuse and exchange.
2) Plan-based missions far from just provide modular specifications, also bring modular interpretation and management. Vehicle's system architecture may profit from this fact —e.g. each plan interpretation may be accomplished by a dedicated subsystem. Therefore, besides control from mission specification separation, plans also allow system to manage each plan separately in different subsystems that may act like internal agents modeled as software components that coordinate to perform the mission, resolve conflicts and understand a common control language.
3) XML is used as mission specification language, with the following benefits:
   a) Human readable, with semantic close to domain represented.
   b) Great variety of software tools. Programming time is reduced and specification, validation and interpretation portability is increased —e.g. parsers, XML validators, etc.
   c) Language formal syntax is encapsulated within XML Schemata that allow mission validation.
4) Configurable parameters aid mission elements modification during execution. XPath query language is employed to specify which mission elements alter. Mission parameters are simply XPath queries shortcuts, hence actually any mission element is accessible directly.
5) GUI tools complement mission life cycle management, specially by means of assisted creation and automatic validation. This relaxes vehicle's system from further checks during mission load and interpretation.

Plan-based mission specification is analyzed according with the following criteria:

1) *Modularity:* Plans are mission specification modules, internally described in terms of elementary mission tasks. Depending on the plan, tasks may be taken in sequence or parallelly —e.g. NP paths and areas are usually specified on a sequential basis.
2) *Flexibility:* Tasks are designed to cover all typical AUV missions. Mission configuration parameters allow dynamic mission modifications.
3) *Monitoring:* Task level monitoring supported by sub-goals and tasks mission specification is extended to plans.
4) *Ease of definition:* Mission definition is textually represented using XML. Although XML facilitates mission definition, the use of assisted GUI tools are encouraged for end users. Furthermore, tasks abstraction level is close to AUV mission semantic.
5) *Portability:* Each mission plan has an XML Schema to take advantage of validation and interpretation tools, which are portable and widely used.
6) *Reconfiguration:* Mission parameters are the basic support for mission reconfiguration. XPath is used to access mission elements directly or through configuration parameters names.

D. Plan specification

Every plan is made up of a task list, each task with name and unique identifier within the plan. Task specification contains the elements below:

1) *Triggers list:* Specifies the conditions that activate the task. Triggers are predicates on measures or system variables. When all triggers are activated a notification signal is generated, forcing execution of all actions. If empty, actions will have to run continuously.
2) *Actions list:* Specifies name and parameters of commands to execute under system supervision and monitoring while triggers are activated. Actions are system primitives or *reactive skills* that represent atomic or elemental capabilities as in RAP system [9].
3) *Inhibition period:* Inhibits trigger's state checking to ensure task execution is kept at least for a given time,

even if triggers get immediately deactivated. Triggers are updated asynchronously and sample-driven. Once triggers get activated they are only checked periodically though, until they get deactivated again.

Three types of triggers are proposed:

1) *Condition:* Comparison between a particular measure sample and a given value applying a relational operator. If comparison result is true, it is triggered.
2) *Interval:* Temporal interval or range of values built from a given tuple of initial, increment/period and final values. If a particular measure sample is inside the interval, it is triggered.
3) *Exception:* System exception name. If such exception is thrown, it is triggered.

Being predicates, triggers may be combined logically. Triggers list is logic-and combined, while logic-or can be expressed using several tasks with the same actions list. Therefore, descriptive specification instead of procedural or sequential is basically supported by means of trigger-driven tasks. List. 1 shows an example where two tasks are defined to indicate turbidity sampling above 100m deep, and salinity and temperature continuously.

```
<mp id="1" name="offshore salinity-turbidity">
  <task id="1" name="surface turbidity">
    <condition measure="depth" operator="le"
               value="100" unit="m"/>
    <sample measure="turbidity">
      <frequency value="1" unit="Hz"/>
      <resolution value="0.1" unit="NTU"/>
    </sample>
    <inhibition value="1" unit="min"/>
  </task>
  <task id="2" name="salinity profile">
    <sample measure="salinity">
      <frequency value="0.1" unit="Hz"/>
      <resolution value="0.01" unit="PSU"/>
    </sample>
    <sample measure="temperature">
      <frequency value="0.1" unit="Hz"/>
      <resolution value="0.1" unit="C"/>
    </sample>
  </task>
</mp>
```

Listing 1: Measurement Plan example

List. 2 shows a basic NP example that contains a forbidden area and three navigation tasks: initial path following, area exploration and final path following. This plan specification may be generated automatically by a planning tool that allows a graphical specification like depicted in Fig. 3.

```
<mp id="1" name="bay exploration">
  <task id="1" name="go to bay">
    <waypoint id="1">
      <position x="9" y="17" z="0" unit="km"/>
    </waypoint>
    <waypoint id="2">
      <position x="10.5" y="16.5" z="0" unit="km"/>
      <aptitude roll="0" pitch="-5" yaw="90"
                unit="degree"/>
    </waypoint>
    <transect id="1" start="1" end="2">
      <speed x="5" y="0" z="0" unit="m*s^-1"/>
    </transect>
  </task>
  <task id="2" name="bay exploration">
    <area id="1" name="bay">
      <depth min="0" max="150" unit="m"/>
      <time value="2" unit="h" margin="10">
      <path id="1">
        <transects mode="zigzag" amount="10">
          <time value="10" unit="min"/>
          <depth value="10" unit="m"/>
          <angle value="60" unit="degree"/>
        </transects>
      </path>
    </area>
  </task>
  <task id="3" name="come back port">
    <waypoint id="1">
      <position x="9.5" y="14.8" z="0" unit="km"/>
      <uncertainty value="1" unit="m"/>
    </waypoint>
  </task>
  <forbidden id="4"/>
</mp>
```

Listing 2: Navigation Plan example

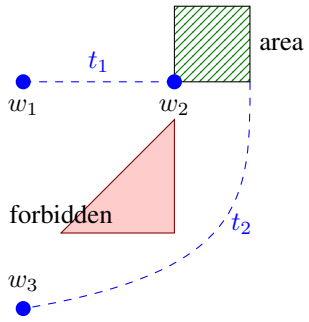


Figure 3: Navigation Plan example

Vehicle's system is configured according with all task triggers of all plans. Measures used by triggers are provided by measurement actions internally and automatically requested. This allows trigger evaluation and signaling to execute task actions in a trigger-driven basis. Fig. 4 shows how *Plan Interpreter* and *Trigger Dispatcher* configure the system (- - -) to enable task activation signaling (——).

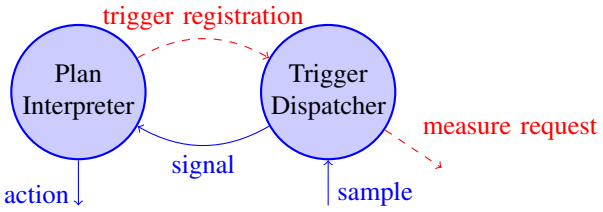


Figure 4: Trigger-driven task issue

Action execution is actually a system configuration process, while action finalization consists in reverting it, i.e. task's actions semantic is activation. Additionally, tasks inside a single plan are only allowed to use a particular set of actions, which are those related with plan semantics. On action failure basic ignore, retry or abort behavior may be selected for each possible action exception. More elaborated exception handlers may be specified with tasks contained in the *Supervision Plan*, which is responsible for mission state monitoring, exception resolution and coordination of system elements.

III. Architectural Design Patterns

In order to design a consistent and integrated system, the underlying software technology supporting the execution

of plans and tasks has to be selected accordingly to high-level objectives. Our election combines the use of a generic component-based programming framework with design patterns for robotic software. On one side, the framework contributes to reduce the programming effort promoting modular and robust reusable code. On the other side, design patterns contribute to improve software quality.

A. Frameworks and Design Patterns

For developing the architecture we are introducing in this document we will take an approach already used in other areas of robotics, where some programming frameworks have come out in order to provide solutions to some of the recurrent problems faced when building control software for robotic systems. Those frameworks have blossomed in many areas, from service and edutainment robotics to space applications [13], [23], [11], [17]. A good detailed survey can be found in [8]. In particular, we will make use of a component-oriented programming framework termed *CoolBOT* [6] developed at our lab in the last years. This framework allows building systems by integrating "off-the-shelf" software components following a port automata model [25] that fosters controllability and observability.

Moreover, some design patterns have been identified as quite useful in order to apply them in robotic software control designs [3]. Here, we will consider design patterns as they are proposed in [10]. Thus, a design pattern can be seen as a design solution for a specific problem which can be applied wherever and whenever that solution is valid considering its prerequisites, requirements, and the consequences at system design level of its utilization. Concretely, we will use three basic design patterns adapted from [3]:

- a *Estimation Pattern*,
- a *Control Pattern*,
- and a combination of the former two, a *Reactive Closed-Loop Control Pattern*.

In next paragraphs we will explain those patterns in more detail in order to reinterpret them in terms of *CoolBOT* components.

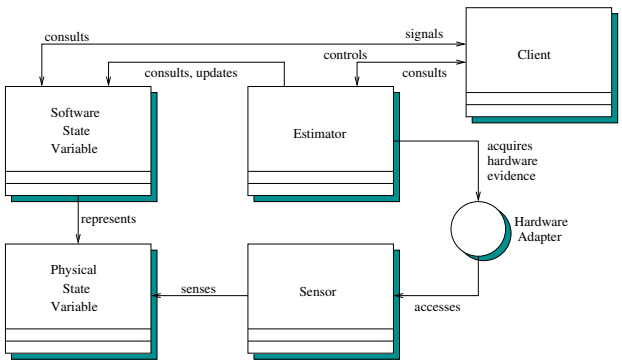


Figure 5: Estimator Pattern.

In Fig. 5 we can observe the architectural structure of the *Estimator Pattern* using a UML object-oriented graphical notation. The distinct entities taking part in the pattern have been represented as objects, independently if they are physically real entities, or just software entities of the control system. This pattern is used for abstracting physical state variables from their counterparts in the control system, establishing a clear separation between the control system and the system under control. Usually, a physical state variable has associated an *Estimator* for estimating the software state variable which represents it in the control system. The *Client* represents an entity of higher level of abstraction which makes use of the pattern. As we can observe, sensory hardware is represented by a *Sensor* object which is accessed through a *Hardware Adapter* which normalizes its interface.

Fig. 6 shows the UML class diagram of the *Control Pattern*. This pattern puts also into practice the principle of separation aforementioned between the control system and the system under control. In this case, the *Controller* makes use of a software state variable in order to actuate and exert actions to control an aspect on the real system, a physical state variable which is its counterpart in the physical system. Similarly to the previous pattern, the *Client* object represents an entity of a higher abstraction level in the control system that makes use of the pattern.

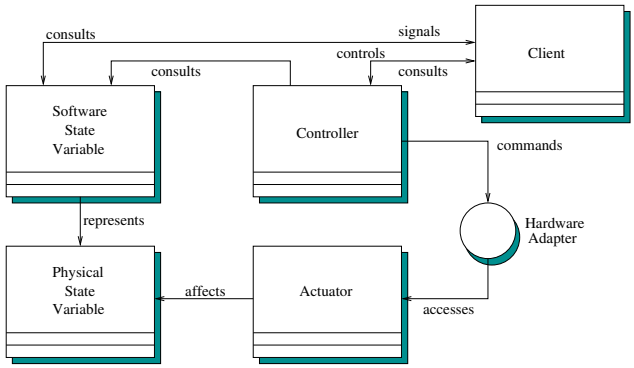
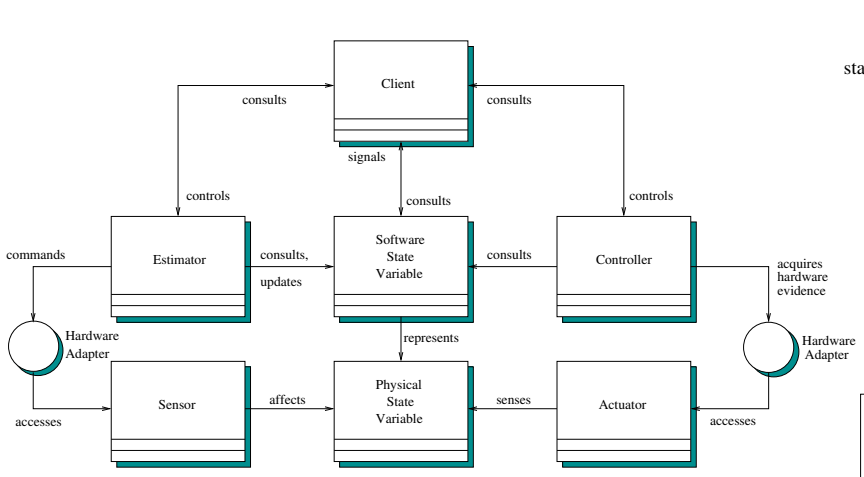


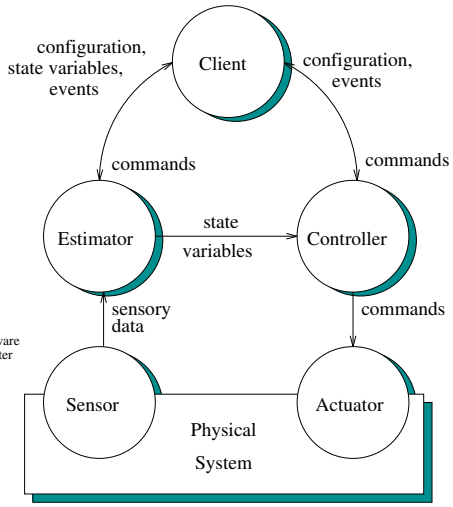
Figure 6: Control Pattern.

The graphical representation of the *Reactive Closed-Loop Control Pattern* is depicted in Fig. 7(a). This pattern is a combination of the previous two, and as a consequence, it is also based on the same principle of control and system separation. Analogously to the previous patterns, there is a *Client* which uses the pattern, and equally, accessing sensors and actuators is normalized using interfaces represented as *Hardware Adapters*. Take into account that in a real control system these patterns are applied indistinctly for any combination of physical-software state variables which are estimated and/or controlled, existing also the possibility of interleaving different state variables in the same reactive closed-loop control pattern. Moreover, many control loops can be operating at the same time in a given moment during system execution.

What is meaningful now for our discussion is how to reinterpret these very well established design patterns in terms of the abstractions and resources available in the software framework we have chosen for implementing our system. In Fig. 7(b) we can observe a representation of the *Reactive Closed-Loop Control Pattern* reinterpreted as a *network* of *CoolBOT* components, where we can distinguish clearly the same entities



(a) Reactive Closed-Loop Control Pattern.



(b) Reactive Closed-Loop Control Pattern in *CoolBOT*.

Figure 7: Reactive Closed-Loop Control Pattern. The design pattern in (a). Translation into *CoolBOT* in (b).

which appears in its corresponding UML representation of Fig. 7(a). With the translation of this design pattern we have tried to map the pattern into *CoolBOT* abstractions, which are usually more complex entities than the objects we frequently find in UML graphical notations representing design patterns. Also notice that entities pertaining to the physical system has not been represented in the figure, contrarily to the UML representations we have used for introducing the design patterns.

In *CoolBOT* a system can be seem as a *network* of *data flow machines* (components) interconnected by data paths (port connections), as we can observe in Fig. 7(b). Port connections among components indicate how data flows within the system, and in this case, how data flows in the pattern, where we can distinguish clearly the control loops present in a system. Notice also that, at each port connection end, and depending on the port connection typology [6], there are some memory buffers storing the information with is under publication on each component. Thus, for example, the *Estimator* component in the figure publishes the software state variables it estimates through its output connections. This explains why in the figure there is no component representing software state variables. It is important also to take into account that UML representations of design patterns, as usually depicted, are mainly static representations of modular designs. In contrast, a configuration of *CoolBOT* components also gives information about the dynamic behavior of the system, because each component is implicitly an *active* data flow machine having its own flow of execution, at least a *thread* in the underlying operating system.

It is important to emphasized here that the design pattern, expressed as a *network* of *CoolBOT* components in Fig. 7(b), is a simplified representation in order to clarify its design. In a real system this pattern may be instantiated multiple times involving multiple instances indistinctly of the different components that integrate the pattern, and where several clients may be sharing and using the services of distinct patterns. Thus, we can outline a system architecture similar to the one shown in Fig. 8 where the lower level, the *Functional Layer*, is mainly composed by multiple reactive closed-loop control patterns which may be indistinctly active/inactive along system's lifetime. Components in higher levels of abstraction make use of the services provided by the system lower level in order to carry out their own services. The intermediate level, *Executive Layer* is mainly integrated by *event dispatcher* and *plan interpreter* components. The last level, the *Supervisor Layer*, is composed by several supervisors that monitor aspects like exception management, system performance, teleoperation, reconfiguration, etc.

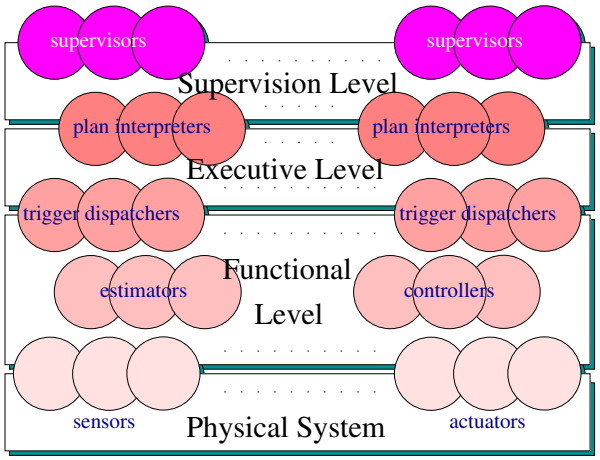
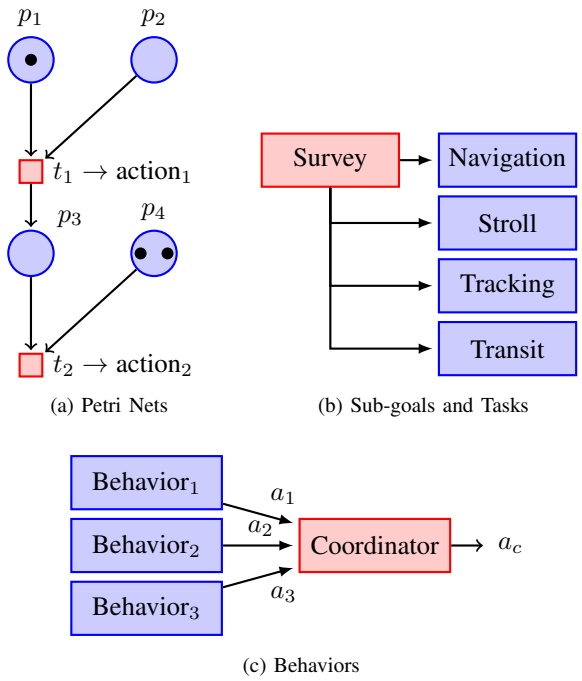


Figure 8: Layered Architecture.

IV. Related Works

MOST research papers describing AUV systems have focused on vehicle's control architectures, and there have been few remarks on how to structure mission specification. However, mission specification architecture is a fundamental aspect, since it constraints the mission spectrum the vehicle can perform and it influences other aspects like robustness, ease of definition, modularity, etc.



(a) Petri Nets (b) Sub-goals and Tasks

(c) Behaviors

Figure 9: Mission specification designs

In Fig. 9 some common mission specification designs are depicted: (a) Petri nets, used in ProCoSa [1] and CORAL [19], [20], [7] systems, (b) tasks, used in ITOCA [21] and others [22], and (c) behaviors [4], [22]. Their main features are summarized below:

1) *Petri Nets:* Petri Net formalism brings robustness and reliability. Its powerful representation allows flexibility and monitoring by means of net marking. Modularity is achieved employing modular Petri nets [7]. Petri net's graphical representation and available building and simulation tools provide portability and ease definition. Furthermore, this representation is suitable for direct interpretation onboard. Their main drawback is the imposed formalism and tools, which need some adaptation to AUV mission domain too.
2) *Tasks:* Sub-goals and tasks mission specification impose DSL development and management, reducing portability. Modularity and monitoring are provided only at task level, while reconfiguration is possible through task configuration parameters.
3) *Behaviors:* Missions specified with behaviors are similar to tasks based ones, but with some important differences. Due to behavior coordination, monitoring becomes difficult, specially if cooperative coordination is needed. It is common to embed learning mechanisms, which in fact allow mission auto-reconfiguration.

Tasks are comparatively easier to specify and monitored than behaviors, so the former is preferable for mission specification. Indeed, many mission specifications are based on tasks [24], [22], [21]. Despite the lack of GUI tools to define missions, translate them into DSL or interpret them, their development time is short. The specification with Petri nets comes with software tools, but they have to be adapted to AUV mission specification. It is also common to find robotic systems that manage behaviors and learning algorithms [4], but their mission specification power is weaker.

Planners are usually embedded in autonomous robotic systems to allow runtime decision making when replanning under uncertainty or failure is demanded. To a large extent, tasks are easily integrated within planners, e.g. in RAP system a task is described by a Reactive Action Package (RAP) which is a context sensitive program specifying a variety of plans for achieving the task in different situations [9]. Most modern AUV architectures as T-REX (Teleo-Reactive EXecutive) goal-oriented system are based on this approach [14], [15]. Embedded automated planning and adaptive execution are T-REX key features, supported by a Constraint-based Temporal Planning approach based on EUROPA$_2$ planning and scheduling solver [12], [2]. Mission specification may support some sort of planner parametrization to allow behavior selection under certain circumstances, by means of action's exception handlers.

We will describe now in more detail some systems that are more closely related to our proposal, as far as they are concerned with mission specification.

A. T-REX (McGaan @ MBARI)

T-REX (Teleo-Reactive EXecutive) [14] is a goal-oriented system, with embedded automated planning and adaptive execution using agents. It encapsulates the operation of a sense-deliberate-act cycle in what is typically considered a hybrid architecture where sensing, planning and execution are interleaved. In order to make embedded planning scalable the system enables the scope of deliberation to be partitioned functionally and temporally inside units called Teleo-Reactors. This discretization tries to guarantee that the current state of the agent is kept consistent and complete during execution. The agent-state is represented as a set of timelines, which capture the evolution of a system state-variable over time in discrete tick units. A timeline is a sequence of tokens that are temporally qualified assertions expressed as a predicate with start and end time bounds defining the temporal scope over which it holds.

Teleo-Reactors in T-REX are characterized by their functional scope, temporal scope and the timing requirements. They are selected for processing agent timelines, according with their functional scope. This process constitutes the basis for inter-reactor communication inside T-REX, using mechanisms such as timeline ownership (internal/external), timeline observation, goal or desired timeline value, and dispatching and notification rules.

A timeline processing algorithm executes as the heart of a T-REX agent at the start of every tick. There are three key steps in the algorithm: 1) all timelines are synchronized at the current execution frontier, 2) new goals are dispatched, and 3) the remaining CPU time can be allocated to reactors for deliberation in incremental steps. Each of these component algorithms operate over the entire set of reactors.

Let's analyze an AUV architecture example proposed by the authors in [15]. There, four Teleo-Reactors are defined to be

responsible for the different control loops operating inside the vehicle: Mission Manager, Navigator, Science Operator and Executive.

The Mission Manager provides high-level directives to satisfy the scientific and operational goals of the mission. Its temporal scope is the whole mission, taking minutes to deliberate if necessary.

The Navigator and Science Operator manage the execution of sub-goals generated by the Mission Manager. The temporal scope for both is in the order of a minute although they differ in their functional scope. Each refines high-level directives into executable commands depending on current system state. The Science Operator is able to provide local directives to the Navigator in case, for example, of detecting something interesting to explore. Deliberation may safely occur at a latency of 1 second for these reactors.

The Executive provides an interface to the underlying AUV functional layer. It encapsulates access to commands and vehicle state variables. The Executive is approximated as having zero latency within the timing model of the application since it will accomplish a goal received with no measurable delay (no deliberation).

B. MOOS (Newman @ MIT)

MOOS [18] refers to a suite of libraries and executables designed and proved to run a field robot in sub-sea and land domains. Included in its scope are a platform-independent communication API, sensor management, state of the art navigation, vehicle dynamic control, concurrent mission task execution, vehicle safety management, mission logging and mission replay.

As far as we know, MOOS is the only programming framework has been used by some groups in programming AUVs. MOOS provides an inter-process communication library. MOOS' processes never communicate directly. Instead, all messages go through a central MOOS server that acts as communication hub, configuring a star-like topology. Every MOOS server together with all processes that communicate through it form a MOOS community and there may be several communities within a system. The communication model proceeds through three actions: *publication*, performed by the process acting as data producer; *subscription*, that must be carried out by the process that will consume some data; and *notification*, issued by the MOOS server to all subscribers whenever a datum is modified.

MOOS provides driver modules to ease the integration of some sensors widely used with AUVs (e.g. GPS, DVL, sonar, altimeter, ...). It also provides modules for navigation, control and data logging, tools for mission replay from log files and communication debugging. It also offers a simple multivehicle simulator.

In MOOS, missions can be defined in terms of tasks. Tasks must belong to certain predefined vehicle's basic capabilities like "GoToWayPoint", "GoToDepth", "LimitDepth" or even to perform some pattern of waypoints. Tasks can be declared and configured statically using a text file. MOOS uses a priority-based scheme to solve arbitration problems among tasks. A mission in MOOS can be specified declaring the sequence of tasks that should be executed and controlled by the mission controller. Missions can be redefined instructing the mission controller to load a different mission file.

Mission plans in MOOS are made up from a set of tasks that synchronize through the exchange of messages. A task can control any number of other tasks sending messages of certain types. In a mission file, tasks are given a unique name, its type is declared and they are configured using the following fields [18]:

- *StartFlags:* A list of messages names that if received will put this task into operation.
- *FinishFlags:* A list of messages that are emitted when the task completes or starves because it is not receiving data.
- *EventFlags:* A list of messages that are emitted when some event happens but the task is not complete.
- *TimeOut:* The maximum time the task should run for. If this timeout expires before the task terminates, Finish-Flags messages are sent and the task terminates. A task can be declared to never timeout.
- *InitialState:* If the task is initially on it does not listen for StartFlags messages. Otherwise, it needs to receive a StartFlags message before it goes active.
- *Priority:* This are used to arbitrate concurrent access to certain resources like actuators. The lower the priority the more important the task is.

Using this combination of message names, priorities and timeout is how a mission is built with MOOS. Functionally, it develops a mission as a network of tasks that are coordinated exchanging typed messages.

V. Summary

In this article we have discussed how a plan-based mission specification may be used to configure a AUV mission. The segmentation of a whole mission into plans simplifies its specification and fosters reutilization. A trigger-based task model for specifying missions abstracts events that allow flexible mission parametrization, modulation, monitoring and control. In comparison with other common mission representations used in AUVs —like Petri nets or behaviors— plan-based missions come up easier to define, as they are closer to AUV mission domain. Moreover, specifying missions using task oriented plans avoids the necessity of onboard planners that might require significant computational costs. In addition, the use of established design patterns for robotic systems fostering separation between the control system and the system under control, have been considered as a key principle in order to ease system architecture design. Furthermore, some of these design patterns have been reinterpreted at design level into a component-based programming framework specifically aimed at developing robotic systems, considering that, in general this kind of frameworks provide already solutions to some of the recurrent problems faced when building robotic control software.

References

[1] Claude Barrouil and Jérôme Lemaire. An integrated navigation system for a long range AUV. *IEEE Oceanic Engineering Society*, (1):1–5, September 1998.

[2] Tania Bedrax-Weiss, Jeremy Frank, Ari Jónsson, and Conor McGann. EUROPA$_2$: Plan Database Services for Planning and Scheduling Applications. November 2004.

[3] Matthew Bennett, Daniel Dvorak, Joseph Hutcherson, Michel Ingham, Robert Rasmussen, and David Wagner. An Architectural Pattern for Goal-Based Control. In *Proceedings of the 2008 IEEE Aerospace Conference*, March 2008.

[4] Marc Carreras Pérez. *A Proposal of a Behavior-based Control Architecture with Reinforcement Learning for an Autonomous Underwater Robot*. PhD thesis, University of Girona, Girona, Spain, May 2003.

[5] Søren Christensen and Laure Petrucci. Modular analysis of Petri Nets. *The Computer Journal*, 43(3), 2000.

[6] Antonio C. Domínguez-Brito, Daniel Hernández-Sosa, José Isern-Gonzalez, and Jorge Cabrera-Gámez. Coolbot: A component model and software infrastructure for robotics. In Davide Brugali, editor, *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*. Springer - Verlag, Berlin/Heidelberg, April 2007.

[7] R. A. Duarte Oliveira. *Supervisão e Controlo da Missão de Veículos Autónomos*. PhD thesis, Universidade Técnica de Lisboa. Instituto Superior Técnico (IST), Lisboa, Portugal, 2003. Dissertação para obtenção do grau de mestre em engenharia electrotécnica e de computadores.

[8] D. Brugali (Editor). *Software Engineering for Experimental Robotics*. Springer Tracts in Advanced Robotics, Volume 30/2007. Springer Berlin/Heidelberg, 2007.

[9] R. James Firby, Roger E. Kahn, Peter N. Prokopowicz, and Michael J. Swain. An Architecture for Vision and Action. In *Fourteenth International Joint Conference on Artificial Intelligence*, pages 72–79, 1995.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[11] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the 11th International Conference on Advanced Robotics (ICAR'03)*, pages 317–323, Coimbra, June 2003.

[12] Ari K. Jónsson, Paul H. Morris, Nicola Muscettola, and Kanna Rajan. Planning in Interplanetary Space: Theory and Practice. Technical report, NASA Ames Research Center, Brekenridge, 2000. AIPS 2000.

[13] Laboratoire d'Analyse et d'Architecture des Systèmes - LAAS (CNRS). LAAS OpenRobots Project. http://softs.laas.fr/openrobots.

[14] Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Rob McEwen. T-REX: A Model-Based Architecture for AUV Control. In *3rd Workshop on Planning and Plan Execution for Real-World Systems 2007*, 2007.

[15] Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Rob McEwen. A Deliberative Architecture for AUV Control. In *International Conference on Robotic and Automation (ICRA) 2008*, 2008.

[16] Tadao Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, 1989.

[17] Issa A. D. Nesnas, Anne Wright, Max Bajracharya, Reid Simmons, Tara Estlin, and Won So Kim. Claraty: An architecture for reusable robotic software. In *in SPIE Aerosense Conference*, 2003.

[18] P. Newman. The MOOS project homepage. [Online ; accessed May 8, 2009].

[19] P. J. C. Ramalho Oliveira, A. Pascoal, V. Silva, and C. Silvestre. Design, development and testing of a mission control system for the marius auv. *Department of Electrical Engineering. Institute for Systems and Robotics*, Instituto Superior Técnico (IST)(1):20, 1996.

[20] P. J. C. Ramalho Oliveira, A. Pascoal, V. Silva, and C. Silvestre. The mission control system of the marius auv: System design, implementation and tests at sea. *International Journal of Systems Science*, 29(10):1065–1080, 1998. Special Issue on Underwater Robotics.

[21] P. Ridao, J. Yuh, J. Batlle, and K. Sugihara. On AUV Control Architecture. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)*, volume 2, pages 855–860, 2005.

[22] G. N. Roberts, R. Sutton, and R. Allen. Guidance and control of underwater vehicles. *Elsevier Science and Technology*, IFAC Proceedings Volumes(1):1–40, 2003.

[23] C. Schlegel. *Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach*. PhD thesis, University of Ulm, 2004.

[24] Reid Simmons and David Apfelbaum. A Task Description Language for Robot Control. In *Proceedings of Conference on Intelligent Robotics and Systems*, pages 1–7, March 1998.

[25] D. B. Stewart, R. A. Volpe, and P. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12):759–776, December 1997.