Model-driven analysis and design for software development of autonomous underwater vehicles

Citation for published version:
https://doi.org/10.1017/S0263574714001027

Digital Object Identifier (DOI):
10.1017/S0263574714001027

Link:
Link to publication record in Heriot-Watt Research Portal

Document Version:
Early version, also known as pre-print

Published In:
Robotica

Publisher Rights Statement:
Copyright Cambridge University Press
This paper has been accepted for publication and will appear in a revised form subject to peer review and/or input from the Journal's editor. Re-use for private research and study only.

General rights
Copyright for the publications made accessible via Heriot-Watt Research Portal is retained by the author(s) and/or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
Heriot-Watt University has made every reasonable effort to ensure that the content in Heriot-Watt Research Portal complies with UK legislation. If you believe that the public display of this file breaches copyright please contact open.access@hw.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Francisco J. Ortiz †*, Carlos C. Insaurralde ‡, Diego Alonso †, Francisco Sánchez †, Yvan R. Petillot ‡

†Universidad Politécnica de Cartagena, División de Sistemas e Ingeniería Electrónica.
Cartagena, E-30202, Spain.

‡Heriot-Watt University, Institute of Sensors, Signals and Systems, Ocean Systems Laboratory,
Edinburgh, United Kingdom.
francisco.ortiz@upct.es

Abstract: Software engineering plays a key role in state-of-the-art robots where more effective and efficient software development solutions are basically needed for implementation and integration of advanced robotics capabilities. Component-based software engineering and model-driven software development are two paradigms suitable to deal with such demand. This paper presents the analysis, design, and implementation of control software for an Autonomous Underwater Vehicle (AUV). The software development stages are carried out using a toolchain based on the two development engineering methodologies mentioned above. A case study of a high-performance AUV control application and experimental results from a software schedulability analysis are presented.

Keywords: model-driven software development; component-oriented software architecture; design analysis; autonomous underwater vehicles.
1. Introduction

Robotics is a multi-engineering discipline in which software plays a key role. The relevance at this role is in great part due to the increasingly-complex robot capabilities that usually entail distributed cross-platform software applications. Software engineering practices facilitate the development of such applications but unfortunately they are not as established as others involved in the development of robotic systems. Nevertheless, the robotics community is aware that specific development solutions are needed not only to simplify implementation and integration but also to improve quality of robotics software.

Recent software developments in robotics show they try to go beyond classical programming techniques (libraries, toolkits, robotic middleware, etc.) by adopting Component-Based Software Engineering (CBSE) paradigms such as encapsulation of proven solutions into reusable building blocks. However, CBSE is not still widely applied in robotics as expected. The lack of support for development of components and their integration into the global robotic system make them difficult to be adopted by researchers and practitioners of robotics. Additionally, the application frameworks that provide run-time support usually require developers of robotic component-based applications to have a deep knowledge of the framework’s interiors. Moreover, most of the current CBSE frameworks do not define a development process that combines capabilities for generation and evolution of software code and evaluation (including early model analysis) of requirements.

Model-Driven Software Development (MDSD), however, has an incremental rate of interest in robotics. This paradigm raises the level of abstraction of the development process since domain experts can express domain concepts by design
models. Thus, the complexity of middleware, frameworks or any other software artifacts is hidden. High-abstraction level models are independent of the implementation technologies so that best practices (including early verifications) can feasibly be applied. MDSD-based toolchains have the ability to deal with such models by automatically transforming them into lower-level other models or even executable code.

The main contribution of this paper is to apply the above software engineering practices to the development process and architecture of control software of unmanned vehicles. In particular, an application for an Autonomous Underwater Vehicle (AUV) is developed to show the benefits of putting into practice such methodological software approaches. This innovation moves away from traditional development techniques by applying poorly exploited practices (CBSE and MDSD) in robotics.

Such a software development is carried out by means of a toolchain called C-Forge\(^1\) which puts the two above development technologies into practice. There is a particular focus on how an MDSD process combined with CBSE facilitates development of AUV applications along with assistance in pre-verification of non-functional requirements. An AUV platform often requires rapid software prototyping in order to deal with specific behaviours provided by the AUV modules (diving, path following, etc.). In addition, the platform software requires support of a hierarchical control structure with concurrent execution of high and low priority tasks, including quick responses to critical events such as collisions. Therefore, C-Forge is suitable to develop the AUV control software, and carry out Real-Time (RT) analyses of a component-oriented AUV architecture. A hierarchical behaviour-based application and a real-time spooling module are developed and studied.

The following sections show the research contributions, technological innovations, and justification of the use of C-Forge. Section 2 presents a background to
support relevant robotics topics of this paper. Section 3 reviews key CBSE and MDSD aspects of current development technologies for robotics. Section 4 discusses details of the MDSD process and the C-Forge architecture. Section 5 describes the AUV case study. Section 6 shows the results obtained from the analysis and design of the AUV control software. The last Section presents conclusions and future research directions.

2. Robotics Software Engineering

This Section presents a three-pillar foundation as a research backbone for robotics software engineering in this paper. Software developments for AUVs do not get away from the requirements discussed below. Therefore, they are also aligned with the technological directions taken by other robotic software systems.

2.1 Software Engineering for Robotic Systems

Best practices of software engineering foster reusable solutions to tackle complexity in robotics software, i.e. modularity, encapsulation, separation of concerns, etc. Modular control architectures facilitate the understanding of the robotics software complexity. The encapsulation of algorithms in library functions hides the computational software complexity from developers. Software designs built over hardware abstraction layers such as those based on Player (robotic middleware) effectively supports reusability of low-level software modules, and ACE (Adaptive Communication Environment) particularly allows robotics software to be distributed by abstracting the communication layer.

Hierarchical modularity is essential to deal with complex software for robotics. However, it has considerable influence on designs of robotic applications where architectural changes mainly impact on: (1) the real-time software performance since the module-to-module latency (in particular at low control level) can be significantly
modified, (2) the distributed behaviour of software components since parallel module processes as well as the interactions between components at high control level, including coordination, competition, cooperation, and collaboration can be affected, and (3) the embedded system configuration since component-based modules are tightly coupled to processing hardware so any modification to the hierarchical system composition can mean behavioural and/or structural drawbacks in the network of software components.

Most of the development processes limit the reuse of robotics software code to re-implementation integration of “glue code”[1], by integrating different libraries but no one of them incorporates domain-specific knowledge and design patterns. However, there are some initiatives such as Best Practice In robotiCS[5] which structures and formalizes the robot development process (including software) by providing developers with tools, models, and functional libraries. Current development approaches such as Component Based Software Engineering (CBSE) deal with reusability and flexibility of software in order to overcome the above limitations by reusing building blocks (software components) to build applications[6].

2.2 Component-Based Software Engineering

A software component is a compositional unit with contractually specified interfaces, and explicit context dependencies. This individually-deployed building block is subject to composition by third parties[7]. A component-oriented architecture entails a repository for software components with well-defined interfaces that reduce coupling, and facilitate the software reusability (components only interact with each other through ports)[8].

---

[1] Glue code does not contribute to the main functionality of the application but join code parts that would otherwise be incompatible when integrating software, e.g. different order of function parameters, conversion between physical units, etc.
A development framework for robotics software components is an implementation skeleton which can be customized to specially develop such architectural elements. This skeletal approach provides designers with more reuse and extension mechanisms than libraries do as well as run-time support for applications. A framework is a semi-complete application that provides a reusable but also common structure to share among applications. Developers incorporate a framework into their application, and extend it to meet their specific needs. Frameworks differ from toolkits by providing a coherent structure instead of a simple set of utility classes.

Robotics software designers (this also happens in other domains) always develop, integrate, and connect components by using object-oriented technologies. It is an important drawback from the above framework since the integration of component-based applications requires abstractions, and tool support that object orientation paradigms cannot cope with (e.g. required interfaces). This makes a clear point to have a development framework (support tool) to design, and implement robotics software components by dealing with them as genuine architecture units instead of object-based block solutions.

There are other limitations from current component-based frameworks as those discussed in previous research works. They probably make CBSE unattractive for robotics. The two major issues to tackle the above adverse situation are as follows. (1) Creation of techniques, tools, and development processes to design applications starting from components as atomic units so that complexity of software artefacts can be hidden from robotic developers. (2) Development of mechanisms to turn component-oriented designs into executable programs, i.e. an “interpreter” or even a “compiler” for a components language. These problems can be overcome by adopting a Model Driven Software Development approach.
2.3 Model-Driven Software Development

Model Driven Software Development (MDSD) allows domain experts to directly express their knowledge by using and reusing models which separate functionality (defined by domain experts) from implementation (defined by software experts by means of model transformations). MDSD provides CBSE with theoretical foundations, and the tools to tackle the two above problems for development of robotics software components. Technically, the combination of MDSD and CBSE is realized by defining meta-models, supporting model transformations, and providing a MDSD toolchain.

**Meta-model definition.** This involves the definition of meta-models to model CBSE concepts (software components). C-Forge includes a meta-model, entitled WCOMM⁹, to model architectonic blocks as white-box components defined by their structures (ports required and provided interfaces), and internal behavior (including hierarchical and orthogonal timed-automata). WCOMM also allows developers to reuse previously modelled components.

**Model transformations.** These mechanisms automate the process of generating robotics software implementations by turning models into other models or executable code. Model transformations included in C-Forge do not generate the full application code, but rather target a component framework, entitled FraCC, which provides the required run-time support for component-based applications.

**MDSD toolchain.** This set of programming tools is to carry out the whole development process. The C-Forge toolchain¹ supports the whole CBSE process, integrating WCOMM, FraCC, and the model transformations.

3. Literature Review
This Section shows an overview of existing methodologies and technologies for robotics software engineering. This state of the art has a strong emphasis on how they contribute to software reuse provided CBSE and MDSD are supported.

3.1 Software Engineering Aspects of Interest

The main aspects studied to evaluate how much current methodologies and technologies support robotics software engineering are as follows.

**Software Reusability (SR).** This aspect considers the degrees by which software blocks (CBSE) can be used and reused by different applications with none or even minimal additional coding work. Robotics software reuse is divided into three following categories.

**Functional Library (FL):** Approaches within this category are software libraries that encapsulate different functionalities. The user code directly invokes algorithms which are normally encoded as linked libraries (static or dynamic libraries). In addition, developers have to provide extra support for the software systems such as glue code (e.g. communication drivers), concurrency, etc.

**Robotics Middleware (RM):** Approaches within this category are solutions that complement the aforementioned category by providing a communication middleware for achieving distribution and modularity. Typically, it involves a publish/subscribe mechanism to exchange messages. Applications are built based on a set of heterogeneous binary software since tools do not modify the internal structure of modules.

**Component-Based Framework (CBF):** Approaches within this category provide similar support as those from robotics middleware as well as they turn modules into components.
**Model Management (MM).** This aspect considers the approach ability to support transformation of models at different abstraction levels as specified by MDSD.

**Performance Analysis (PA).** This aspect considers the approach capability to analyse real-time performance on robotics software models.

### 3.2 Comparison of Methodologies and Technologies

Table I shows how the approaches studied deal with different software engineering aspects for robotics required to have a good support for development such systems.

### 3.3 Discussion

**Software Reusability:** Approaches that support SR offer slightly different solutions, and uses different development paradigms. This makes it difficult a methodical comparison of the above solutions. However, a survey of middleware for robotics\(^{10}\) and a quantitative comparison of different frameworks (estimated source code for driver and algorithm implementation, communication middleware and the robotic software)\(^{11}\) were already presented. A review of the state of the art can also be found in BRICS deliverables.

**Model Management:** Many CBSE aspects are not considered by the robotics community. Nevertheless, BRICS highlights the need of formal models, reuse of designs, and architectural elements independently of the implementation platform and toolchains that support the generation and validation of models and code. This formal definition of component model and meta-model is not covered by component frameworks that use OO as design methodology such as OROCOS, RoboComp, ORCA or the currently widely-used framework; ROS. A research work\(^{10}\) considers the Object Management Group (OMG) specification RTC (Robotics Technology Component) as the most relevant approaches for robotics based on MDSD. It also includes a description of the three implementations of the standard: Gostai RTC\(^{13}\), OpenRTM-aist\(^{14}\), and OPRos\(^{15}\) but concludes that none of them are matured enough.
Performance Analysis: there are some MDSD-based approaches for RT systems such as ArtistDesign\textsuperscript{16} for the embedded systems design, and the OpenEmbeDD project\textsuperscript{17}. Furthermore, the automotive industry domain has standardized AUTOSAR\textsuperscript{18}). The situation is different in the robotics domain. All of the current component models for robotics are not analysable. The component model and the supporting infrastructure are usually required to involve analysable designs. Most importantly, no every extra-functional property has to be analysed but those explicitly considered by the component model and supported by the implementation platform. The main objective is to be able to analyse the performance of the component-based application as early as possible in order to avoid waiting until the implementation code is written.

C-Forge facilitates the reuse of software through a CBF. It also has the ability to support transformations of models since it is MDSD-based approach. One of the transformations leads to models that involves notation of RT requirements. C-Forge facilitates the reuse of software through the use of models and a CBF, where the component is the unit of reuse. The application architecture in C-Forge is separated from its deployment in nodes and processes/threads by means of a deployment model. Therefore, modifications on deployment configurations do not affect the architectural aspects. This toolchain also has the ability to support transformations of models since it is MDSD-based. One of the transformations leads to models that involves notation of RT requirements. This allows developers to carry out pre-verification of timing performance of the component-based application under development. C-Forge includes the Cheddar tool\textsuperscript{19} to carry out such RT analyses (e.g. schedulability).

4. C-Forge Toolchain
This Section presents an overview of architectural aspects of C-Forge with emphasis on the two main elements of the MDSD process of such a toolchain.

4.1 C-Forge Architecture Overview

C-Forge is a MDSD tool-chain to develop component-based software applications. It has two main elements: (1) a modelling language (WCOMM) for component-based applications (at the top of Figure 1), and (2) a C++ framework (FraCC) that facilitates the implementation of WCOMM-modeled applications (at the bottom of Figure 1).

4.2 The WCOMM Component Meta-model

A software component is modelled as a white box with WCOMM. Unlike existing solutions, (RobotML, Smartsoft and BRICS mainly, see comparison in Table I) functionality is encapsulated in components by modelling timed automata with hierarchy and orthogonality. In addition, algorithms (represented as activities) executed in states are not modeled but linked with components in order to get a clear separation of concerns between design (modelling) and implementation. C-Forge transforms models to final code implementation by providing execution support (a framework) instead of generating directly the code.

A component-oriented architecture can be defined with WCOMM (step ‘a’ in Figure 1) based on the requirements for the application under development. A WCOMM component is a software entity that encapsulates its internal state, only communicates with other components through messages sent to as well as received from their ports, and its behaviour is modelled by means of timed automata. Component messages are asynchronous (without response communication scheme). This makes it possible the implementation of any communication scheme as required.

Timed automata are finite state machines, similar to those defined in UML but richer in notation, since it includes temporal properties like clocks and timed guards. It enables
users to model component behaviour by using concepts such as states, transitions, events, guards, as well as hierarchical and orthogonal regions. A region is a behaviour fragment that may be concurrently executed with the rest of the orthogonal regions contained in the same state. They can greatly simplify state modelling by avoiding the common “state explosion” problem that appears when creating medium-size state machines. Timed automata cannot only model the lifecycle of WCOMM components but also the reaction of a component to the arrival of external messages or internal events. This formalism is particularly suitable for modelling timing constraints in reactive systems such as those typically found in robotics.

The WCOMM toolset is composed of three tools: two graphical tools for describing the application architecture as a set of connected components, and for defining the structure (ports and interfaces) and the behaviour (through timed automata) of components, and a textual editor for modelling the application interfaces and datatypes. The modelling of applications is a pure CBSE process at this stage of the development process. Therefore, the modeller does not have to know the implementation details of the framework which ultimately support the platform.

4.3 The FraCC Framework

FraCC efficiently deals with the concurrent and real-time requirements of robotic software systems. The FraCC framework takes into account non-functional aspects such as task scheduling, concurrency management, fault tolerance, and distributed communication that are not considered by existing approaches/frameworks. This enables FraCC to carry out schedulability analyses.

Once the application is modelled in WCOMM, the artifacts needed to run the application in FraCC (deployment model as shown in step ‘c’ of Figure 1), and the C++
templates where the user adds the code (step ‘c’ of Figure 1) are generated. FraCC allows developers to explicitly control the concurrency properties of the application by using a deployment model that is generated by a model-to-model transformation (step ‘b’ in Figure 1) from an input WCOMM model. This transformation hides the FraCC implementation complexity to developers. The generated model enables developers to select how many threads run the WCOMM application, and to assign each of the timed-automata regions to a thread.

A region is defined as the assignment unit of workload to threads instead of a state because only one active state at a given time in a region is allowed. In addition to regular WCOMM regions, FraCC adds a special type of region for managing messages sent between components. This region copies output messages from a sending component to the input ports of receiving components. The user can also assign these kinds of regions to any thread in the deployment model. This feature does not only maintain the balance in the workload assignment to threads but also gives the user full control on the framework execution (every activity executed by the framework is assigned to a thread by the user). Thus, FraCC executes no “hidden code”.

The transparent code execution provided by FraCC makes C-Forge different to other approaches such as Smartsoft, YARP, ROS, BRICS, or OROCOS. Additionally, C-Forge does not force developers to implement a middleware layer if the application is not distributed, nor a one-to-one relation between components and processes. The user, through the deployment model, sets the resources that are really needed. The flexibility provided by the separation between architecture and execution in the deployment model is exploited in this paper (Section 6).

The generated C++ templates facilitate the integration of existing algorithms, and the implementation of each of the component activities (i.e. the realization of
software code) separately from the framework. These activities are then compiled into
dynamic libraries, linked to states in the deployment model, and then dynamically
loaded when the application is executed (step ‘d’ and ‘f’ in Figure 1). However, FraCC
sets some restrictions to the code contained in an activity, e.g. infinite loops are
forbidden, and activities cannot spawn threads or create mutexes.

The above allocation scheme generates a temporal model from the deployment
model by means of an automatic transformation (step ‘e’ and ‘g’ in Figure 1) of the
FraCC model. This resulting model is used to carry out an early verification of RT
requirements by using an analysis tool (Cheddar19). Appropriate changes can be made in
the applications models as needed based on the analysis outcomes. The deployment
model provides C-Forge with flexibility since it can test components with various
deployment configurations, i.e. same architecture; different platforms.

5. Control Software Analysis and Design

This Section presents details of the analysis and design of control software for
AUVs. In particular, the way mission tasks are spooled is studied.

5.1. Case Study

Current AUV missions usually demand rapid software prototyping in order to
quickly evaluate if autonomous capabilities are capable of carrying out the tasks
assigned. They rely on specific behaviours provided by the AUV modules (diving, path
following, etc.). They also require the AUV platform software to support a hierarchical
control structure with concurrent execution of high- and low-priority tasks, including
quick respond to critical events such as collision detection. C-Forge is a good toolchain
suitable to develop the AUV control software since it provides agile development for
the above requirements, i.e. support for developing component-oriented software
architectures, analysing real-time software performance, and generating automatically software code.

The case study mainly focuses on modelling and early verification of a real-time software application for the AUV reactive components and the executive mission spooler. This platform-independent module works in coordination with the AUV mission planner. It is able to parcel out the tasks specified in the mission plan. Tasks are carried out by executing functions from other software modules available (already developed) in the AUV platform. Before dispatching the tasks, the spooler schedules them by following an initial order given by the mission plan but constraining the execution of tasks according to well-defined conditions. The tasks execution is according to the plan generated in the mission planner.

The architectural approach for the module ‘mission spooler’ is based on the SAE JAUS standard\textsuperscript{20} and inspired by the multi-stage pipeline computing technique applied to computer architecture. The performance of the mission spooler to simultaneously deal with different tasks from a mission plan is a critical aspect to be analysed. Therefore, a RT analysis is essential to know whether the above module is able to execute tasks in due time.

5.2 Architecture of the AUV Control Application

The first development step with C-Forge consists on the description of the component-oriented architecture for the AUV. The functional requirements of the AUV Control Application (AUVCA) and the domain knowledge lead to a design of a three-tier architecture (Figure 2) modelled with a graphical tool included in C-Forge. This architectural diagram is a computational model that can be executed and transformed into analysis models, or even to software code. The model is independent of the
platform at this early development stage so it can be reused, and refined to include specific requirements for the implementation platform as needed.

The AUVCA interfaces and data types are defined by using a textual tool along with the graphical one. This step is necessary to describe the interaction points (ports) between components. The next step is to define the internal behavior of the components with timed automata. It also refines the design of messages and common interfaces. These two steps can be carried out as many times as needed until the model is fully described (completed). Therefore, the design process is iterative and incremental.

The component $C_{MissionInterface}$ (top of Figure 2) provides the interaction support between the human operator and the AUV system. At the deliberative level, the component $C_{MissionPlanner}$ interprets the user mission commands in order to build a plan based on sequential or parallel tasks. The component $C_{PathPlanner}$ provides a list of waypoints from origin to goal. This functionality is required by the component $C_{MissionSpooler}$ when positioning the AUV.

The execution layer acts as the interface between the planning layer and the behavioural control. The component $C_{MissionSpooler}$ is responsible for translating high-level plans (list of tasks) into low-level actions by invoking behaviours at the appropriate times, monitoring execution, and handling exceptions.

Four different components are placed in the behavioural layer (bottom of Figure 2). $C_{DataCollector}$ gathers acoustic or visual data from the seafloor. It also filters and processes data from the component $C_{AUVNavigator}$ in order to add position information to the acoustic/visual data and give better information about landmarks and obstacles. The component $C_{ObstacleAvoider}$ embeds two types of behaviours, the obstacle detection and the implementation of strategies of obstacle avoidance as vector-field histogram or potential fields.
At the lowest level, closer to the hardware, the component \textit{C\_AUVNavigator} and the component \textit{C\_AUVPilot} are designed as wrappers of AUV platform modules. The navigation system in the component \textit{C\_AUVNavigator} is based on an extended Kalman filter which aims at estimating the position of the AUV from the measurements of the on-board sensors. A Doppler Velocity Logger (DVL) measures linear and angular velocities as accurately as possible (velocities of the vehicle along the surge, sway and heave axis as well as roll, pitch and yaw). The AUV is also equipped with a gyroscope and a depth sensor used in the correction step. On the other hand, the algorithm embedded in the component \textit{C\_AUVPilot} turns spatial and velocity coordinates (surge, sway, heave, yaw, pitch and roll) into velocity commands to actuators (thrusters). Most of the AUVs capable of hovering control only surge, sway, heave and yaw. They rely on the centre of mass and buoyancy of the AUVs to approximately keep null the pitch and roll.

5.3. Behaviour Modelling of the Mission Spooler

The behaviour of the software components (which structure of components is shown in Figure 2) can be described by a timed-automata tool provided by C-Forge. Only the \textit{C\_MissionSpooler} behaviour (figure 3) is discussed in this Section since it is the most critical element of the AUVCA. This component gets mission commands as inputs from the component \textit{C\_MissionPlanner}, and carries out a concurrent process to decode and execute the tasks from a mission plan. This plan can include a set of concurrent/non-concurrent blocking/non-blocking tasks with hard real-time requirements.

The component \textit{C\_MissionSpooler} contains four concurrent regions (mission spooler functionality which functions can be executed in parallel and real-time). These regions are (figure 3; from left to right): mission command processing, control of mission execution in progress, motion tracking, and monitoring data collection. In the
region \texttt{R\_MissionSpooler} there is a hierarchical state \texttt{(S\_Running)} that is displayed at the bottom of Figure 3. \texttt{S\_Running} is composed of two concurrent regions: (1) the region where each mission task is decoded and dispatched to the components that execute such a region \texttt{(R\_RunningSpool)}, and (2) the region in charge of collecting synchronization events from the components that are performing the tasks submitted by the mission spooler \texttt{(R\_RunningCollect)}.

When \texttt{C\_MissionPlanner} sends a new mission to the component \texttt{C\_MissionSpooler}, the mission is stored as a list of tasks in the mission spooler (including timing and synchronization requirements). The event \texttt{E\_StartSpool} fires the beginning of a cyclic process involving the three main states. Tasks are picked up one by one in the state \texttt{S\_Feching}, decoded in the state \texttt{S\_Decoding}, and finally dispatched or queued in the state \texttt{S\_Dispatching}.

The execution unit of tasks in the mission spooler has a pipeline-based architecture (3-stage pipeline processing: fetching, decoding, and dispatching). For example, a list of tasks in a mission plan could include \texttt{“go to waypoint 1”} followed by \texttt{“record data 1 minute after starting moving”}, and after reaching the first waypoint \texttt{“move to waypoint 2 without collecting data”}. As the AUV heads to the first waypoint (once the task is launched in the state \texttt{S\_Dispatching} of the component \texttt{C\_AUVPilot}), next instruction is decoded. The task \texttt{“record data”} requires no synchronization with the AUV positioning so it is released to the component \texttt{C\_DataCollector} with a time-out requirement. Next task requires the first waypoint to be reached before dispatching the next positioning task; therefore, the timed automata remain in the state \texttt{S\_WaitingSynch}. Concurrently with this process, the region \texttt{Running\_Collect} (in the state \texttt{S\_CollectingEvents}) receives the event of first waypoint reached. A completion event \texttt{E\_Done} is sent to the region \texttt{R\_RunningSpool} in order to dispatch the task \texttt{“go to}
waypoint 2 and stop recording”, and continue decoding if there are more tasks in the list.

The data-collection task is monitored in the region R_DataCollectionMonitor along with the process above described to assure that data are processed every 500 ms. Otherwise, the automaton goes to the state S_Error. The AUV motion is monitored in parallel in the region R_MotionMonitor. For example, the movement to a goal position requires a path planning capability. The mission spooler requests such a path to the AUV path planner which returns a list of intermediate waypoints to be sent to the pilot. If there is an unexpected obstacle that makes the AUV stray too far from the path (the component C_ObstacleAvoider is in the state S_Avoiding), the automaton goes to the state S_ExceptionHandling, and requires the calculation of a new path. If it is impossible to find a path towards the target, the automaton goes to the state S_Aborting. Thus, the state S_Error is reached in the region R_MissionSpooler where an alert report is sent to the operator. Then, she/he can decide either to abort the mission or take manual control to retrieve the AUV from the trapped area. Once the above situation is fixed, an event E_Recovery takes the automaton back in the state S_Monitoring.

The component model is completed by specifying the activity to be undertaken in each state. In fact, only the “shell” of the activity is modelled by means of an interface with pins (Figure 4).

Each pin represents an input/output message or an event generated in such an activity, e.g. Figure 4 shows the component C_MissionInterface which contains three states, two of them include activities. The model is graphically completed by linking the pins, and activity events with interface messages. Input messages in the ports cannot only provide the components with information but also generate an event, e.g. the event misEnd in the port PIn_MisSts.
5.4 Flexible deployment with FraCC of the AUV Control Application

Once finished the WCOMM model of the application, a model-to-model transformation (step ‘b’ in Figure 1) is automatically carried out in C-Forge to obtain the FraCC deployment model (step ‘c’ in Figure 1) with a default configuration. At the same time, a model-to-text transformation (step 'b' in Figure 1) generates: (i) the C++ templates (step 'c' in Figure 1) of the software code corresponding to the "shells" of the activities (including the code to access the input and output messages and events), (ii) the C++ implementation of the messages exchanged by the components, and (iii) the C++ implementation of the timed automata events.

The application developer must add domain-specific software code to the generated templates. There are four methods where the developer can add specific code: init(), onEntry(), onExit(), and doCode(). All these methods are automatically invoked by the FraCC infrastructure when the activity is first loaded (init method), and when the state containing the activity is entered (onEntry method), exited (onExit method) or while it is the current active state (doCode method). If the activity is periodic then the doCode method is periodically executed. If the activity is sporadic the method is invoked only once. Figure 5 shows an excerpt of code for the activity 'ProcessingCmds', including the four methods mentioned above, and the use of some messages (rooted at the AUVCAMsg namespace) and events (rooted at the AUVCAEvents namespace).

The templates filled in with the user code are then compiled as dynamic libraries. These libraries can independently evolve since they are linked to the model (not embedded in it). The model is able to keep the links between activity pins and messages in ports. Thus, the developer can write the activity code independently from the rest of the AUVCA. This code is dynamically loaded when the application is executed (step ‘d’ and ‘f’ in Figure 1) to become FraCC execution processes.
The deployment model with a default configuration includes a single process per node and a single thread per component containing the component regions (Region Activity in Figure 56). The framework adds an additional region to each component in charge of updating periodically the data in ports. This region is automatically included in the deployment (Port Activity in Figure 56). The robotic designer modifies this configuration depending on the AUVCA requirements and design decisions. Initially, the AUVCA is distributed on two computers: (1) a tele-operation station computer which is the operator control unit (including the components C_MissionInterface and C_MissionPlanner without strict RT requirements), and (2) an on-board control computer (AUV) with the rest of components. Therefore, the default deployment is divided into two nodes with one processor in each one, as shown in Figure 56.

Starting from this default deployment, an automatic transformation generates a temporal model (step ‘e’ and ‘g’ in Figure 1) to carry out an early verification of the RT requirements of the AUV on-board components by using Cheddar analysis tool. Table 4 shows the periods of execution (T) and Worst Case Execution Times (C) for each component region (task). Tasks are annotated with a type to specify a periodic execution or respond to sporadic service requests. Sporadic tasks are assigned with their minimum inter-arrival times as period of execution. If a region has several states with periodic activities, the time of the region is assumed as the higher T and C of their different states. The timing properties (basically, T and C) of the regions are calculated by applying equations (1) and (2). The timing properties of the threads are derived from the properties of the regions assigned to such threads by applying equations (3) and (4).

\[
T_{\text{reg}}^i = \gcd(T_{\text{act}}^i), \text{ act}_i \in \text{reg}_i \\
C_{\text{reg}}^i = \max(C_{\text{act}}^i), \text{ act}_i \in \text{reg}_i \\
T_{\text{th}}^i = \gcd(T_{\text{reg}}^i), \text{ reg}_j \in \text{th}_i
\]
\[ C_{th}^i = \sum C_{reg}^j, \text{reg}_j \in \text{th}_i \]  

Starting from the timing requirements shown in Table 4II, obtained from experiments carried out with AUV prototypes, several experiments with the AUV computer components were carried out by testing several distributions of tasks (flexible deployment). These tests are described in the following section, including a discussion of the results.

6. Experimental Results

This Section presents experimental results obtained from the real-time schedulability analysis carried out with C-Forge. It is done on the AUVCA and based in the deployment configuration presented in previous Subsection 5.4.

6.1 Schedulability Analysis

A model-to-text transformation in C-Forge (‘e’ in Figure 1) generates automatically an analysis file for the Cheddar tool to perform the schedulability analysis. Cheddar will simulate it, calculate response times for every thread, and check whether it is schedulable or not. The analysis is performed by assuming a fixed priority with preemption scheduler, rate monotonic assignment of priorities to threads, and immediate ceiling priority protocol for shared resources.

The schedulability analysis allows different deployments of the software architecture to be generated and analysed with Cheddar. It defines a set of tasks to execute the components regions in order to make sure that all timing requirements (stated in Table 4III) are fulfilled at run-time. If the Cheddar-based analysis process concludes it is not schedulable, the deployment is modified by creating new threads and re-assigning regions to different threads. Then, the analysis is performed again to check the new deployment. Table 2-III shows details of the five deployments studied, and the main results of the Cheddar-based analysis.
**Initial deployment.** Figure 5–6 considers a process with six threads, one per component, including the different concurrent regions which characteristics are shown in Table 4II. The result of the Cheddar analysis for the first deployment is shown in Figure 67. The conclusion is that this deployment is not schedulable, as some task deadlines are missed.

A more detailed study of the results of the schedulability analysis shows that the deadlines are not met if two regions, which sum of computation times exceeds the period of the task, are grouped in the same thread. This is the case of the $C_{MissionSpooler}$ since its regions have periods of 100 ms and 150 ms, the thread they are assigned to is given a period of 50 ms (the greatest common divisor of the regions periods), and a worst case execution time of 49 ms (the sum of all regions execution times). Therefore, the rest of the tasks cannot meet their deadlines, and the deployment is not schedulable.

**Second deployment.** To avoid the lack of schedulability of the first deployment, a new assignment of the component regions in two new threads for a total of seven threads is created. Although this second deployment is now schedulable, the creation of more threads and shared resources increases the system overhead.

**Third deployment.** In order to decrease overhead, a third deployment is tested. Regions of the components $C_{ObstacleAvoidance}$ and $C_{DataCollector}$, as well as $C_{Pilot}$ and $C_{Navigator}$ are assigned to the same threads (two in total). This reduces the number of threads to five. The results of the analysis show that this new deployment is schedulable and also reduces system overhead.

**Fourth deployment.** To try to further reduce the number of threads, a fourth deployment is created. Two threads (and the regions assigned to them) are merged in a single one. Unfortunately, the system is not schedulable in this case.
**Fifth deployment.** The last tested deployment is built around the idea of grouping together regions with the same period or which periods are integer multiples. In this case, only three threads are created which periods are 100 ms, 150 ms, and 500 ms. Regions with periods 100 ms and 200 ms are assigned to the first thread, while regions with 150 and 500 ms are assigned to the second and third thread, respectively. The analysis process concludes that this last deployment is also schedulable.

**6.2 Discussion of Results**

The deployment model provides an agile development process by facilitating any modification on the distribution of concurrent regions to threads. New deployment models can be easily generated from existing ones just by creating new threads or re-assigning regions, as discussed before.

A comparison among the resources needed by each of the three schedulable deployments is shown in Figure 89. The figure compares the number of shared resources created in each of the deployment by FraCC as well as the number of context switches and thread preemptions performed by the operating system scheduler. These parameters represent the overhead of the application execution. That is, the time added to the execution of the regions by the deployment decisions. Of all, the number of preemptions is perhaps the worst one, since it is related to the number of times a thread has been interrupted by a higher priority thread without having finished its current execution.

It is also worth discussing why the number of shared resources does not decrease with the number of threads, as happens with preemptions and contexts switches. This is because shared resources are created whenever a region sends or receives messages from regions assigned to a different thread that the former. Highly cohesive deployments, that is, deployments where regions that exchange messages are assigned
to the same thread, tend to have fewer shared resources. This is exactly the case of the first deployment (one thread per component), in which, despite being not schedulable, only half of the shared resources created for the rest of the tested deployments are needed. These data are not shown in Figure 8–9 since the first deployment is not schedulable.

7. Conclusions

A component-oriented control software application for an AUV has been developed by means of an automated software engineering toolchain (C-Forge) which combines the CBSD and MDSD paradigms. A CBSD approach favours the rapid development and reuse of software components whilst a MDSD approach allows domain experts to express their knowledge independently from the implementation platform. C-Forge includes a component framework (FraCC) that provides the required runtime support for component applications. Developers use FraCC to set application deployment, i.e. the number of threads in which the application will be run, their temporal characteristics, and workload. FraCC models can be analysed during the early design stages based on the generation of real-time and concurrency analysis models (Cheddar-based tool). The separation between architecture and deployment enables application developers to generate, analyse and test various deployment scenarios without changing its architecture or losing control over the execution as some existing approaches do.

A case study where RT software execution is required by an AUV has been presented. A particular analysis of RT performance has been carried out on an AUV control application. Five alternative deployment configurations were analysed. The results obtained from schedulability analyses with Cheddar in different tests show that the deployment of concurrent regions in threads can be easily changed by just moving one line in the model. The software process based on C-Forge paves the way for a
reduction of costs and risks in development of AUVCAs. It is a solution for the development of an AUVCA that opens opportunities to be applied to other control applications for autonomous robotics.

Future investigation will include the RT analysis of the execution of tasks from additional AUV capabilities required by other missions. A study on potential combinations of deployment configurations will also be considered in order to analyse resource management in AUV systems. Further work will also be done on C-Forge in order to integrate transformations to other frameworks such as ROS to exploit other features different than real-time. In addition, dynamic application reconfiguration at runtime will be considered, e.g. by means of dynamic wiring of components to permit task-dependent composition of skills to behaviours. A reporting tool will display different deployment alternatives that optimize certain parameters (e.g. number of threads, shared resources, and communication bandwidth).

Acknowledgments

This work has been partially supported by the Region of Murcia's Government Project MISSION-SICUVA (ref. 15374/PI/10) and the Spanish MEC FPU Program (grant AP2009-5083).

Thanks to the European Commission for the financial support. Also thanks to the partners of the TRIDENT Project (grant agreement 248497) for the information provided about the marine vehicles platform to make this research work possible.

References


Figure 1. The C-Forge MDSD process.

Figure 2. Diagram of components of the AUV control application.

Figure 3. Concurrent regions in the component $C_{MissionSpooler}$ (top). The hierarchical state $S_{Running}$ (bottom). Periodic states in bold relief.

Figure 4. The component $C_{MissionInterface}$ with activities shells linked to the in/out messages with the C-Forge tool for simple component definition.

Figure 5. Excerpt of the code for the activity 'ProcessingCmds'.

Figure 56. Example of a deployment model in a localhost node with one process and two threads.

Figure 67. Schedulability analysis for initial deployment configuration.

Figure 78. Schedulability analysis for the second tested deployment.

Figure 89. Comparison among the number of threads, shared resources, number of thread preemptions, and context switches of the three schedulable deployment models.
**Figure 1:** The C-Forge MDSD process
Figure 2. Diagram of components of the AUV control application

Figure 3. Concurrent regions in the component C_MissionSpooler (top). The hierarchical state S_Running (bottom). Periodic states in bold relief.
Figure 4. The component C_MissionInterface with activities *shells* linked to the in/out messages with the C-Forge tool for simple component definition.

```cpp
#include "A_ProcessingCmds.h"
#include "../userInterface.h"
    // defines ‘reportUser’ and ‘userCmd’

void A_ProcessingCmds::init(){ ... }
void A_ProcessingCmds::onEntry(){ ... }

void A_ProcessingCmds::doCode(){
    AUVCAMsg::MisEnd  inMsg_misEnd;
    AUVCAMsg::SetPlan outMsg_setPlan;
    AUVCAMsg::Abort  outMsg_abort;
    [...]  
    AUVCAMEvents::E_NewPlan  E_NewPlan;

    // *** USER CODE HERE ***
    if(get_E_NewPlan(E_NewPlan)){
        set_set_plan(outMsg_setPlan);
        reportUser(E_NewPlan);
    }
    if(userCmd(cmd)="abort")
        set_abort(outMsg_abort);
    [...]  
    // *** END USER CODE ***
}

void A_ProcessingCmds::onExit(){ ... }
```

Figure 5. Excerpt of the code for the activity 'ProcessingCmds'.
Figure 5. Example of a deployment model in a localhost node with one process and two threads.
Figure 6. Screen-shot of Cheddar schedulability analysis for the initial deployment, concluding that it is not schedulable. A graphical simulation of the tasks scheduling (top) and the results of the response time calculations (bottom) is displayed.
Figure 7. Screen-shot of the results of Cheddar schedulability analysis for the second deployment, concluding that it seems to be schedulable.

Figure 8. Comparison among the number of threads, shared resources, number of thread preemptions, and context switches of the three schedulable deployment models.
Table I. Contribution to Robotics Software Engineering Aspects from Current Approaches.

Table II. Timing requirements of the components on board the control PC in the AUV. In bold regions with hard-RT requirements. Deadlines not displayed since \( D_i = C_i \).

Table III. Summary of the tested deployment configurations and Cheddar analysis results (\( P = \) task priority; \( T = \) task period; \( C = \) task execution time; \( RT = \) task response time, calculated by Cheddar).
<table>
<thead>
<tr>
<th>Approach</th>
<th>Highlights</th>
<th>SR</th>
<th>FL</th>
<th>RM</th>
<th>CBF</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRPT</td>
<td>Mobile Robot Programming Toolkit provides an extensive set of libraries that cover the most common data structures and algorithms employed in robotics (SLAM, motion planning, etc.)</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>OROCOS KDL and BFL</td>
<td>Libraries developed in the context of the OROCOS project providing a Kinematics and Dynamics Library (KDL) for kinematics chain and Bayesian Filtering Library (BFL) for performing inference in Dynamic Bayesian Networks.</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>ROS</td>
<td>Robot Operating System provides libraries and tools to develop distributed robotics software applications with the ROS middleware.</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>YARP</td>
<td>Yet Another Robot Platform is an open-source framework that supports distributed computation with a main focus on robot control and efficiency.</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>ORCA</td>
<td>Open-source framework for developing component-based robotic non-RT systems. ORCA components can be pieced together to form arbitrarily complex robotic systems, from single vehicles to distributed sensor networks.</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>RoboComp</td>
<td>Component-oriented robotics framework based on Ice middleware providing a skeleton of components and a set of tools to generate and test components.</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>OROCOS</td>
<td>Provides a toolchain that includes a Component Library (OCL). OCL uses the OROCOS Real Time Toolkit (RTT). RTT provides a C++ framework to create robotics applications using modular, run-time configurable software components.</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>RT-MAPS</td>
<td>Commercial component-based framework for rapid prototyping of real-time applications.</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>OpenRTM</td>
<td>Component object model in robot technology middleware for robot system integration, CORBA based implementation of RTC specification.</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>OPRoS</td>
<td>Open Platform for Robotic Services is a component based open source platform. It has Integration Development Environment (IDE) tools, a robot framework for robot operation, a server, and a test and verification tool.</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>BRICS</td>
<td>The BRICS project aims at exploiting MDSD as enabling approach to reducing the development effort of engineering robotic systems by making best-practice robotics solutions more easily reusable. It provides a tool-chain that includes a CBSE meta-model, and a repository of harmonized algorithms.</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>RobotML</td>
<td>The PROTEUS project proposes an ontology and a Domain Specific Modeling Language for robotics, RobotML. It includes a Model Driven toolchain, automated code generation, simulation and robot targeting.</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>SmartSoft</td>
<td>SmartSoft has many similarities to C-Forge, since it models components as architectural units that are later on translated to the target platform, which is implemented as a framework that provides the required run-time support.</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>EasyLab</td>
<td>EasyLab is a model-based development tool for modelling, simulation, code generation and debugging with a primary focus on mechatronics. It uses MDSD to model robot systems at different levels of abstraction.</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>AADL</td>
<td>Architecture Analysis &amp; Description Language is perfectly suitable for high-integrity control systems since it allows deploying software models on different computer resources for reconﬁgurable distributed control applications.</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>MAST</td>
<td>MAST is a tool aligned with MARTE to specify and analyze RT systems. It also provides developers with a sensitivity analysis facility that informs the user of how far or close the system is to meet its time requirements.</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Uppaal</td>
<td>Tool for component-based modelling, simulation, and verification of real-time and embedded systems modelled as real-time components.</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Cheddar</td>
<td>Cheddar is a tool for checking task temporal constraints of a real time application/system. Systems to analyse can be described with AADL or with the Cheddar architecture design language.</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>C-Forge</td>
<td>C-Forge is a model-driven tool-chain that relies on WCOMI, a modelling language for component-based applications, and FrACC, a framework that provides the runtime for implementation of applications, including flexible deployment and real-time pre-verification.</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
</tbody>
</table>

Table key: ● fully-supported, ○ partly-supported, ○ non-supported.
Table II. Timing requirements of the components on board the control PC in the AUV. In bold regions with hard-RT requirements. Deadlines not displayed since $D_i = TC_i$.

<table>
<thead>
<tr>
<th>Component // Region</th>
<th>Type</th>
<th>$T$ (ms)</th>
<th>$C$ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C AUVPilot</td>
<td>R</td>
<td>Pilot</td>
<td>50</td>
</tr>
<tr>
<td>C AUVAutoNavigator</td>
<td>R</td>
<td>Navigator</td>
<td>50</td>
</tr>
<tr>
<td>C ObstacleAvoider</td>
<td>R</td>
<td>Detecting</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>Avoiding</td>
<td>80</td>
</tr>
<tr>
<td>C DataCollector</td>
<td>R</td>
<td>CollectingData</td>
<td>100</td>
</tr>
<tr>
<td>C PathPlanner</td>
<td>Sporadictic</td>
<td>1000</td>
<td>40</td>
</tr>
</tbody>
</table>

Table III. Summary of the tested deployment configurations and Cheddar analysis results ($P$=task priority; $T$=task period; $C$=task execution time; $RT$=task response time, calculated by Cheddar).

<table>
<thead>
<tr>
<th>Thread (tasks)</th>
<th>$P$</th>
<th>$T$</th>
<th>$C$</th>
<th>$RT$</th>
<th>Sch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Deployment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>th_DataCol</td>
<td>2</td>
<td>150</td>
<td>12</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>th_MS1</td>
<td>6</td>
<td>50</td>
<td>49</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>th_Nav</td>
<td>4</td>
<td>100</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>th_OA</td>
<td>3</td>
<td>150</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>th_Path</td>
<td>1</td>
<td>500</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>th_Pilot</td>
<td>5</td>
<td>100</td>
<td>6</td>
<td>1150</td>
<td></td>
</tr>
<tr>
<td>2nd Deployment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>th_DataCol</td>
<td>2</td>
<td>150</td>
<td>12</td>
<td>85</td>
<td>Y</td>
</tr>
<tr>
<td>th_MS1</td>
<td>5</td>
<td>100</td>
<td>34</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>th_MS2</td>
<td>3</td>
<td>150</td>
<td>15</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>th_Nav</td>
<td>6</td>
<td>100</td>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>th_OA</td>
<td>4</td>
<td>150</td>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>th_Path</td>
<td>1</td>
<td>500</td>
<td>10</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>th_Pilot</td>
<td>7</td>
<td>100</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3rd Deployment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>th_DataCol</td>
<td>2</td>
<td>150</td>
<td>22</td>
<td>85</td>
<td>Y</td>
</tr>
<tr>
<td>th_MS1</td>
<td>4</td>
<td>100</td>
<td>34</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>th_MS2</td>
<td>3</td>
<td>150</td>
<td>15</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>th_Nav, th_OA</td>
<td>5</td>
<td>100</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>th_Pilot</td>
<td>1</td>
<td>500</td>
<td>10</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>4th Deployment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>th_ReactiveLayer</td>
<td>4</td>
<td>150</td>
<td>34</td>
<td>36</td>
<td>N</td>
</tr>
<tr>
<td>th_MS1</td>
<td>3</td>
<td>100</td>
<td>34</td>
<td>196</td>
<td></td>
</tr>
<tr>
<td>th_MS2</td>
<td>2</td>
<td>150</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5th Deployment</td>
<td>7th Path</td>
<td>1</td>
<td>500</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>---</td>
<td>-----</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>7th Path</td>
<td>1</td>
<td>500</td>
<td>42</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>7th MS1_an</td>
<td>3</td>
<td>100</td>
<td>46</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>7th MS2_an</td>
<td>2</td>
<td>150</td>
<td>37</td>
<td>85</td>
<td></td>
</tr>
</tbody>
</table>

Y