Robotica (2015) volume 33, pp. 1731–1750. © Cambridge University Press 2014
doi:10.1017/S0263574714001027

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

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

†*Universidad Politécnica de Cartagena, División de Sistemas e Ingeniería Electrónica. Cartagena, E-30202, Spain*
‡*Institute of Sensors, Signals and Systems, Ocean Systems Laboratory, Heriot-Watt University, Edinburgh, UK*

(Accepted March 26, 2014. First published online: April 29, 2014)

SUMMARY
Software engineering plays a key role in state-of-the-art robotics where more effective and efficient software development solutions are 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 challenges. This paper presents the analysis, design, and implementation of control software for an Autonomous Underwater Vehicle (AUV). The software development stages are carried out by using a model-driven toolchain that provides support to design and build component-based software for robotics applications. 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 importance 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 other disciplines 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 have tried 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 has limited its adoption by researchers and practitioners of robotics. In addition, 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.

* Corresponding author. E-mail: francisco.ortiz@upct.es

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

Model-Driven Software Development (MDSD), however, has been shown an increase in interest from robotics researchers of late. 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 artefacts 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) for robotics.

Such 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[1] is suitable to develop the AUV control software, and to 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.[1] Section 2 presents a background to support relevant topics on robotics in 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[1] 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 are aligned with the issues on requirements discussed below. Therefore, these are also aligned with the technological directions taken by other robotic software systems.

2.1. Software engineering for robotic systems

Software engineering best practises 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.[2] 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),[3] effectively support reuse of low-level software modules. Adaptive Communication Environment (ACE)[4] 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 software modifications affect: (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.

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

Most of the development processes limit the reuse of robotics software code to the 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 CBSE, deal with reusability and flexibility of software 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 software reusability (SR; components only interact with each other through ports).[8]

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, as well as run-time support for applications. A framework is a semi-complete application that provides a reusable 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 predominantly develop, integrate, and connect components using object-oriented technologies. This 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). Robotic developers should 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 the current component-based frameworks, such as those discussed in previous works.[9] These limitations probably make CBSE unattractive for robotics. The two major milestones needed for the CBSE uptake in robotics are as follows: (1) Creation of techniques, tools, and development processes to design applications. Applications should start from components as atomic units so that the complexity of software artefacts can be hidden from robotic developers. (2) Development of mechanisms to turn component-oriented designs into executable programs, e.g. an interpreter or a compiler for a components language. These problems can be overcome by adopting an MDSD approach.

2.3. Model-driven software development

Model-Driven Software Development allows domain experts to directly express their knowledge by using, and reusing, models that 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 the development of robotics software components. Technically, the combination of MDSD and CBSE is realized by defining meta-models, supporting model transformations, and providing an MDSD toolchain.

Meta-model definition: This process involves the definition of meta-models to model CBSE concepts (software components). C-Forge[1] includes a meta-model, entitled WCOMM,[9] to model architectonic blocks as white-box components defined by their structures (ports required and provided interfaces), and internal behaviour (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 to executable code. Model transformations included in C-Forge[1] do not generate the full application code, but rather target a component

[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 orders of functional parameters, conversion between physical units, etc.

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

framework, entitled FraCC, which provides the required run-time support for component-based applications.

MDSD toolchain: This set of programming tools is for implementing the whole development process. The C-Forge toolchain[1] 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 by CBSE and MDSD.

3.1. Software engineering aspects of interest

The criteria used to evaluate how much current methodologies and technologies support robotics software engineering are as follows.

Software reusability: This aspect considers the degrees by which software blocks (CBSE) can be used and reused by different applications with little additional coding work. Robotics software reuse is divided into the following three categories:

- Functional Library (FL): Approaches within this category are software libraries that add specific functionality to a system. The user code directly invokes algorithms which are normally encoded as linked libraries (static or dynamic libraries). In addition, developers typically 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 Functional Libraries by providing a communication middleware for achieving distribution and modularity. Typically, it involves a publish/subscribe mechanism to exchange messages. Applications are built from sets of heterogeneous binary software modules. Robotics Middleware do not modify the internal structure of modules.
- Component-Based Framework (CBF): Approaches within this category provide similar support as those from robotics middleware. CBFs go beyond RM 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 and how they provide support for the development of such systems.

3.3. Discussion

Software Reusability: Approaches that support software reusability offer slightly different solutions, and use different development paradigms. Thus, it is difficult to compare 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 robotic software)[11] were presented earlier. A review of the state of the art can also be found in the Best Practice in Robotics (BRICS) deliverables.

Model Management: Many CBSE aspects are not considered by the robotics community. Nevertheless, BRICS highlights the need of formal models, and reuse of designs and architectural elements independent 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 previous research work[12] considered the Object Management Group (OMG) specification Robotics Technology Component (RTC) as the most relevant approach for robotics based on MDSD. It included the description of the three implementations of the standard Gostai RTC[13], OpenRTM-aist[14], and OPRos,[15] but concluded that none of these are matured enough yet.

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

Table I. Contribution to robotics software engineering aspects from current approaches.

		SR				
Approach	Highlights	FL	RM	CBF	MM	PA
MRPT	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.*; www.mrpt.org).	●	○	○	○	○
OROCOS KDL and BFL	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 (www.orocos.org/wiki/orocos/kdl-wiki and www.orocos.org/bfl).	●	○	○	○	○
ROS	Robot Operating System provides libraries and tools to develop distributed robotics software applications with the ROS middleware (www.ros.org).	○	●	●	○	○
YARP	Yet Another Robot Platform is an open-source framework that supports distributed computation with a main focus on robot control and efficiency (http://wiki.icub.org/yarpdoc/).	○	●	○	○	○
ORCA	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 (http://orca-robotics.sourceforge.net/).	○	●	●	○	○
RoboComp	Component-oriented robotics framework based on Ice middleware providing a skeleton of components and a set of tools to generate and test components (http://robolab.unex.es/index.php/robocomp).	○	●	●	○	○
OROCOS	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 (http://www.orocos.org/).	●	●	●	○	○
RT-MAPS	Commercial component-based framework for rapid prototyping of real-time applications (http://www.intempora.com/).	○	●	●	●	●
OpenRTM	Component object model in robot technology middleware for robot system integration. CORBA-based implementation of RTC specification (http://www.openrtm.org/openrtm/en).	○	●	●	●	○
OPRoS	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 (https://sites.google.com/a/odm-tech.com/opros/home)	○	○	●	●	○

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

Table I. Continued.

Approach	Highlights	SR			MM	PA
		FL	RM	CBF		
BRICS	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 toolchain that includes a CBSE meta-model, and a repository of harmonized algorithms (www.best-of-robotics.org).	○	○	●	●	○
RobotML	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.	○	○	●	●	○
SmartSoft	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 (http://smart-robotics.sourceforge.net/).	○	○	●	●	◍
EasyLab	EasyLab is a model-based development tool for modeling, simulation, code generation, and debugging with a primary focus on mechatronics. It uses MDSD to model robot systems at different levels of abstraction.	○	○	●	●	○
AADL	Architecture Analysis & 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 (http://www.aadl.info).	○	○	○	◍	●
MAST	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 (http://mast.unican.es/)	○	○	○	○	●
Uppaal	Tool for component-based modeling, simulation, and verification of real-time and embedded systems modeled as real-time components (www.uppaal.com)	○	○	○	○	◍
Cheddar	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 (http://beru.univ-brest.fr/ singhoff/cheddar/).	○	○	○	○	●
C-Forge	C-Forge is a model-driven toolchain that relies on WCOMM, 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 (www.dsie.upct.es/cforge/).	○	◍	●	●	●

Table key: ● = fully supported, ◍ = partly supported, ○ = non-supported.

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

Performance Analysis: There are some MDSD-based approaches for real-time systems such as ArtistDesign[16] for embedded systems design, and the OpenEmbeDD project.[17] Furthermore, the automotive industry domain has standardized AUTOSAR.[18] The situation is different in the robotics domain, where none of the currently available component models are analysable. This is mainly because neither the underlying component model nor its supporting runtime infrastructure was designed to be analysed. Most importantly, not 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.

Our MDSD approach, 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 based on MDSD. One of the transformations leads to models that involve notation of real-time 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[19] to carry out such real-time 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 an MDSD toolchain to develop component-based software applications. It has two main elements: (1) A modelling language (WCOMM) for component-based applications (at the top of Fig. 1), and (2) a C++ framework (FraCC) that facilitates the implementation of WCOMM-modelled applications (at the bottom of Fig. 1).

4.2. The WCOMM component meta-model

A software component is modelled as a white box with WCOMM. Unlike the 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 modelled 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 the code directly. A component-oriented architecture can be defined with WCOMM (step "a" in Fig. 1) based on the requirements for the application under development. A WCOMM component is a software entity that encapsulates its internal state, only communicating with other components through messages sent to, as well as received from, their ports, and its behaviour is modelled by means of timed automata. Messages are asynchronous without response, which makes possible to support any communication scheme.

Timed automata are finite state machines, similar to those defined in Unified Modeling Language (UML) but richer in notation, since it includes temporal properties such as 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 comprises 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

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

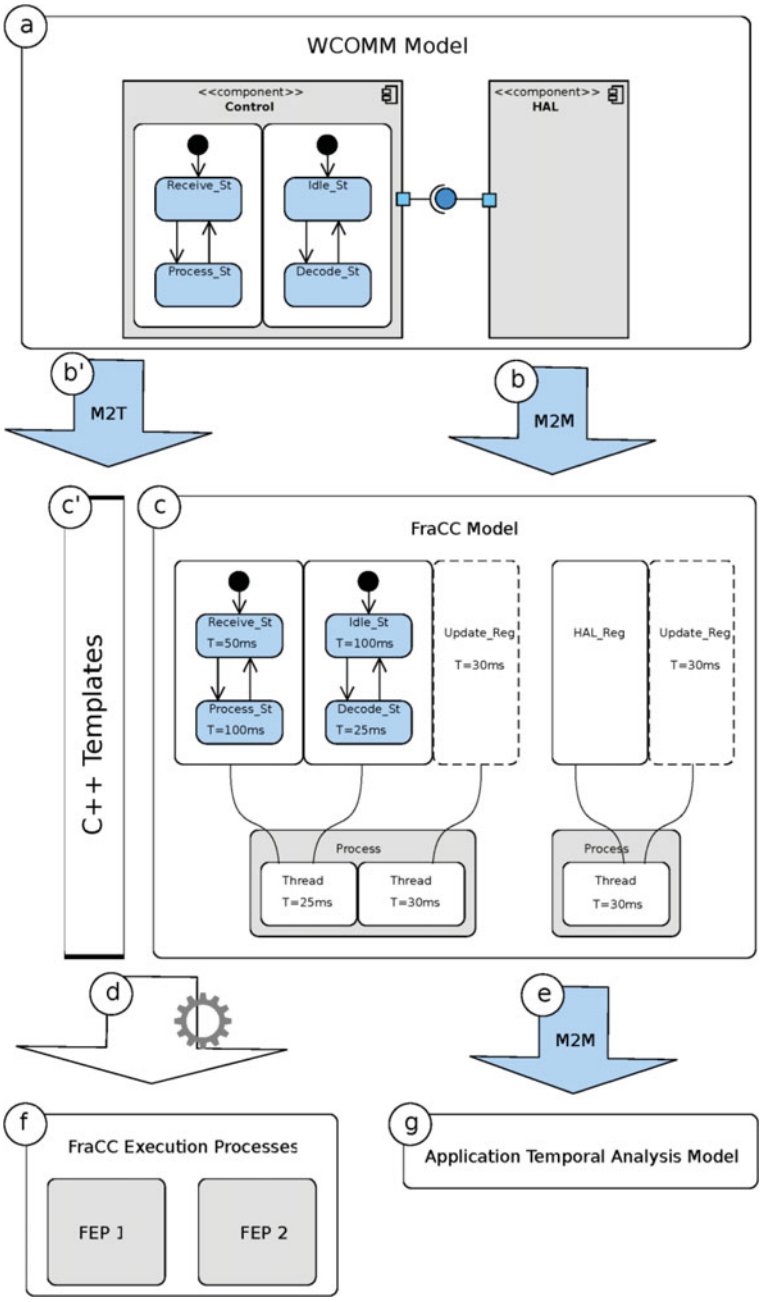


Fig. 1. The C-Forge MDSD process.

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

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

Once the application is modelled in WCOMM, the artefacts needed to run the application in FraCC (deployment model as shown in step "c" of Fig. 1), and the C++ templates where the user adds the code (step "c"' of Fig. 1) are generated. FraCC allows developers to explicitly control the concurrency

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

properties of the application by using a deployment model that is generated by a model-to-model transformation (step "b" in Fig. 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 unit of workload that can be assigned to a thread. Regions, not states, play this role 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 from other approaches such as Smartsoft, YARP, ROS, BRICS, or OROCOS. In addition, 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 (steps "d" and "f" in Fig. 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 (steps "e" and "g" in Fig. 1) of the FraCC model. This resulting model is used to carry out an early verification of real-time requirements by using an analysis tool (Cheddar[19]). Appropriate changes can be made in the applications models as needed based on the outcomes of analysis. The deployment model provides C-Forge with flexibility since it can test components with various deployment configurations, e.g. 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 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 dispatch the tasks specified in the mission plan. Tasks are carried out by executing functions from other software modules (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 execution of tasks 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[20] and inspired by the multi-stage pipeline computing technique applied in computer architecture. The

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

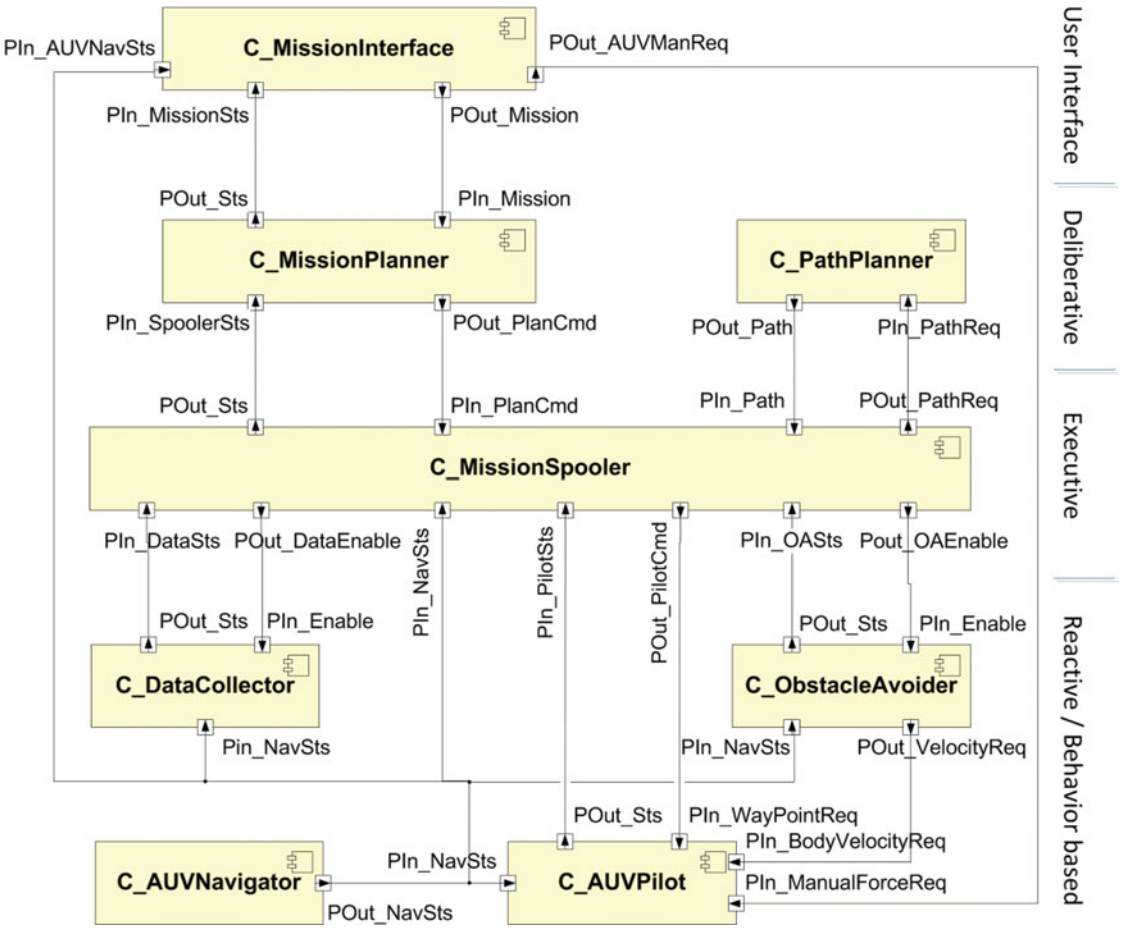


Fig. 2. Diagram of components of the AUV control application.

performance of the mission spooler to simultaneously deal with different tasks from a mission plan is a critical aspect to be analysed. Therefore, a real-time 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 comprises the description of the component-oriented architecture for AUV. The functional requirements of the AUV Control Application (AUVCA) and the domain knowledge lead to a design of a three-tier architecture (Fig. 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 behaviour of the components with timed automata. It also refines the design of messages and common interfaces. These two steps can be carried out for 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 Fig. 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.

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

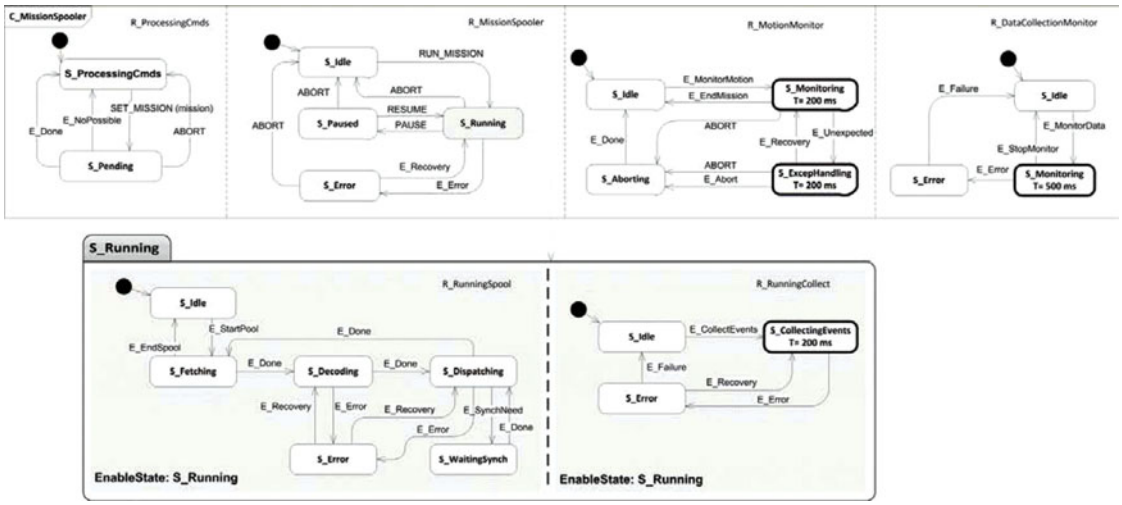


Fig. 3. Concurrent regions in the component *C_MissionSpooler* (top). The hierarchical state *S_Running* (bottom). Periodic states in bold relief.

The execution layer acts as an 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 Fig. 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, components *C_AUVNavigator* and *C_AUVPilot* are designed as wrappers of the AUV platform modules. The navigation system in the component *C_AUVNavigator* is based on an extended Kalman filter which aims at estimating the position of AUV from the measurements of 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 axes, as well as roll, pitch, and yaw). The AUV is also equipped with a gyroscope and a depth sensor used in the correction step. In addition, the algorithm embedded in the component *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 and exclude pitch and roll. Existing approaches rely on the centre of mass and buoyancy of the AUVs to prevent pitch and roll disturbances.

5.3. Behaviour modelling of the mission spooler

The behaviour of the software components (the structure of components is shown in Fig. 2) is described by a timed-automata tool provided by C-Forge. We discuss only the *C_MissionSpooler* behaviour (Fig. 3) since it is the most critical element of the AUVCA. The spooler gets mission commands as inputs from the component *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 constraints.

The component *C_MissionSpooler* contains four concurrent regions (mission spooler functions that can be executed in parallel and in real-time). These regions (Fig. 3; from left to right) are: mission command processing, control of mission execution in progress, motion tracking, and monitoring data collection. In the region *R_MissionSpooler* there is a hierarchical state (*S_Running*) that is displayed at the bottom of Fig. 3. *S_Running* comprises two concurrent regions: (1) the region where each mission task is decoded and dispatched to the components that execute such a region (*R_RunningSpool*), and

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

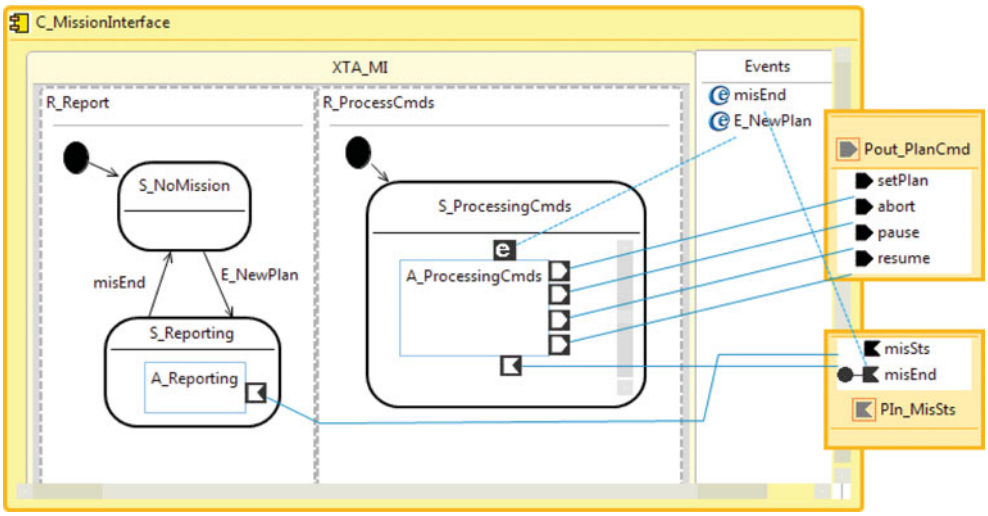


Fig. 4. The component *C_MissionInterface* with activities *shells* linked to the in/out messages with the C-Forge tool for simple component definition.

(2) the region in charge of collecting synchronization events from the components that are performing the tasks submitted by the mission spooler (*R_RunningCollect*).

When *C_MissionPlanner* sends a new mission to the component *C_MissionSpooler*, the mission is stored as a list of tasks in the latter (including timing and synchronization requirements). The event *E_StartSpool* fires the beginning of a cyclic process involving the three main states. Tasks are picked up one by one in the state *S_Feching*, decoded in the state *S_Decoding*, and finally dispatched or queued in the state *S_Dispatching*.

The execution unit of tasks in the mission spooler has a pipeline-based architecture (three-stage pipeline processing: fetching, decoding, and dispatching). For example, a list of tasks in a mission plan could include "*go to waypoint 1*" followed by "*record data 1 minute after starting moving*," and after reaching the first waypoint "*move to waypoint 2 without collecting data*." As the AUV heads to the first waypoint (once the task is launched in the state *S_Dispatching* of the component *C_AUVPilot*), the next instruction is decoded. The task "*record data*" requires no synchronization with the AUV positioning, so it is released to the component *C_DataCollector* with a time-out requirement. The next task requires the first waypoint to be reached before dispatching the next positioning task; therefore, the timed automata remain in the state *S_WaitingSynch*. Concurrently with this process, the region *Running_Collect* (in the state *S_*CollectingEvents) receives the event of first waypoint reached. A completion event *E_Done* is sent to the region *R_RunningSpool* to dispatch the task "*go to waypoint 2 and stop recording*," and continues decoding if there are more tasks in the list.

The data-collection task is monitored in the region *R_DataCollectionMonitor* along with the process described above 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_ExceptHandling*, 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 to either 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* (Fig. 4).

Each pin represents an input/output message or an event generated in such an activity, e.g. Fig. 4 shows the component *C_MissionInterface* which contains three states, two of these include activities.

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

```
#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_outMsg_abort;
     [...]
    AUVCAEvents::E_NewPlan  E_NewPlan;

   // *** USER CODE HERE ***
   if(get_E_NewPlan(E_NewPlan)){
        set_setPlan(outMsg_setPlan);
        reportUser(E_NewPlan);
   }
   if(userCmd(cmd)=="abort")
        set_abort(outMsg_abort);

    [...]
   // *** END USER CODE ***
}

void A_ ProcessingCmds::onExit(){ ... }
```

Fig. 5. Excerpt of the code for the activity "ProcessingCmds."

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 Fig. 1) is automatically carried out in C-Forge to obtain the FraCC deployment model (step "c" in Fig. 1) with a default configuration. At the same time, a model-to-text transformation (step "b" in Fig. 1) generates: (i) the C++ templates (step "c" in Fig. 1) of the software code corresponding to the "shells" of the activities (including the code to access 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 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 (steps "d" and "f" in Fig. 1) to become FraCC execution processes.

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

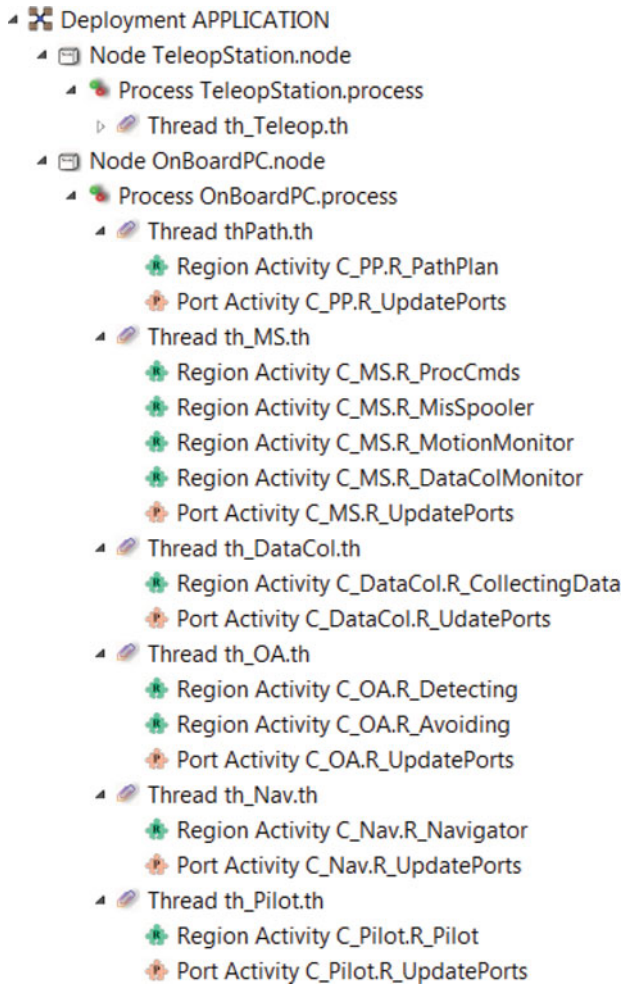


Fig. 6. Example of a deployment model in two nodes with one process per node and one thread per component in the OnBoardPC node.

The deployment model, using a default configuration, includes a single process per node and a single thread per component containing the component regions (*Region Activity* in Fig. 6). The framework adds an additional region to each component in charge of updating the data in ports periodically. This region is automatically included in the deployment (*Port Activity* in Fig. 6). The robotic designer modifies the configuration of the deployment model depending on the AUVCA requirements and design decisions. Initially, the AUVCA is distributed between two computers: (1) a tele-operation station computer, which serves as an operator control unit (including the components *C_MissionInterface* and *C_MissionPlanner* without strict real-time requirements); and (2) an on-board control computer (AUV) with the remaining components. Therefore, the default deployment is divided into two nodes with one process per node, as shown in Fig. 6.

Starting from this default deployment, an automatic transformation generates a temporal model (steps "e" and "g" in Fig. 1) to carry out an early verification of the real-time requirements of AUV's on-board components by using the Cheddar analysis tool. Table II 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. The period of execution for sporadic tasks is equivalent to their *minimum inter-arrival times* as period of execution. The timing properties (basically, T and C) of the regions are calculated by applying Eqs. (1) and (2). The timing properties of a thread are derived from the properties of the regions assigned to it by applying Eqs. (3) and (4):

$$T^i_{\text{reg}} = \gcd\left(T^j_{\text{act}}\right), \text{act}_j \in \text{reg}_i, \tag{1}$$

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

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 $D_i = T_i$.

Component/region	Type	T (ms)	C (ms)
C_AUVPilot			
R_Pilot	Sporadic	50	2
C_AUVNavigator			
R_Navigator	Periodic	50	2
C_ObstacleAvoider			
R_Detecting	Periodic	80	20
R_Avoiding	Periodic	80	20
C_DataCollector			
R_CollectingData	Periodic	100	10
C_PathPlanner	Sporadic		
R_PathPlanning	Periodic	1000	40
C_MissionSpooler			
R_ProcessingCmd	Sporadic	500	5
R_MissionSpooler	Sporadic	200	5
R_RunningSpool	Sporadic	200	5
R_RunningCollect	Periodic	200	5
R_MotionMonitor	Periodic	200	10
R_DataCollectMonitor	Periodic	500	10

$$C^i_{\text{reg}} = \max\left(C^j_{\text{act}}\right), \text{act}_j \in \text{reg}_i, \tag{2}$$

$$T^i_{\text{th}} = \text{gcd}\left(T^j_{\text{reg}}\right), \text{reg}_j \in \text{th}_i, \tag{3}$$

$$C^i_{\text{th}} = \sum (C^j_{\text{reg}}), \text{reg}_j \in \text{th}_i. \tag{4}$$

Starting from the timing requirements shown in Table II (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. Experiments were carried out on the AUVCA and used the deployment configuration presented in Section 5.4.

6.1. Schedulability analysis

A model-to-text transformation in C-Forge ("e" in Fig. 1) automatically generates an analysis file for the Cheddar tool to perform schedulability analysis. Cheddar is used to calculate response times for every thread, and check whether it is schedulable. 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 components' regions in order to make sure that all timing requirements (stated in Table II) are fulfilled at run-time. If the Cheddar-based analysis concludes, it is not schedulable, the deployment is modified by creating new threads and re-assigning regions to different threads. The analysis is then performed again to check the new deployment. Table III shows details of the five deployments studied, and the results of the Cheddar-based analysis.

Initial deployment: Figure 6 considers a process *OnBoardPC* with six threads, one per component, including the different concurrent regions whose characteristics are shown in Table II. The result of

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

Table III. Summary of the tested deployment configurations and Cheddar analysis results..

	Threads (tasks)	P	T	C	RT	Sch.
1st deployment	*th_DataCol*	2	150	12		N
	th_MS1	6	50	49	50	
	th_Nav	4	100	6		
	th_OA	3	150	10		
	th_Path	1	500	42		
	th_Pilot	5	100	6	1150	
2nd deployment	*th_DataCol*	2	150	12	85	Y
	th_MS1	5	100	34	50	
	th_MS2	3	150	15	76	
	th_Nav	6	100	6	12	
	th_OA	4	150	10	75	
	th_Path	1	500	42	254	
	th_Pilot	7	100	6	7	
3rd deployment	*th_DataCol_and_OA*	2	150	22	85	Y
	th_MS1	4	100	34	53	
	th_MS2	3	150	15	70	
	th_Nav_AND_Pilot	5	100	12	12	
	th_Path	1	500	42	254	
4th deployment	*th_Reactive_Layer*	4	150	34	36	N
	th_MS1	3	100	34	196	
	th_MS2	2	150	15		
	th_Path	1	500	42		
5th deployment	*th_Path*	1	500	42	254	Y
	th_MS1_and100	3	100	46	46	
	th_MS2_and150	2	150	37	85	

P = task priority; T = task period; C = task execution time; RT = task response time, calculated by Cheddar.

the Cheddar analysis of the first deployment is shown in Fig. 7. As not all task deadlines are met, the conclusion is that this deployment is not schedulable.

A more detailed study of the results of the schedulability analysis and Eqs. (3) and (4) reveals that if two regions with very different periods are assigned to the same thread, then the calculated timing properties can make the deployment non-schedulable. This is the case of the *C_MissionSpooler*, whose regions are assigned to *th_MS*. According to Eqs. (3) and (4), the calculated period of this thread is 50 ms and its worst case execution time is 49 ms. Therefore, the rest of the tasks cannot meet their deadlines, and the deployment is thus not schedulable.

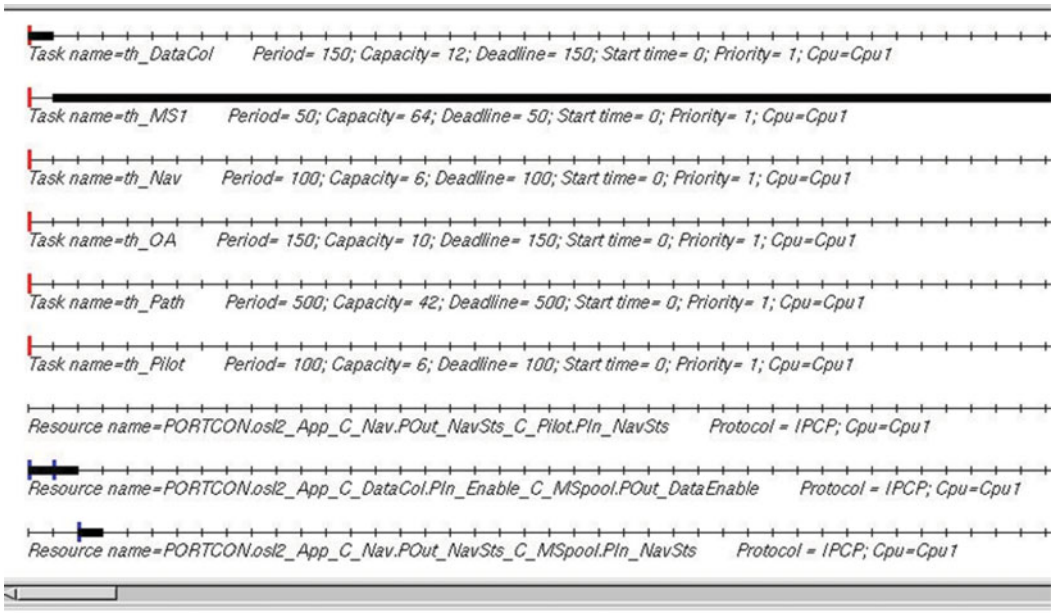
Second deployment: To avoid the lack of schedulability of the first deployment, a new assignment of the *C_MissionSpooler* regions in two new threads (*th_MS1* and *th_MS2*) for a total of seven threads is created. Although this second deployment is now schedulable (Fig. 8), the creation of more threads and shared resources increases the system overhead.

Third deployment: In order to decrease the system overhead introduced from the second deployment, a third deployment is tested. Regions of the components *C_ObstacleAvoidance* and *C_DataCollector*, and *C_Pilot* and *C_Navigator* are assigned to two threads respectively. Reducing the number of threads to five, the results of the analysis show that this new deployment is schedulable and also reduces the 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 into one. Unfortunately, the system is not schedulable in this case.

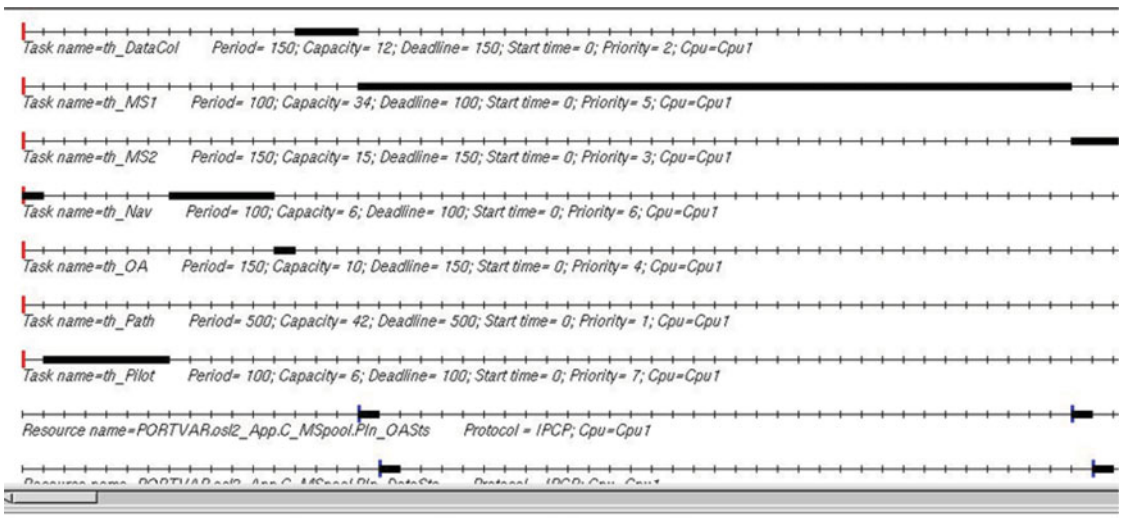
Fifth deployment: The last tested deployment is built around the idea of grouping together regions with the same period or regions where their periods are integer multiples of each other. In this case, only three threads are created having periods of 100, 150, and 500 ms. Regions with periods 100 and 200 ms are assigned to the first thread, while regions with 150 and 500 ms are assigned to the

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press




```
Task response times computed from simulations:
  th_DataCol => 12/worst
  th_MS1 => 0/worst, response time not bounded, missed deadlines
  th_Nav => 0/worst,  response time not computed since the task did not run all its capacity
  th_OA => 0/worst,  response time not computed since the task did not run all its capacity
  th_Path => 0/worst,  response time not computed since the task did not run all its capacity
  th_Pilot => 0/worst,  response time not computed since the task did not run all its capacity
- Some task deadlines will be missed: the task set is not schedulable
```

Fig. 7. Schedulability analysis for initial deployment configuration.




```
Task response times computed from simulations:
  th_DataCol => 85/worst
  th_MS1 => 50/worst
  th_MS2 => 76/worst
  th_Nav => 12/worst
  th_OA => 75/worst
  th_Path => 254/worst
  th_Pilot => 7/worst
The task set seems to be schedulable
```

Fig. 8. Schedulability analysis for the second tested deployment.

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

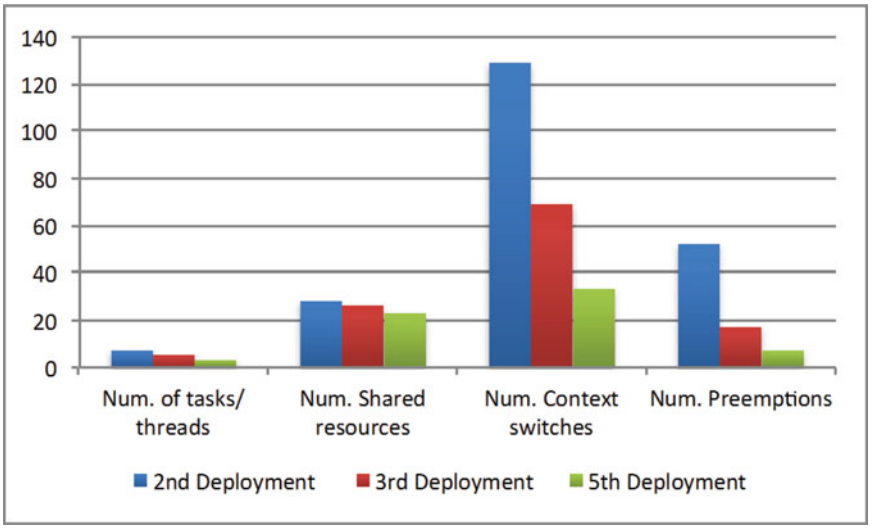


Fig. 9. Comparison among the number of threads, shared resources, number of thread preemptions, and context switches of three schedulable deployment models.

second and third thread respectively. The analysis process concludes that this last deployment using only three threads 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 the existing ones by creating new threads or reassigning regions, as discussed previously.

A comparison among the resources needed by each of the three schedulable deployments is shown in Fig. 9. The figure compares the number of shared resources created for each of the deployments 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 the most relevant 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 context switches. This is because shared resources are created whenever a region sends or receives messages from regions assigned to a different thread than 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. The data for this is not shown in Fig. 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, while an 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

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

various deployment scenarios without changing its architecture or losing control over the execution as some existing approaches do.

A case study where real-time software execution is required by an AUV has been presented. A particular analysis of real-time performance has been carried out on an AUV control application. Five alternative deployment configurations were analysed. These configurations were easily and quickly developed by changing some properties of the default FraCC deployment model. This makes possible to analyse the schedulability of the application with Cheddar from early development stages, and provides valuable information to add a new node, select alternative algorithms, modify the application architecture, *etc.* The software process based on C-Forge paves the way for a reduction of costs and risks in the 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 robots.

Future investigation will include the real-time 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 to analyse resource management in AUV systems. Further work will also be done on C-Forge to integrate transformations with 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 create behaviours. A reporting tool will display different deployment alternatives that optimize certain parameters (e.g. number of threads, shared resources, and communication bandwidth).

Acknowledgements

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

1. C-FORGE. Available at: http://www.dsie.upct.es/cforge/ (visited on April 2014).
2. E. Coste-Manière and R. Simmons, "Architecture, the Backbone of Robotic System," *Proceedings of the IEEE International Conference on Robotics & Automation*, San Francisco, CA (2000).
3. T. H. Collet, B. A. MacDonald and B. Gerkey, "Player 2.0: Toward a practical robot programming framework," *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*, Sydney, Australia (Dec. 2005) pp. 27–39.
4. D. C. Schmidt and C. Cleeland, "Applying patterns to develop extensible ORB middleware," *IEEE Commun. Mag.* **37**(4), 54, 63 (1999).
5. Homepage of 7FP EU Project, "BRICS: Best Practice in Robotics," Available at: http://www.best-of-robotics.org/ (visited on April 2014).
6. D. Brugali and P. Scandurra, "Component-based robotic engineering. Part I: Reusable building blocks," *IEEE Robot. Autom. Mag.* **16**(4), 84–96 (2009).
7. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. (Addison-Wesley, Boston, MA, 2002).
8. K. Lau and Z. Wang, "Software component models," *IEEE Trans. Softw. Eng*. **33**(10), 709–724 (Oct. 2007).
9. D. Alonso, C. Vicente-Chicote, F. J. Ortiz, J. A. Pastor and B. Álvarez, "V3CMM: A 3-view component meta-model for model-driven robotic software development," *J. Softw. Eng. Robot. (JOSER)* **1**, 3–17 (2010).
10. N. Mohamed, J. Al-Jaroodi and I. Jawhar, "Middleware for Robotics: A Survey," *Proceedings of The IEEE International Conference on Robotics, Automation, and Mechatronics (RAM)*, Chengdu, China (Sep. 2008) pp. 736–742.
11. A. Makarenko, A. Brooks and T. Daupp, "On the Benefits of Making Robotic Software Frameworks Thin," *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Diego CA, USA (Nov. 2007) pp. 167–180.
12. C. Schlegel, C. Steck and A. Lotz, "Model-driven software development in robotics: Communication patterns as key for a robotics component model," **In:** *Introduction to Modern Robotics*. (online ed.) (iConcept Press, Hong Kong, 2011).
13. Gostai RTC. Available at: http://www.gostai.com/products/rtc/ (visited on April 2014).
14. N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku and W. Yoon. "RT-Component Object Model in RT-Middleware – Distributed Component Middleware for RT (Robot Technology)," *Proceedings of the*

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press

IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA), Espoo, Finland (Jun. 2005) pp. 457–462.
15. B. Song, S. Jung, C. Jang and S. Kim. "An Introduction to Robot Component Model for OPRoS (Open Platform for Robotic Services)," *Workshop Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*, Venice, Italy (2008) pp. 592–603.
16. Artist-ESD, 2008–2011. "ArtistDesign – European Network of Excellence on Embedded Systems Design." Available at: http://www.artist-embedded.org/ (visited on April 2014).
17. OpenEmbeDD, 2008–2011. "OpenEmbeDD project, Model Driven Engineering Open-Source Platform for Real-Time & Embedded Systems." Available at: http://openembedd.org/home_html (visited on April 2014).
18. Autosar, 2008–2011. "AUTOSAR: Automotive Open System Architecture." Available at: http://www.autosar.org (visited on April 2014).
19. F. Singhoff, A. Plantec, P. Dissaux and J. Legrand. "Investigating the usability of real-time scheduling theory with the Cheddar project," *J. Real Time Syst.* **43**, 259–295 (2009).
20. US Department of Defense, *The Joint Architecture for Unmanned Systems (JAUS) Architecture Framework*, Vol. II, Part 1 Reference Architecture Specification (The Pentagon, Arlington County, VA, 2007).

https://doi.org/10.1017/S0263574714001027 Published online by Cambridge University Press