Developing an ROV software control architecture: a formal specification approach

Fabio Henrique de Assis
Xmobots Sistemas Robóticos
São Carlos - Brazil
Email: fhassis@gmail.com

Fabio Kawaoka Takase
Atech Negócios em Tecnologia SA
São Paulo - Brazil
Email: fktakase@gmail.com

Newton Maruyama
and Paulo Eigi Miyagi
Departamento de Engenharia Mecatrônica
Escola Politécnica - Universidade de São Paulo
São Paulo - Brazil
Email: maruyama@usp.br, pemiyagi@usp.br

***Abstract*—**
The software development of control architectures for Remotely Operated Vehicles (ROVs) is a complex task. The use of formal specifications for critical systems can improve both correctness and completeness of specifications and implementations. In this work, a new method for developing control architectures based on formal specifications is introduced. The chosen formal specification language is the CSP-OZ, a combination of the CSP language for behavioral model and the Object-Z language for data model. At first, the CSP parts of specifications are verified using the FDR2 model checker. Then, CSP-OZ model specifications are coded using the ADA language. More specifically, the ADA language profile Ravenscar for concurrency and the SPARK language with its annotations for data modelling are used. The SPARK annotations give support for the Object-Z specifications. Later, the SPARK examiner can be used to statically check the code against the annotations. In order to illustrate the application of the method, the development of the software control architecture of the LAURS ROV is introduced. The embedded system is based on a PC104 Intel x86 running the real time operating system Vxworks.

I. INTRODUCTION

The development of software control architectures for Remotely Operated Vehicles (ROVs) is a complex task. These control architectures might be characterized by the following attributes: real-time, multitasking, concurrency, and distributed over communication networks. In this scenario, there are multiple processes running in parallel, possibly distributed, and engaging in communication between each other. In this context, the behavioral model might lead to phenomena like deadlocks, livelocks, race conditions, among others.

For safety critical systems, the use of formal specifications for software development is known to improve software correctness and completeness for both specifications. This might be achieved mainly because formal specifications establish models that can be checked using verification tools. A recent and extensive survey of the industrial use of formal methods are presented in [1].

A formal method is composed by a specification language and verification tools. The formal specification languages are usually classified into two types: behavioural modeling languages that describes concurrent processes and their interactions, like CCS [2] and CSP [3], [4]; and data modeling languages, like VDM [5], Z [6] and Object-Z [7]. Some languages try to combine both aspects, these includes languages like CSP-OZ [8] and Circus [9]. Other methods, like Timed Automata [10], allow both behavioral modeling and reasoning about time properties.

The model checking of formal specifications is a task of reasoning on software system specifications, in which a tool verifies certain properties, by means of an exhaustive search of all possible states that a software system could enter during its execution. In this context, it is possible to check about correctness, liveness, deadlock, etc.

At first, some related work is reviewed. In [11] a formal method based on CSP-OZ is integrated into a software engineering process with UML and the Java programming language. A UML profile is developed for the CSP-OZ language. CSP models are derived from initial UML models. The Object-Z part can be transformed into assertions in the Java Modeling Language (JML) [12]. Jassda [13] offers trace assertions in a CSP-like notation for specifying the required order of method calls. The CSP part of the model design is verified using the FDR2 model checker [14] and the JML runtime assertion checker verifies the model implementation.

In [15] different techniques are applied that covers from requirement analysis to code testing. The key elements are: a restricted use of Timed Automata that uses *delay* and *deadline* to define temporal behavior notions, notions of *rely* and *guarantee* to cover temporal dependencies, model checking based on the UPPAAL verification tool [16], for design verification, the SPARK language [17] that allows annotations and the Ravenscar profile [18] for concurrency model, scheduling and response time analysis for implementation compliance.

In this work, a new method for developing software control architectures based on formal specifications is introduced. The chosen formal specification language is the CSP-OZ language [8], [19], a combination of the CSP language for behavioral modelling and the Object-Z language for data modelling.

At first, the CSP parts of specifications are verified using the FDR2 model checker [14]. Then CSP-OZ model specifications are coded using the ADA language [20]. More specifically, the ADA language profile Ravenscar for concurrency and the SPARK language with its annotations for data modelling

[21], [22], [20]. The combination of the Ravenscar profile and the SPARK language is usually known as RavenSPARK [18]. The SPARK annotations give support for the Object-Z specifications. Later, the SPARK examiner can be used to statically check the code against the annotations.

In order to illustrate the application of the method, the development of the software control architecture of the LAURS ROV is provided. The embedded system is based on a PC104 Intel x86 running the real time operating system Vxworks.

The paper is organized as follows. Section II details the proposed method. Section III introduces the LAURS ROV control architecture. Section IV highlights some results on the application of the proposed method. Finally, section V draws general conclusions.

II. The method

As illustrated in figure 1, the method is composed by three steps. The method is detailed in the following sections. More detailed information about the method can be found in [23].

A. System modeling

1) System structure diagram: The system structure diagram represents model components and their communication interactions. The gCSP [24], a graphical tool for CSP model editing, is used for this task. Each component represents an independent process which might communicate with others via synchronous unidirectional typed channels.

Channels are named according to the following pattern:

start_end:type

where the first part of the name represents the process that sends the communication message and the second part represents the processes that receives it. After the name, the data type used by the communication channel is declared after the symbol ': '.

2) Components specification: At this stage, the objective is to do a more detailed specification of each process that composes the system, including its internal data structure and dynamic behaviour. For that, the formal specification language CSP-OZ [8], [19], a combination of the process algebra CSP with Object-Z (an object-oriented extension of Z). In CSP-OZ, processes are described as classes (*Class*) that have two parts: a CSP part and an Object-Z part.

The CSP part contains the specification of the process communication interface together with the process dynamic behaviour. The interface includes declarations of all channels used by the process, and declarations of the methods described in the Object-Z part. When using the CSP-OZ language, Object-Z methods are interpreted as internal events of the CSP process. The difference between a channel and a method in the interface is detached through the use of the reserved words " `chan`", for channels, and " `method`".

The main part, indicated by the reserved word "`main`", contains all events that a process might accept, including calls on communication channels and internal methods. So, an example of specification of a basic behaviour that cyclically executes two methods might be:

$$\texttt{main} = \mathit{doSomething} \rightarrow \mathit{doOtherThing} \rightarrow \texttt{main}$$

The Object-Z part consists of specification of the internal data structure including its associated methods. Internal variables with their initial values and internal methods must be declared. Methods associated with communication channels must be identified using the prefix " `com_`" plus the name of the channel. For a channel named *theChannel* there must exist a method named `com_`*theChannel* in both process related with the channel.

A general class C might be represented by the following schema:

C	
I	[Interface]
P	[CSP Part]
Z	[OZ Part]

which can be written horizontally as specs I P Z end. In the Interface I, channels and types are declared. P is a CSP process and Z is an abstract data type specification using Object-Z and is defined by:

Z	
$st : State$	[State Space]
$Init(st)$	[Initial Condition]
$\ldots \texttt{com\_}c(st, in?, out!, st') \ldots$	[one schema for each c in I]

3) Model checking: The graphical tool gCSP exports the structure model into CSP_M language, which is the language of FDR2 model checking tool [14]. The CSP_M language is an extension of CSP [25]. While using FDR2 it is possible to verify issues like deadlock, liveness, determinism and refinement relationships. There are works [25], [26], [27] that try to check the Object-Z part of CSP-OZ models together with the CSP part using data structures provided by the CSP_M language. In this work, only the CSP part is checked using the FDR2 tool.

B. Model implementation

In this project, the Ada programming language is adopted [20]. The use of ADA is quite widespread in the development of software for critical systems. Among the advantages of its use are its high legibility, real time programming facilities, strong typing and existence of profiles and language subsets for critical systems, like the Ravenscar profile and the SPARK language [21], [22], [20]. The use of SPARK language together with the Ravenscar profile is also known as RavenSPARK [18], which consists on the use of the SPARK language for sequential code constructs and the Ravenscar profile for concurrent programming.

1) Code generation: In this work, the generation of code is done manually.

The SPARK language and the Ravenscar profile impose restrictions related to the use of the Ada language constructs

[18], [28], [29]. In this work, the following issues are of interest:

- Use of dynamic allocation, pointers and generic types are prohibited;
- Communication between tasks are restricted to protected or atomic objects;
- Use of select, abort and relative delays inside tasks are prohibited;

Taking into account these restrictions, a set of implementation rules have been elaborated in order to convert CSP-OZ specifications into RavenSPARK. The creation of processes and channels are of special interest. Package encapsulation, tasks, protected objects and strong typing are the utilised concepts.

The creation rules of processes are:

- Each process is created as a separate package, that contains internal methods and an Ada task, that acts as the actual CSP process. This has the same name of the package plus the suffix " _task ".
- The internal methods of each process are created inside the package, and not inside the task. This facilitates verifications with the SPARK examiner and also reduces size and complexity of the task internal code.
- The internal variables of the processes are located inside the task, and not in the package. They are encapsulated inside the task, and cannot be accessed directly by other tasks in the same way that occurs in the CSP language.
- All processes execute in infinite loop. This is due to the Ravenscar profile that states that task termination is considered as a program error.

For the CSP channels, their implementation using RavenSPARK has been developed based on [30]. The RavenSPARK channels are composed by two protected objects: *Data* that stores the transmitted data through the channel and blocks the receiver process; and *Sync*, that blocks the sender process until that the data is received by the other process. Based on that, the following implementation rules for the CSP channels in RavenSPARK are established:

- Each channel type demands a different package. By convention, the package receives the channel type as part of its own name. For example, if the channel is of the type integer than the correspondent package type is *IntegerChannel*. If the channel is of the type message than the correspondent package is of the type *MessageChannel*, and so on.
- The internal data stored in the protected object *Data* has the same type of the channel.
- Each channel that is presented in the gCSP diagram should be created inside the package that represents its channel type. In this way, all channels that have the same type are created together inside the same package. This is due to restrictions imposed by the RavenSPARK profile
- The name of the channel instances obeys the following rule: the prefix is the same name present in the gCSP diagram, and the suffix differs for the two components of the channel: " _d " for the object of type *Data* and " _s " for the object of type *Sync*. For instance, for an integer channel whose name is *myChannel* in the gCSP diagram, its RavenSPARK implementation will consist on the creation of two objects : *myChannel_d* and *myChannel_s*, both created inside the package *IntegerChannel*.
- The message sending and receiving are realised via the methods *put*, *get*, *stay* and *proceed*. It works in the following way:
  - Inside the task who is transmiting the data *someData*, the code for using the channel would be, for example:

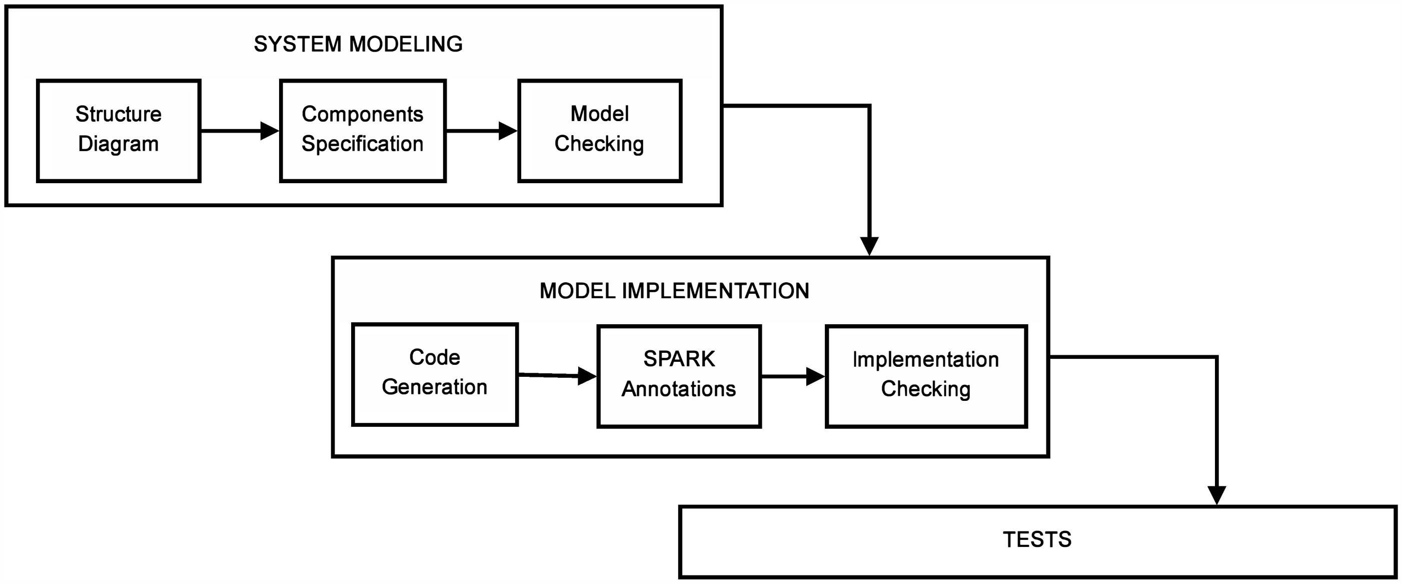

Fig. 1. The proposed method.

```
IntegerChannel.myChannel_d.put(someData); --
      send the data
IntegerChannel.myChannel_s.stay;          --
      wait for the reading
```

– Inside the task that is receiving the data *myData*, the code for using the channel would be, for example:

```
IntegerChannel.myChannel_d.get(myData); --
     read the data
IntegerChannel.myChannel_s.proceed;     --
     release the sender
```

2) SPARK annotations: Annotations are inserted inside the code as ADA comments which begins with symbols "–#". The piece of code shown above illustrates an example of these annotations for the *Add* procedure. The types of annotations include pre-conditions, post-conditions and invariants of variables and methods, priorities for tasks and protected objects, and specifications on the use of variables (reading or writing) along the program. More detailed information might be found in [17] and [18].

```
procedure Add(X: in Integer);
--# global in out Total;
--# derives Total from X;
--# pre X > 0;
--# post Total = Total~ + X;
```

Object-Z part specifications can be easily mapped onto SPARK annotations.

3) Implementation checking: Based on the inserted SPARK annotations, the SPARK examiner is capable of performing static verifications:

- Check the compliance between the implementation code and the SPARK language rules;
- Check the consistency between the implementation code and its annotations, being able to perform data flow analysis, control flow analysis or generate verification conditions.

C. Tests

After the modelling and implementation phases together with their respective checking procedures have been performed, tests are made inside the simulation environment of the Windrivers Workbench development system. The simulation environment emulates an Intel x86 embedded system architecture running the real time operating system Vxworks. Later, tests are conducted in the real hardware which is an Intel x86 PC104 system.

III. The LAURS control architecture

Unmanned underwater vehicles (UUVs) are usually classified into two categories: the first one is named remotely operated vehicles (ROVs). They are connected to a remote station (usually inside a vessel) and require direct human assistance for all operations, for example, vehicle guidance, positioning, manipulator operation, etc. The second one is known as autonomous underwater vehicles (AUVs) and are characterised by autonomous behaviour and absence of a tether cable [31].

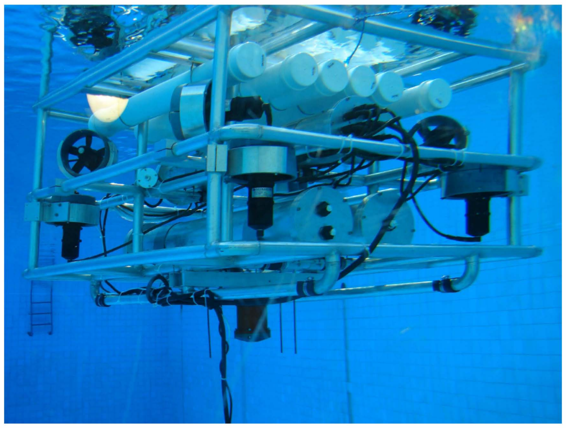
Fig. 2. The LAURS ROV.

A holonomic semi-autonomous UUV, named LAURS [32], is being developed at the Laboratory of Sensors Actuators at the University of Sao Paulo. The LAURS has been conceived as having semi-autonomous behaviour, i.e., the vehicle are remotely operated but have also an autonomous mode that can be used to approach the target. Despite the possibility of having an autonomous behaviour, the vehicle is mostly a ROV due to is operation characteristics and mechanical design. The vehicle has been initially conceived to provide inspection and intervention capabilities in specific missions in deep water oil fields.

The vehicle configuration is composed of an aluminum tubular structure, $l = 1.4m \times w = 1.2m \times h = 0.9m$, equipped with three pressure vessels of the same dimensions, $l = 1.0m \times d = 0.167m$. Its weight in air is about $200Kg$ and the weight-buoyancy force is $35N$ positive. For description convenience, the LAURS is divided into two parts: upper and bottom. The upper part of the vehicle contains a layer of PVC tubes for buoyancy properties, a pressure vessel for the electronics and sensors, and four horizontal thrusters (see figure 2). The bottom part of the vehicle consists of two pressure vessels that contain batteries and four vertical thrusters. Modular structural components allows that LAURS can be easily reconfigured in agreement with specific tasks. The overall structure of the vehicle is symmetric with respect to both the xz and yz planes, and the eight thrusters are arranged two by two in the corners, with the horizontal ones parallel to the diagonals of the xy plane. This particular thruster configuration enables the full controllability of the vehicle motion.

The development of the architecture follows the hybrid paradigm [33]. Two functional layers (See figure 3) are identified: a *deliberative* layer, responsible for mission planning activities, and a *reactive* layer, responsible for sensorial and motion control activities.

Processes, that are independent pieces of software, are represented by rectangles. Interactions between processes oc-

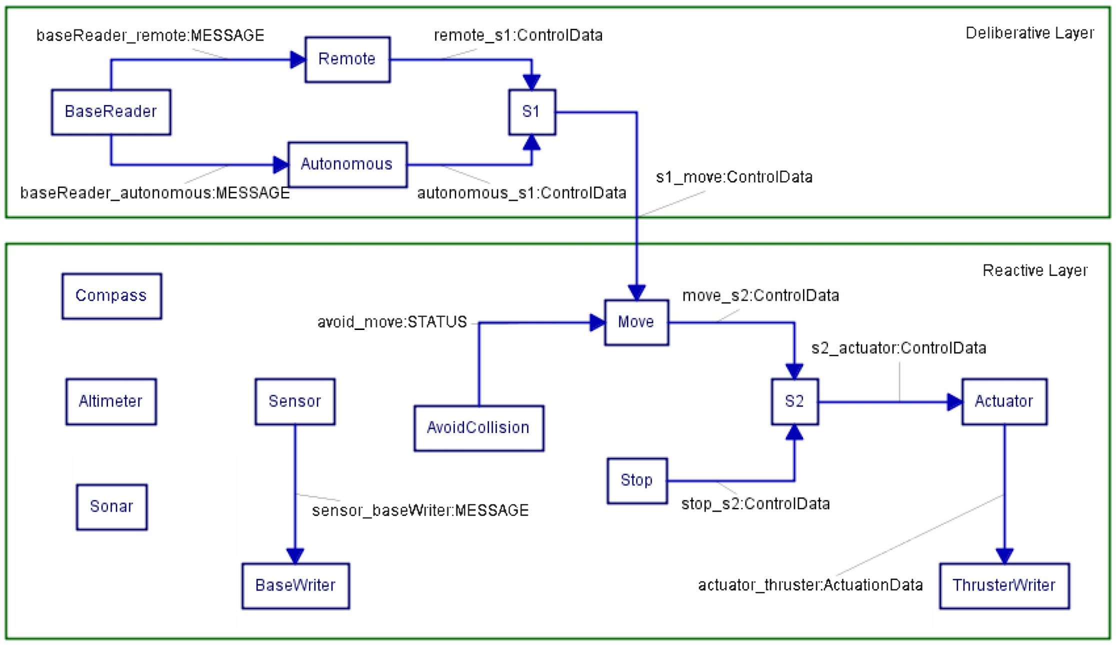

Fig. 3. The LAURS control architecture: processes and channels.

cur via unidirectional synchronous communication channels, represented by arrows. Channel names are chosen to indicate the message flow direction together with its data type: *start_end:type*. The processes *Compass*, *Altimeter* and *Sonar* do not possess any communication channels. Readings of their associated sensors are stored on a data structure named *BlackBoard* that serve as an asynchronous exchange of information.

IV. Experimental results

The CSP_M script generated based on the CSP-OZ model is checked in FDR2 for concurrency issues like deadlocks, livelocks and determinism. These tests must be conducted for the complete system, i.e. considering all modules and their interactions.

The process named *ROV_Control_Module* (which represents the whole system) is tested successfully regarding the existence of *deadlocks*, *livelocks* and determinism. It is important to emphasize that the system under analysis is of considerable size, which indicates that the proposed modeling method allows that larger systems can be analyzed. According to the output report obtained from the tool, the system has the following complexity:

- Number of States: 2,519,424
- Number of Transitions: 23,549,616

It is observed that the number of system states is quite large, which makes impossible to conduct a manual analysis of its state machine. Even with large number of states, as can be observed in table I, the time spent with the analyses is relatively short. The analysis of determinism is the largest one, taking a little less than one hour of processing time. The other analyses take less than two minutes to be conducted.

TABLE I
Architecture Analysis Time on FDR2.

Analysis done	**Time spent [s]**
deadlock	82
livelock	82
determinism	3301

These analyses have been done on a notebook, with an AMD Semprom processor of 1.8 GHz with 1 GB of RAM memory.

The code developed using RavenSPARK has been verified successfully using the SPARK Examiner tool.

V. Conclusions

In this work, the development of a software control architecture for the LAURS ROV has been approached as a safety critical system. The use of formal specifications for software development is a recommended technique in order to achieve software correctness.

A new method has been proposed based on the use of the CSP-OZ specification language, the gCSP graphical tool, the FDR2 model checker of Formal Systems Ltd, the Ravenscar profile, the SPARK language and examiner of Altran Praxis, the GNAT Ada compiler of Adacore, the Workbench development system and the VxWorks real-time operating system of Windriver.

The use of model checking tool has been performed in two different stages of the method. First, the CSP parts of specifications that model concurrent aspects is checked using the FDR2 tool, Later, the Object-Z parts of specifications are converted into SPARK language annotations and then checked using the SPARK examiner.

In our example, the performance of model checkers have been satisfactory. Further analysis, however, must be made for the cases of much larger systems in order to check the feasibility of the method. A major drawback of the model is that conversions from specifications into code must be done manually, which is an error prone approach. An important improvement would be the design of Design Patterns mapping CSP-OZ specifications into code. In this context, a software tool might be designed to convert specifications into code automatically or semi-automatically.

ACKNOWLEDGMENT

The authors would like to thank the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) and the Coordenação de Aperfeiçoamento de Nível Superior (CAPES) for the finantial support, the Formal Systems Ltd for the academic license of the FDR2 model checker, the Adacore for the academic license of the GNAT ADA compiler, the Altran Praxis for the academic license of the SPARK toolset, the Windriver for the academic license of the Workbench and the Vxworks real time operating system.

REFERENCES

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, 2009.

[2] R. Milner, *A Calculus of Communicating Systems*, ser. Lecture Notes in Computer Science. Springer, 1980, vol. 92.

[3] C. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[4] A. W. Roscoe, C. A. R. Hoare, and R. Bird, *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[5] C. B. Jones, *Systematic software development using VDM (2. ed.)*, ser. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[6] J. Woodcock and J. Davies, *Using Z: specification, refinement and proof.* Prentice-Hall, 1996.

[7] R. Duke and G. Rose, *Formal object-oriented specification using object-z*, ser. cornestones of computing, R. Bird and T. Hoare, Eds. Macmillan, 2000.

[8] C. Fischer, "CSP-OZ: a combination of Object-Z and CSP," in *FMOODS '97: Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*. London, UK, UK: Chapman & Hall, Ltd., 1997, pp. 423–438.

[9] M. Oliveira, A. Cavalcanti, and J. Woodcock, "A denotational semantics for circus," *Electr. Notes Theor. Comput. Sci.*, vol. 187, pp. 107–123, 2007.

[10] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[11] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim, "Integrating a formal method into a software engineering process with uml and java," *Formal Asp. Comput.*, vol. 20, no. 2, pp. 161–204, 2008.

[12] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with jml and esc/java2," in *FMCO*, 2005, pp. 342–363.

[13] M. Brorkens and M. Moller, "Jassda trace assertions, runtime checking the dynamic of java programs systems," in *Proceedings of the International Conference on Testing of Communicating Systems*, 2002.

[14] F. Systems, *Failures-Divergence Refinement: FDR2 User Manual*, june 2005.

[15] A. Burns and T.-M. Lin, "An engineering process for the verification of real-time systems," *Formal Asp. Comput.*, vol. 19, no. 1, pp. 111–136, 2007.

[16] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *SFM*, 2004, pp. 200–236.

[17] J. Barnes, *High Integrity Software - The SPARK Approach to Safety and Security*. Addison-Wesley, 2006.

[18] S. Team, *SPARK Examiner - The SPARK Ravenscar Profile*, 1st ed., Praxis, December 2006.

[19] C. Fischer, "Combination and Implementation of Processes and Data: from CSP-OZ to Java," Ph.D. dissertation, Universidade de Oldenburg, january 2000.

[20] J. F. Ruiz, "Ada 2005 for Mission-Critical Systems," *Adacore report*, 2006. [Online]. Available: http://www.adacore.com/wp-content/uploads/2006/02/Ada05_mission_critical.pdf

[21] S. J. Goldsack, *Ada for Specification: Possibilities and Limitations*. New York, NY, USA: Cambridge University Press, 1985.

[22] J. G. Barnes, *Programming in Ada*, 3rd ed. Wokingham [u.a.]: Addison-Wesley, 1989.

[23] F. H. de Assis, "Checagem de arquiteturas de controle de veículos submarinos: uma abordagem baseada em especificações formais (in Portuguese)," Master's thesis, Escola Politécnica da Universidade de São Paulo, 2009.

[24] D. S. Jovanovic, B. Orlic, G. K. Liet, and J. F. Broenink, "gCSP: A Graphical Tool for Designing CSP Systems," *Communicating Process Architectures 2004*, 2004. [Online]. Available: http://www.djov.net/DT/JovanovicCPA2004.pdf

[25] C. Fischer and H. Wehrheim, "Model-Checking CSP-OZ Specifications with FDR," in *IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods*. London, UK: Springer-Verlag, 1999, pp. 315–334.

[26] G. Kassel and G. Smith, "Model Checking Object-Z Classes: Some experiments with FDR," in *Proc. Eighth Asia-Pacific Software Engineering Conference APSEC 2001*, 4–7 Dec. 2001, pp. 445–452.

[27] A. Mota, A. Farias, and A. Sampaio, "De CSPz para CSPm: Uma ferramenta transformacional Java (in Portuguese)," in *Workshop de Sistemas Formais*. In: Workshop de Métodos Formais, 2001, pp. 1–10.

[28] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in High Integrity Systems," *Ada Lett.*, vol. XXIV, no. 2, pp. 1–74, 2004.

[29] P. Amey and B. Dobbing, "High Integrity Ravenscar," in *Ada-Europe*, 2003, pp. 68–79.

[30] D.-A. Atiya and S. King, "Extending Ravenscar with CSP Channels," in *Ada-Europe*, 2005, pp. 79–90.

[31] Souza, E C de and Maruyama, N, "Intelligent UUVs: Some issues on ROV dynamic positioning," *IEEE Transactions On Aerospace And Electronic Systems*, vol. 43, no. 1, pp. 214–226, 2007.

[32] J. Avila, J. Adamowski, N. Maruyama, F. Takase, and M. Saito, "Modeling and identification of an open-frame underwater vehicle: The yaw motion dynamics," *Journal of Intelligent & Robotic Systems*, vol. 66, pp. 37–56, 2012.

[33] R. R. Murphy, *Introduction to AI Robotics*. Cambridge, MA, USA: MIT Press, 2000.