Copyright © 2009 by ABCM

MODEL CHECKING A ROV REACTIVE CONTROL ARCHITECTURE

Fábio Henrique de Assis, fhassis@usp.br
Newton Maruyama, maruyama@usp.br
Roberto Ferraz de Campos Filho, roberto.campos@poli.usp.br
Escola Politécnica da Universidade de São Paulo

Fabio Kawaoka Takase, fktakase@mind.eng.br
Mind Open Source Technology

Abstract. *The development of control architectures for Underwater Vehicles 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, we have 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. In order to try to minimize the effects of such difficulties, in this work a method for model checking control architectures of underwater vehicles based on formal methods is presented. The chosen formal specification language is CSP-OZ, a combination of CSP and Object-Z. Object-Z is an object-oriented extension of Z for the specification of predicates, typically, data pre, post and invariant conditions. CSP (Communicating Sequential Process) is a process algebra developed to describe behavioral models of parallel process. The model checking of formal specifications is a task of reasoning on specifications in which a system verifies certain properties by means of an exhaustive search of all possible states that a system could enter during its execution. In this context, it is possible to check about correctness, liveness, deadlock, etc. Also, one can relate two different specifications in order to check a refinement ordering. For the specifications, the model checker FDR of Formal Systems Ltd. is utilized. The implementation is developed using an ADA language profile called RavenSPARK, a union of the Ravenscar profile (developed at the University of York) and the SPARK language (a subset of the ADA language developed by Praxis, Inc.). The Ravenscar is a profile for developing processes, so CSP processes including their message channels can be easily deployed. On the other hand, SPARK is a language where one can insert data predicates (originally specified in Object-Z) using language annotations. The SPARK language has a tool, the Examiner, that can model check code based on these annotations. In summary, the proposed method allows model checking of CSP processes but does not allow any checking in the code level. On the contrary, Object-Z specifications must first be converted into a SPARK language code, together with proper annotations, and then model checking can be realised in code. The development of a real-time reactive control architecture of an ROV named VSOR (Veiculo Submarino Operado Remotamente) is used as an example of the use of the proposed method. The whole control architecture is coded using the ADA Language with the RavenSPARK profile and deployed into a PC104 cpu system running the Vxworks real-time operating system of Windriver, Inc.*

Keywords: *Model Checking, Formal Methods, CSP-OZ, SPARK*

1 INTRODUCTION

The development of control architectures for underwater vehicles 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, we have 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.

In this work, a method for model checking reactive control architectures of underwater vehicles based on formal methods is presented. The chosen formal specification language is CSP-OZ (Fischer, 1997; Fischer and Wehrheim, 2000; Fischer, 2000), a combination of CSP (Roscoe et al., 1997) and Object-Z (Duke and Rose, 2000). Object-Z is an object-oriented extension of Z for the specification of predicates, typically, data pre, post and invariant conditions. CSP (Communicating Sequential Process) is a process algebra developed to describe behavioral models of parallel process. The general idea is to develop the embedded system trying to reach its correctness by construction, a concept difficult to be reached without the use of formal methods. In a general way, formal methods are techniques that use mathematical language to describe computational systems. The specification of what a computational system should do using mathematical notation is known by Formal Specification (Lightfoot, 1991). A formal specification is a description of software or hardware that can be used both to implementation or tests purposes. The specification is focused on the behavioral aspects of the program, in other words, on the "what" the program does, and not on the operational aspect, that is the "how" it is implemented. The mathematical expressions have the advantage of being unambiguous, what eliminates confusions and the need of discussions on the meaning of a specification. Another great advantage consists of the conciseness of the mathematical expressions, which is relevant in the specification of bigger systems with higher levels of complexity

(Lightfoot, 1991).

There are several applications of formal methods in the industry. Some examples can be found in the works of (Sherif et al., 2003) and (Lawrence, 2005), that used formal specification in the embedded software of the Brazilian satellite SACI-I and in a commercial application made by the IBM company, respectively. But in the robotic field, the use of formal methods is not so common. There are two works that use them (de Medeiros et al., 1996) (Champeau et al., 2000), both used in underwater vehicles. The most used specification language in robotics is the UML (OMG, 2004), due mainly to its graphical notation and be supported for several tools. But there are several works proposing the formalization of UML diagrams, like Akhlaki et al. (2006), Yeung et al. (2005) and Ng and Butler (2002).

In this work, the control architecture for underwater vehicles is modeled using formal specification. The whole control architecture is hybrid (Murphy, 2000), having a deliberative and a reactive part. The deliberative part is represented by the autonomous behavior of the robot, that is not implemented in this work yet. So, the focus in on the reactive part. It was based on the *Subsumption* architecture (Brooks, 1986), which establishes that the system must be divided into several small and concurrent processes, that communicate among themselves over channels. This approach is very close to CSP, which facilitates the system specification. The system decomposition into small processes facilitates its model checking, once the system state machine is broken down into several smaller state machines that can be checked separately with a model checker tool. The compliance between the checkings in the model and implementation is guaranteed by the CSP-OZ rules, that relates the events of the CSP part with method calls of the Object-Z part.

So in this method is given a major focus on the system modeling, in order to reduce the time spent with implementation and tests. The main advantage in the use of formal methods is the possibility of checking the specified model before its implementation. The model is analyzed with the FDR model checker (Systems, 2005), that allows the checking of CSP models in order to find some common problems related to concurrency like deadlocks, livelocks and determinism. The implementation is done using subsets of the programming language Ada that are used in the development of critical systems, the SPARK Ada (Barnes, 2006) and Ravenscar profile (Amey and Dobbing, 2003; Dobbing and Burns, 1998). Their combination is also known as RavenSPARK (Team, 2006).

Figure 1. ROV used in this work.

As an example of application, a control architecture will be designed in order to be embedded in the robot show in the Fig. 1. For this architecture are done both model and implementation checkings.

2 PROPOSED METHOD

We are proposing in this work a robust development method for the embedded software of underwater vehicles. This one is based on the use of formal methods for the system specification, together with model checking tools to analyze both the model and its implementation. As can be seen in the figure Fig. 2, it is composed by three phases, witch can be divided in sequential steps.

The use of formal methods is done based in the possibility of doing formal proofs and verifications about the correctness of the developed model. Besides that, its use still allows the specification of details related to the conditions and limits of use of methods and variables of the system. In the case of this work, the focus will be only the use of model checking tools, not being elaborated any formal proofs. The use of model checking allows that a good part of the common mistakes can be discovered before the implementation phase of the system, what implicates in a great reduction of the software development costs (Amey, 2002). Depending on the size of the models, especially for the small ones, there is a significant increase on its reliability due to its checking with FDR (Lawrence, 2005). The checking of the implementation allows to find and to correct pieces of code prone to generate exceptions.

In the following sections each phase and stage of the method will be more detailed.

2.1 System Modeling

The first phase of the development consists of the software model elaboration. This should contain not only a specification of the system structure, but also a more detailed specification of each individual element, considering their data

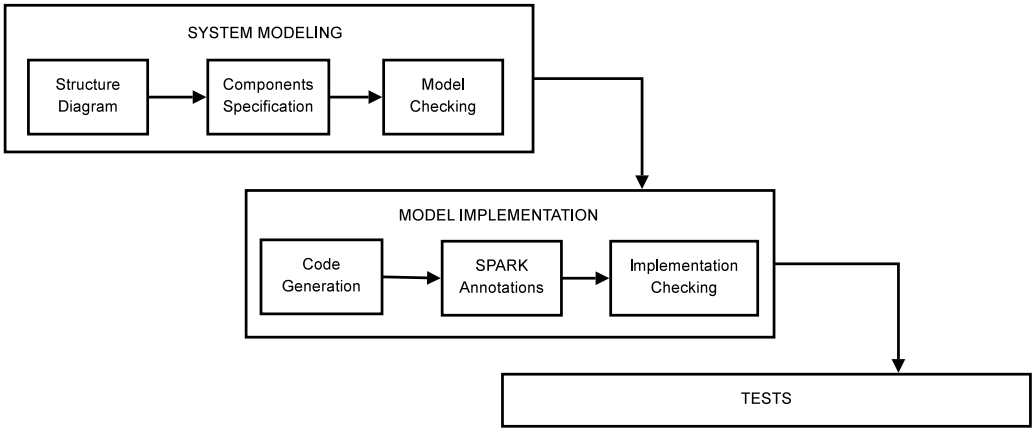


Figure 2. Proposed Development Method.

structures and behaviors. After concluded the specifications, these should be verified in FDR against deadlocks, livelocks, divergences and determinism.

2.1.1 Structure Diagram

In this stage will be elaborated the structure of the system, identifying all of its components and the interaction mode among them. This specification will be made in CSP, using the gCSP graphical tool (Jovanovic et al., 2004). The basic idea is that the system should be decomposed into independent processes, which communicate with each other in a synchronous way through unidirectional channels. The main idea is that they should just accomplish a specific function, with different functionalities being accomplished by different processes.

It should be also defined all the communication channels used among the system processes. These ones serve for exchange of information and synchronization. Regarding the nomenclature, this will follow the following pattern:

start_end:type

where the first part of the name represents the origin process and the second the process of destiny of the channel. After the name, we have specified the type of data transmitted by the channel, separated for the marker ': '. This standardization in the channel names facilitates the understanding of the generated code in the system implementation.

The division of the system in several smaller processes presents advantages and disadvantages. Among the advantages we can mention the increase of modularity, which facilitates the development and the performing of tests in each module in an independent way. Due to the reduced size of the processes, their implementation in code becomes much simpler and concise, what facilitates the verification process. However, the disadvantage of such division consists of the increase system complexity due to the increase in the number of components and consequently in their interaction manners. The use of tools like FDR facilitates the analysis of the system behaviors, making possible the performing of automatic analyses, reducing the effects of this complexity increase.

2.1.2 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 behavior. For that, the formal language CSP-OZ (Fischer, 1997, 2000) is used', a combination of the process algebra CSP with an extension of the language Z guided to objects, denominated Object-Z. In CSP-OZ, the processes are described as classes (*Class*) that have two parts: one in CSP and one in Object-Z.

The part CSP of a class in CSP-OZ consists of the specifications of both its communication interface and dynamic behavior. The interface includes all of the channels used by the process, besides all of the methods described in the Object-Z part. That is because the Object-Z methods can be seen in CSP as internal events of the process, what allows the methods calls to appear in the specification of the dynamic behavior of the process in CSP. 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`", for the methods implemented internally in the process. The names of the channels should be the same ones used in the diagrams done in gCSP. The main behavior is indicated by the reserved word "`main`", that contains all the events that the process can accept, including the calls in the communication channels and internal methods of the processes. So, an example of basic behavior that executes cyclically two methods would be:

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

The Object-Z part consists of the specification of the internal data structure, including their manipulation methods. In

this part must be specified the process internal variables with their initial values and all the internal methods. Besides, each communication channel used by the process should be specified in the class as a method whose name has the prefix " `com_`" plusthe name of the channel according to the name used in the gCSP diagram. This represents the effect of the use of the channel in the class, that usually consists of an exchange of information. So, for a channel name *theChannel* it would exist method `com_`*theChannel* in the two processes linked by the channel.

2.1.3 Model Checking

After having done the complete specification of the model of the system, in other words, the specification of the structure gCSP and the internal specifications of all the processes in CSP-OZ, they should be checked in order to eliminate possible errors related to concurrency. This type of errors doesn't appear during tests in the components individually, just existing as a result of their relationships. It is important to emphasize that in this stage the verification is just made in the model, not taken into account any aspect of the implementation.

The model checking is done based on a script done using the CSP_M language. This is the language used by FDR, and it is not just pure CSP, being an extension of CSP with a basic functional language (Fischer and Wehrheim, 1999). Through it, it is possible to do a series of verifications in the model, as the non existence of deadlocks and livelocks, check if the processes are deterministic and also to verify refinement relationships among different specification levels, since higher level specifications to ones closer the implementation. The gCSP tool allows the generation of this script automatically. It contains the specifications of all the processes in the model, together with their compositions and connections through the declared channels.

There are some works that propose some modes of elaboration of these scripts in CSP_M based on models in CSP-Z and CSP-OZ, as the works of Fischer and Wehrheim (1999); Kassel and Smith (2001), and Mota et al. (2001). These present mapping rules for both the dynamic part and for the data structure of the classes. As in this work the SPARK checking tools will be used to analyze the data structures of the processes, these won't be analyzed in FDR. Therefore the specifications in CSP_M should contain only the dynamic aspects of the CSP-OZ classes, or in other words, its CSP part.

The gCSP allows the specification of the system structure and the internal behavior of a process. In this method the internal specifications are inserted in the CSP_M script manually. So, the automatic generation of the processes specification done by the gCSP tool must be ignored. If you don't specify any internal behavior for a process, the tool automatically set a "null behavior" represented by the CSP `Skip` process. Besides this, the manual edition is also due to the Object-Z specifications not be inserted in the gCSP tool, and therefore these ones cannot appear in the automatically generated script. This manual edition follows a small set of rules:

1. All the methods declared in the interface of the CSP-OZ classes should be inserted manually in CSP_M as channels.
2. In the processes specifications in CSP_M, the automatically generated process `Skip` must be replaced by the specification present in the `main` of the CSP-OZ process specification.
3. The channel names in the gCSP diagram are kept in the channels in CSP_M. These channels are seen for the processes as events, that is, in the CSP_M specification will never appear any data being transmitted over the channels.

After all these modifications in the CSP_M script, it can be analyzed with the FDR tool. Once validated, we can go to the next phase of the method, the model implementation.

2.2 Model Implementation

The chosen programming language in this method was Ada. According to Ruiz (2006), Ada possesses a long success history in its use in critical systems. Among the advantages of its use are its high legibility (clear sintax), existence of basic resources for real time programming like creation and manipulation of tasks and use of temporal resources already included in the language (that is, they are not implemented as a library), use of strong typing and existence of patterns and subsets of the language for critical systems, like Ravenscar and SPARK (Goldsack, 1985; Barnes, 1989; Ruiz, 2006). These ones seek the development of deterministic and verifiable systems, and for that they will be used in this method. The use of SPARK together with Ravenscar it is also known as RavenSPARK (Team, 2006), consisting in the use of the language SPARK for sequential code and of a restricted subset of the Ada tasks in agreement with Ravenscar for concurrent programming. A more detailed description of the RavenSPARK use implications in this method will be shown later.

In the development of critical software, it is not desirable to use all the facilities of a complex language, once excessive complexity can harm the reliability. Instead, it could be used a reduced subset of the language, what would implicate in an embedded system more compact and efficient. Another advantage of subseting the language is the reduction of its complexity, which facilitates the generation of correctness proofs, previsibility, reliability and analysis of code covering, if necessary (Ruiz, 2006). That is the approach used in Ada by Ravenscar (Burns et al., 2004) and SPARK (Barnes,

2006). Both use that idea of restricting the language, in order that to obtain a more efficient, reliable and deterministic implementation . This approach is also observed in other languages, as the attempt of defining a subset "safe C" for the language C, denominated MISRA C, and the MISRA C++ for C++.

2.2.1 Code Generation

In most of the cases, the transformation of a formal specification into a program is a manual process. An example of this process can be seen in the work of Lawrence (2005), where a system made in the IBM was specified in CSP and implemented in the Java language. However, there are some tools that allow the automatic generation of code based in models like gCSP (Jovanovic et al., 2004), that allows automatic code generation in C++ and Occam based on CSP models. Another example of these tools is the Rhapsody of IBM (Telelogic, 2009). This allows the code generation in C, C++, Java and Ada based on UML-RT models. In this work, the transformation of the formal specifications into code will be done manually, based on the set of rules stated in this section.

The SPARK language together with the Ravenscar profile put a series of restrictions related to the use of the Ada language facilities, that can be seen in more details in Team (2006); Burns et al. (2004); Amey and Dobbing (2003). The ones who directly affect this method are:

- Prohibited the use of dynamic allocation, pointers and generic types;
- Communication between tasks are restricted to protected or atomic objects;
- Prohibited the use of select, abort and relative delays in tasks;

Taking into account these restrictions, it was elaborated a set of implementation rules in order to establish the way the CSP-OZ specifications are implemented in RavenSPARK. These include the creation of the processes and channels present in the control architecture. The implementation uses all the allowed language resources like package encapsulation, tasks, protected objects and specially strong typing.

Starting by the processes, their creation rules are:

- Each process was implemented like a separate package, that contains the process internal methods and an Ada Task, that indeed acts as the CSP process. This one has the same name of the package plus the suffix " _task ".
- As said before, the internal methods of each process were created inside of the package, and not inside the Task. This facilitates the verifications with the SPARK examiner and also reduces the size and complexity of the Task internal code.
- The internal variables of the processes are located inside the Task, and not in the package. This way they are encapsulated in Task, not being accessed directly by other Tasks in the same way that occurs in CSP.
- All processes execute in an infinite loop that never finishes. This is due to the Ravenscar profile, that states that task termination is considered as an error in the program.

For the CSP channels, its implementation in RavenSPARK was developed based on the works of Atiya and King (2005) and Atiya (2004). These propose an implementation for CSP channels in Ada using the Ravenscar prifile. As in this work will be used SPARK besides the Ravenscar profile, some modifications had to be done due to the restrictions imposed by the SPARK language and also to facilitates the verification of the developed code.

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, they were established the following implementation rules for the CSP channels in RavenSPARK:

- It must be created a different package for each channel type. The package name is defined by the type of the object transferred by the channel, in other words, if the transferred data is an integer, the channel will be of the type *IntegerChannel*, being created a package with this same name. If it transmit messages, it will be of the type *MessageChannel*, and so on.
- The internal data stored by the protected object *Data* has the same type of the channel.
- Each channel present in the gCSP diagram should be created inside the package that represents its channel type. This way, all the channels that have the same type are created together inside the same package. This is due to restrictions imposed by the RavenSPARK profile and also facilitates the implementation verification process.
- 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*. So, for instance, for an integer channel whose name is *myChannel* in the gCSP diagram, its RavenSPARK implementation will consist of two object creation : *myChannel_d* and *myChannel_s*, both created inside the package *IntegerChannel*.

- The data transmition and reception through the channels will be done by the call of the methods *put*, *get*, *stay* and *proceed*, in the following way:
  - In the task who send the data *someData*, the code for using the channel would be, for example:

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

  - In the task that receives the data and stores it into the variable *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
```

2.3 SPARK Annotations

Based on the CSP-OZ specifications, besides all the code that implements the system, the SPARK annotations are generated for all the components. These annotations will be used in the next stage of the methodology, that is the verification of the implemented code.

The annotations are inserted inside the code in the form of comments who beginn with the symbols "–#". The piece of code above shows an example of these annotations by means of the *Add* procedure. These ones serve as a base for the analyses done by the examiner of the SPARK language. Among the inserted types of comments are pre and post conditions of variables and methods, priorities for tasks and protected objects and specifications of the manners of use of variables (reading or writing) along the program. The details of which annotations should be inserted and their use can be seen in (Barnes, 2006) and (Team, 2006).

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

The main advantage in the use of CSP-OZ together with SPARK it consists in the possibility of pre and post conditions elaboration for methods, as well as the specification of valid values intervals for all the variables used in the implementation during the system modelling. In this way the restrictions done in CSP-OZ would be mapped directly into RavenSPARK annotations, reducing the elaboration time of them during the implementation phase. Another advantage in the use of this formal language consists in the elaboration of data types in Ada based on the formal specification of these types in CSP-OZ, in other words, to use the Ada strong typing to implement the basic types defined in CSP-OZ. These Ada types, that possess predefined valid intervals, they are used in the analyses of the created variables in order to find error prone pieces of code.

2.4 Implementation Checking

Based on the inserted SPARK annotations, the SPARK examiner is capable to do certain verifications in the code. This one has two basic functions:

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

The analysis performed by the examiner is strongly based in the interfaces among the components, assuring that the implementations in all the components are in agreement with the specifications present in their respective interfaces. This way, the SPARK language together with their annotations assure that a program cannot contain certain mistakes related to information flow, once that it detects the use of non initialized variables or the writing of variables before they are used. The dynamic behavior is verified through pre and post conditions annotations, allowing the analyzer to generate theorems that therefore should be proven in order to verify that the program doesn't have errors.

In this work won't be generated any pre and post conditions for the variables and methods, neither in the formal nor in the SPARK annotations. So, any theorems or formal proofs won't be elaborated. It will be explored intensively the strong typing concept, where all the data types used in the implementation will be predefined and specified with their validity intervals. These intervals will be checked for consistency by the examiner to all the variables in the program.

2.5 Tests

After having concluded the modeling and implementation stages, with each stage having its verification phase, they are made the tests in the embedded system. These are the normal tests accomplished in any software development.

In this work the real time operating system VxWorks is being used, and so its analysis tools of the runtime code will be used for the tests. It has a simulation environment where the developed code can be embedded, which has several possible hardware configurations. After configured the hardware, which consists basically of the processor and amount of memory, it is possible to verify in runtime which processes are being executed, how much each one consumes in processing and memory terms, among others.

3 ROV CONTROL ARCHITECTURE

The proposed method was used in the development of a control architecture for an underwater vehicle of the type ROV. It will be embedded into the robot show on the Fig. 1 in future works. Its development was made according to the hybrid paradigm (Murphy, 2000), being therefore divided in two conceptual levels (Fig. 3): a *deliberative* layer, where are the functions related to the robot's decisions, and therefore is the responsible for indicating the next movement of the vehicle, and; a *reactive* layer, where are encapsulate all the protection and movement functions of the robot, together with the processes that interact directly with the embedded sensors. In this work, only the reactive part of the architecture was implemented, that include the *Remote* behaviour. It was developed based on the Subsumption paradigm (Brooks, 1986).

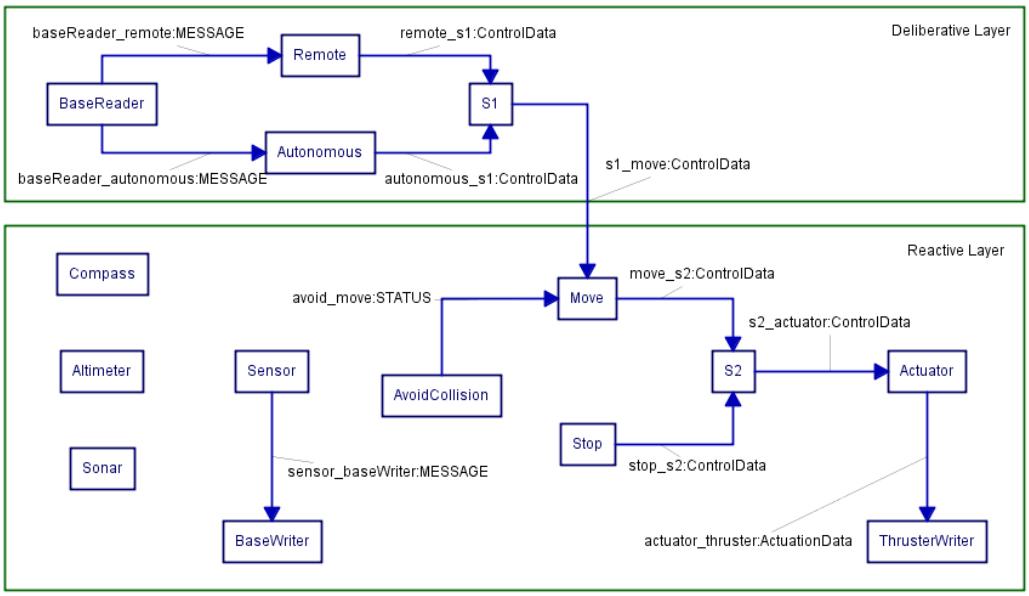


Figure 3. Software Architecture: Processes and Channels.

In this diagram, all the elements represented by a rectangle are processes, which execute in an independent way amongst themselves. The only interaction point among them is made through unidirectional communication channels, represented by the arrows in the diagram. These arrows represent the information flow direction. This directions is also represented into the channel names, which also include the type of the transmitted data, being named always following the rule: *start_end:type*. The processes *Compass*, *Altimeter* and *Sonar* don't possess any communication channels. This is because they don't communicate directly with the other processes of the architecture. They capture the information of their respective sensors and store them in a data structure named *BlackBoard*. This one is used as an area for asynchronous exchange of information.

4 RESULTS

The CSP_M script generated based on the CSP-OZ model was checked in FDR in order to find the common concurrency errors like deadlocks, livelocks and to verify if the system is deterministic. These tests can be made in the system as a whole, considering the interaction of all the processes, as well as in each of the individual processes that compose it.

As can be seen in the figure Fig. 4, the process *ROV_Control_Module* (which represent the whole system) was tested regarding the existence of *deadlocks*, *livelocks* and determinism, being successfull in all of them. Regarding the performance, the FDR accomplished the three programmed analyses in a relatively short time. It is important to emphasize that this system presents a considerable size, what indicates that the proposed modelling method allows that larger systems can be analyzed too. According to the report obtained from the tool, the analyzed system has the following information:

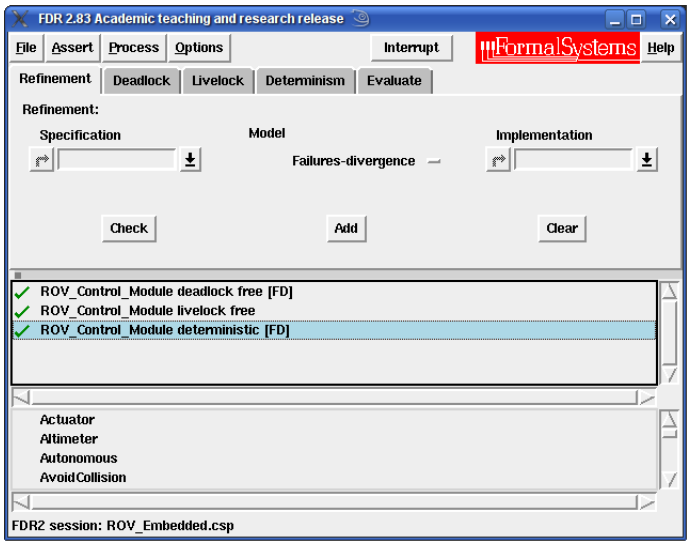


Figure 4. Model Checking in FDR.

Table 1. Architecture Analysis Time on FDR.

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

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

It can be observed that the number of states of the system is quite high, what make impossible a manual analysis of its state machine. Even with that high number of states, as can be observed in the Tab. 1, the time spent with the analyses is relatively short. The analysis of determinism is the longest one, taking a little less than one hour. The others take less than two minutes to be done[1].

All the code developed in RavenSPARK was checked using the SPARK Examiner tool. In this way, it was checked the code consistency in relation to the program information and data flow, overflows, among others. According to the report produced by the tool, the architecure made is free of the checked errors.

After done all the model and implementation checkings, the control architecture was tested with the VxWorks development environment, the Workbench. The compiled code was embedded into a PC104 similar to the one used in the ROV.

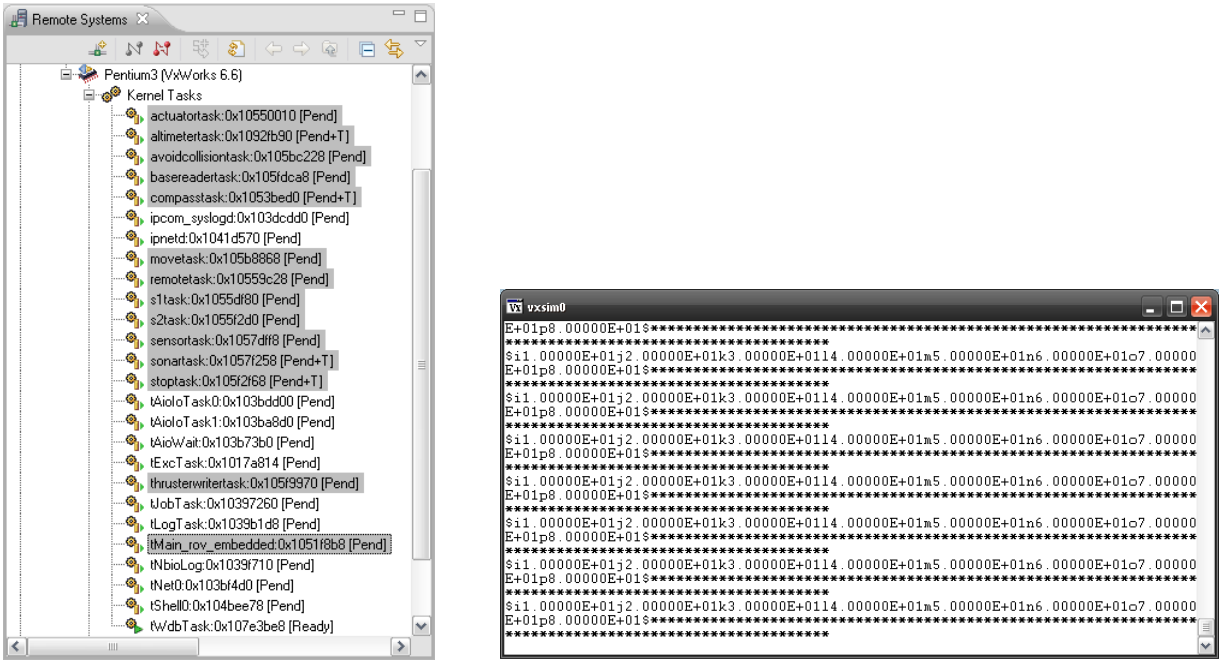
The Fig. 5(a) shows the processes being executed in the VxWorks kernel, being showed also the message sent to the ROV thrusters controller(Fig. 5(b)). With the embedded system several tests can be done. Possible tests include memory analyses, processing, response times of the tasks, among others.

5 CONCLUSIONS

The software development based in the use of formal specifications is a way to reach the correctness by construction, concept difficult of being applied through the project methodologies that don't use formal languages due to difficulties in the verification of the developed models. Based on the CSP-OZ specifications it was possible to perform both model checking and implementation checking. The model checking is made by the model checker FDR based on a script in the CSP_M language. The implementation checking is made by the SPARK Examiner tool, based in the annotations inserted in the code as Ada comments. Great part of these annotations is generated based in the data structure specifications of the processes Object-Z part.

The implementation of the model using restricted subsets of the Ada language used in critical systems like Ravenscar and SPARK results in a more robust, reliable and deterministic implementation, once the code generated according to these patterns is free from a series of dangerous characteristics of the Ada language like pointers, dynamic allocation and other problems related to concurrency and that prejudices its project and analysis. In the case of SPARK, the generation

[1]These analyses were done in a common computer, with a 1.8 GHz processor and 1 GB of memory. So one can conclude that it is not necessary a powerful hardware to execute the analyses proposed in the development method.



(a) Tasks in the Kernel.

(b) Outputs during execution.

Figure 5. Execution in VxWorks.

of annotations and the verification of the code is totally in agreement with the objectives of the work, that it is to look for software correctness by construction, giving more emphasis on the preliminary development of the system, in other words, on the modelling. Another great advantage of the use of RavenSPARK consists in the easiness in the transformation of CSP-OZ models in code, once this programming language possesses the necessary resources for a direct and safe implementation, like tasks, protected objects, mechanisms of task sinchronization, hiding with packages, among others. This made possible the elaboration of the processes and channels creation rules in the method.

Regarding the model checking technique used in this work, it was observed that this presented a satisfactory performance, once all of the analyses accomplished with FDR were concluded in a time relatively short. This is due to the internal data structures of data of the processes were not analyzed in FDR, being these done with the SPARK examiner which also made all the analyses in a quite short time.

6 REFERENCES

Akhlaki, K. B., Tunon, M. I. C., Terriza, J. A. H., and Morales, L. E. M. 2006. Formal Specification of Real-Time Systems by Transformation of UML-RT Design Models. In Barjis, J., Ultes-Nitsche, U., and Augusto, J. C., editors, *MSVVEIS*, pages 16–25. INSTICC Press.

Amey, P. 2002. Correctness By Construction: Better Can Also Be Cheaper. *CrossTalk Magazine, The Journal of Defense Software Engineering*.

Amey, P. and Dobbing, B. 2003. High Integrity Ravenscar. In *Ada-Europe*, pages 68–79.

Atiya, D.-A. and King, S. 2005. Extending Ravenscar with CSP Channels. In *Ada-Europe*, pages 79–90.

Atiya, D. M. 2004. *Verification of Concurrent Safety-critical Systems: The Compliance Notation Approach*. PhD thesis, University of York.

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

Barnes, J. G. 1989. *Programming in Ada*. Addison-Wesley, Wokingham [u.a.], 3 edition.

Brooks, R. 1986. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23.

Burns, A., Dobbing, B., and Vardanega, T. 2004. Guide for the use of the Ada Ravenscar Profile in High Integrity Systems. *Ada Lett.*, XXIV(2):1–74.

Champeau, J., Dhaussy, P., Moitie, R., and Prigent, A. 2000. Object Oriented and Formal Methods for AUV development. *OCEANS 2000 MTS/IEEE Conference and Exhibition*, 1:73–78 vol.1.

de Medeiros, A., Chatila, R., and Fleury, S. 1996. Specification and Validation of a Control Architecture for Autonomous Mobile Robots. *Intelligent Robots and Systems '96, IROS 96, Proceedings of the 1996 IEEE/RSJ International Conference on*, 1:162–169 vol.1.

Dobbing, B. and Burns, A. 1998. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *SIGAda '98: Proceedings of the 1998 annual ACM SIGAda international conference on Ada*, pages 1–6, New York, NY, USA. ACM.

Duke, R. and Rose, G. 2000. *Formal object-oriented specification using object-z.* cornestones of computing. Macmillan.

Fischer, C. 1997. 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*, pages 423–438, London, UK, UK. Chapman & Hall, Ltd.

Fischer, C. 2000. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Universidade de Oldenburg.

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

Fischer, C. and Wehrheim, H. 2000. Failure-divergence Semantics as a Formal Basis for an Object-Oriented Integrated Formal Method. *Bulletin of the European Association for Theoretical Computer Science*, 71:92–.

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

Jovanovic, D. S., Orlic, B., Liet, G. K., and Broenink, J. F. 2004. gCSP: A Graphical Tool for Designing CSP Systems. *Communicating Process Architectures 2004*.

Kassel, G. and Smith, G. 2001. Model Checking Object-Z Classes: Some experiments with FDR. In *Proc. Eighth Asia-Pacific Software Engineering Conference APSEC 2001*, pages 445–452.

Lawrence, J. 2005. Practical Application of CSP and FDR to Software Design. pages 151–174.

Lightfoot, D. 1991. *Formal Specification using Z.* Macmillan.

Mota, A., Farias, A., and Sampaio, A. 2001. De CSPz para CSPm: Uma ferramenta transformacional Java. pages 1–10.

Murphy, R. R. 2000. *Introduction to AI Robotics*. MIT Press, Cambridge, MA, USA.

Ng, M. Y. and Butler, M. 2002. Tool Support for Visualizing CSP in UML. In George, C. and Miao, H., editors, *International Conference on Formal Engineering Methods(ICFEM)*, pages 287–298. Springer Verlag.

OMG 2004. Unified Modeling Language: Superstructure. version 2.0.

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

Ruiz, J. F. 2006. Ada 2005 for Mission-Critical Systems.

Sherif, A., Sampaio, A., and Cavalcante, S. 2003. Specification and Validation of the SACI-1 On-Board Computer Using Timed-CSP-Z and Petri Nets. pages 161–180.

Systems, F. 2005. *Failures-Divergence Refinement: FDR2 User Manual*.

Team, S. 2006. *SPARK Examiner - The SPARK Ravenscar Profile*. Praxis, 1.5 edition.

Telelogic 2009. Rhapsody.

Yeung, W., Leung, K., Wang, J., and Dong, W. 2005. Improvements towards formalizing UML state diagrams in CSP. *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, pages 7 pp.–.

7 Responsibility notice

The authors are the only responsible for the printed material included in this paper.