Fábio Henrique de Assis

Checagem de Arquiteturas de Controle de Veículos Submarinos: uma abordagem baseada em especificações formais

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção de Título de Mestre em Engenharia.

São Paulo
2009

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

Fábio Henrique de Assis

Checagem de Arquiteturas de Controle de Veículos Submarinos: uma abordagem baseada em especificações formais

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção de Título de Mestre em Engenharia.

Área de concentração:
Controle e Automação

orientador: Newton Maruyama

São Paulo
2009

Ficha Catalográfica

Assis, Fábio Henrique de

Checagem de Arquiteturas de Controle de Veículos Submarinos: uma abordagem baseada em especificações formais. / F. H. de Assis – São Paulo, 2009. 144 p.

Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1. Submersíveis não tripulados 2. Arquitetura de Software (Especificação) I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos. II. t.

Dedicatória

à minha família.

Agradecimentos

Gostaria de agradecer primeiramente aos meus pais e ao meu irmão, que sempre me apoiaram em todas as escolhas que tomei em relação à minha carreira, principalmente nos momentos mais difíceis.

Agradeço também à minha namorada por estar sempre comigo, e muitas vezes por compreender minhas ausências que se fizeram necessárias ao longo deste trabalho.

Não poderiam ficar de fora as pessoas que contribuíram de maneira efetiva para que esse trabalho fosse concluído, que são meu orientador e meus colegas de laboratório, além de todos os membros da XMobots, opinando, discutindo e muitas vezes ajudando a desenvolver certas partes do trabalho.

"O único lugar onde sucesso vem antes do trabalho é no dicionário"
Albert Einstein

Resumo

O desenvolvimento de arquiteturas de controle para veículos submarinos é uma tarefa complexa. Estas podem ser caracterizadas pelos seguintes atributos: tempo real, multitarefa, concorrência e comunicações distribuídas em rede. Neste cenário, existem múltiplos processos sendo executados em paralelo, possivelmente distribuídos, e se comunicando uns com os outros. Neste contexto, o modelo comportamental pode levar a fenômenos como *deadlocks*, *livelocks*, disputa por recursos, entre outros. A fim de se tentar minimizar os efeitos de tais dificuldades, neste trabalho será apresentado um método para checagem de modelos de arquiteturas de controle de veículos submarinos baseado em Especificações Formais. A linguagem de especificação formal escolhida foi CSP-OZ, uma combinação de CSP e Object-Z. Object-Z é uma extensão orientada a objetos da linguagem Z para a especificação de predicados, tipicamente pré e pós condições, além de invariantes de dados. CSP (*Communicating Sequential Process*) é uma álgebra de processos desenvolvida para descrever modelos comportamentais de processos paralelos. A checagem de modelos especificados formalmente consiste na análise das especificações para verificar se um sistema possui certas propriedades através de uma busca exaustiva em todos os estados em que este pode entrar durante sua execução. Neste contexto, é possível checar corretude, *livelocks*, *deadlocks*, etc. Além disso, pode-se relacionar duas especificações diferentes a fim de se checar relações de refinamento. Para as especificações, o verificador de modelos FDR da Formal Systems Ltd. será utilizado. A implementação é desenvolvida utilizando um perfil da linguagem Ada denominado RavenSPARK, uma junção do perfil Ravenscar (desenvolvido na Universidade de York) com a linguagem SPARK (um subconjunto da linguagem Ada desenvolvido pela Praxis, Inc.). O Ravenscar é um perfil para desenvolvimento de processos, e portanto os processos de CSP, incluindo seus canais de comunicação, podem ser facilmente criados. Por outro lado, SPARK é uma linguagem onde podem ser inseridos predicados para os dados (originalmente especificados em Object-Z) utilizando anotações da própria linguagem. A linguagem SPARK possui uma ferramenta, o Examinador, que pode checar códigos de modelos baseado nestas anotações. Em resumo, o método proposto permite tanto a checagem de modelos em CSP quanto a checagem no nível de código. Para isso, as especificações em Object-Z devem inicialmente ser convertidas em um código na linguagem SPARK juntamente com suas respectivas anotações, para que então a checagem do modelo possa ser realizada no código. O desenvolvimento de uma arquitetura de controle reativa para um ROV denominado VSOR (Veículo Submarino Operado Remotamente) é utilizado como exemplo de uso do método proposto. Toda a arquitetura de controle é codificada utilizando a linguagem Ada com o perfil RavenSPARK e embarcada em um computador do tipo PC104 com o sistema operacional de tempo real VxWorks, da Windriver, Inc.

Abstract

The development of control architectures for Underwater Vehicles is a complex task. These control architectures might be chracterised 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 behavioural 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 specifications 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 behavioural 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 utilised. 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.

Lista de Figuras

1.1 Exemplos de Veículos Não-Tripulados. . . . . p. 16

1.2 Desempenho entre os diferentes métodos de verificação de modelos (BRAT et al., 2003). . . . . p. 19

1.3 Esquema do Método Proposto. . . . . p. 21

2.1 Máquina de estados da especificação CSP do processo P. . . . . p. 30

2.2 Esquema de funcionamento do paradigma Híbrido. . . . . p. 34

3.1 Método de Desenvolvimento Proposto. . . . . p. 37

3.2 Exemplo de Especificação em gCSP. . . . . p. 40

3.3 Modelo Produtor / Consumidor em gCSP. . . . . p. 50

3.4 Verificação do modelo Produtor - Consumidor no FDR. . . . . . p. 53

3.5 Execução do modelo Produtor / Consumidor no VxWorks. . . . p. 62

4.1 Esquema de monitoramento com ROVs. . . . . p. 64

4.2 Robô utilizado no trabalho. . . . . p. 67

4.3 Interface gráfica de controle do ROV. . . . . p. 69

4.4 Comandos de atuação do ROV. p. 70

4.5 Módulos de Software presentes no Sistema. . . . . p. 71

4.6 Casos de Uso dos Módulos de Software do Sistema. . . . . p. 72

5.1 Arquitetura de Software - Canais e Estruturas de Dados. . . . . p. 75

6.1 Análise da Arquitetura no FDR. . . . . p. 99

6.2 Execução da Arquitetura de Controle no VxWorks. . . . . p. 104

Lista de Tabelas

4.1 Conjunto de sensores embarcados no ROV. . . . . p. 67

4.2 Semântica de Comunicação. . . . . p. 71

6.1 Tempo de Análise da Arquitetura no FDR. . . . . p. 99

Sumário

1 Introdução p. 15

1.1 Objetivos . . . p. 20

1.2 Uma breve apresentação do método proposto . . . p. 20

1.3 Contribuições . . . p. 23

1.4 Organização do Trabalho . . . p. 24

2 Conceitos Básicos p. 25

2.1 Modelagem de Software . . . p. 25

2.2 UML-RT . . . p. 25

2.3 Métodos Formais . . . p. 27

2.3.1 CSP . . . p. 29

2.3.2 Object-Z . . . p. 30

2.3.3 CSP-OZ . . . p. 32

2.4 Arquiteturas de Controle . . . p. 33

2.5 Conclusões do Capítulo . . . p. 36

3 Método de Desenvolvimento p. 37

3.1 Modelagem do Sistema . . . p. 38

3.1.1 Diagrama de Estrutura em CSP . . . p. 38

3.1.2 Especificação dos Componentes em CSP-OZ . . . p. 40

3.1.3 Checagem do Modelo . . . p. 42

3.2 Implementação do Modelo . . . p. 44

3.2.1 Geração de Código . . . p. 45

3.2.2 Anotações SPARK . . . p. 48

3.2.3 Checagem da Implementação . . . . p. 49

3.3 Testes . . . . p. 49

3.4 Exemplo de Aplicação: modelo Produtor/Consumidor . . . . . . p. 50

3.4.1 Modelagem . . . . p. 50

3.4.2 Implementação . . . . p. 54

3.4.3 Testes . . . . p. 61

3.5 Conclusões do Capítulo . . . . p. 61

4 Sistema para Controle de ROVs p. 63

4.1 Descrição Geral . . . . p. 63

4.2 Requisitos . . . . p. 64

4.2.1 Modos de Operação . . . . p. 64

4.2.2 Modos de Movimentação . . . . p. 65

4.2.3 Requisitos de Desempenho . . . . p. 65

4.2.4 Requisitos Temporais . . . . p. 66

4.3 Componentes do Sistema . . . . p. 66

4.3.1 O Robô . . . . p. 66

4.3.2 Estação Base . . . . p. 67

4.4 Arquitetura de Hardware . . . . p. 71

4.5 Conclusões do Capítulo . . . . p. 73

5 Especificação da Arquitetura do Software Embarcado p. 74

5.1 Introdução . . . . p. 74

5.2 Definições Gerais . . . . p. 75

5.2.1 Tipos Básicos de Dados . . . . p. 75

5.2.2 Estruturas de Dados . . . . p. 77

5.2.3 Processos Periódicos . . . . p. 78

5.3 Camada Deliberativa . . . . p. 80

5.3.1 BaseReader . . . . p. 81

5.3.2 Remote . . . . p. 82
5.3.3 Autonomous . . . . p. 82
5.3.4 S1 . . . . p. 83
5.4 Camada Reativa . . . . p. 84
5.4.1 Compass . . . . p. 85
5.4.2 Altimeter . . . . p. 86
5.4.3 Sonar . . . . p. 87
5.4.4 BaseWriter . . . . p. 88
5.4.5 ThrusterWriter . . . . p. 89
5.4.6 Move . . . . p. 90
5.4.7 S2 . . . . p. 92
5.4.8 Stop . . . . p. 92
5.4.9 Sensor . . . . p. 93
5.4.10 AvoidCollision . . . . p. 94
5.4.11 Actuator . . . . p. 95
5.5 Conclusões do Capítulo . . . . p. 96

6 Resultados p. 98
6.1 Verificação do Modelo . . . . p. 98
6.2 Verificação da Implementação . . . . p. 100
6.3 Testes no Sistema Embarcado . . . . p. 104

7 Conclusões p. 106

Referências p. 108

Apêndice A – Notação de Object-Z e CSP p. 112
A.1 Notação da linguagem Object-Z . . . . p. 112
A.2 A notação da linguagem CSP . . . . p. 115

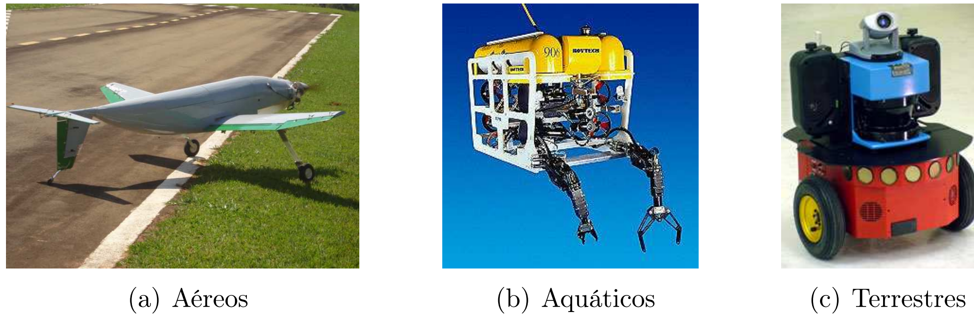
Anexo A – Script $\mathbf{CSP_M}$ da Arquitetura de Controle p. 118

1 Introdução

Atualmente, sistemas embarcados estão presentes em equipamentos com os quais temos contato direto em nosso dia a dia, como aparelhos eletro-eletrônicos, automóveis, sistemas de controle de temperatura de ambientes, entre outros. Além destas aplicações mais comuns, eles também são encontrados em sistemas onde é necessário haver maior confiabilidade e determinismo no comportamento do sistema. Isso porque uma falha em tais sistemas pode implicar em grandes prejuízos financeiros ou até mesmo na perda de vidas. Alguns exemplos são os softwares de controle de trens, aeronaves, plantas de usinas nucleares, entre outros. Sistemas deste tipo são denominados *Sistemas Críticos* ou *Sistemas de Alta Integridade*[1].

No contexto deste trabalho, os sistemas embarcados são aplicados em robôs móveis, também conhecidos como Veículos Não-Tripulados. Estes são veículos capazes de operar em diversos meios como ar (UAVs[2]), terra (UGVs[3].) e água (AUVs[4] e ROVs[5]). A figura 1.1(a) mostra um exemplo de Veículo Aéreo Não-Tripulado (VANT), o Apoena, da empresa brasileira XMobots (XMOBOTS, 2009). Na figura 1.1(b) temos um exemplo de veículo submarino do tipo ROV que apresenta garras atuadoras, o Panther Plus, da empresa inglesa Saab Seaeye (SAAB, 2009). E por fim, a figura 1.1(c) mostra o robô terrestre Pioneer 2DX, da empresa Mobile Robots (MOBILE, 2009). A característica marcante destes veículos consiste na sua capacidade de operar sem a presença de um piloto em seu interior. Existem dois modos básicos de operação: *remoto*, onde o piloto envia comandos de atuação para o robô por meio de um canal de comunicação, que pode ser um cabo, um link de rádio, entre outros; *autônomo*, onde o veículo opera sem nenhum tipo de intervenção humana. Dos tipos de veículos mencionados acima, neste trabalho será utilizado um submarino do tipo ROV, operando em modo remoto por meio de uma estação de controle.

[1]em inglês estes sistemas são denominados *Safety-Critical* ou *High Integrity Systems*.
[2]do inglês, *Unmanned Aerial Vehicles*.
[3]do inglês, *Unmanned Grounded Vehicles*
[4]do inglês, *Autonomous Underwater Vehicles*.
[5]do inglês, *Remotely Operated Vehicles*.



(a) Aéreos (b) Aquáticos (c) Terrestres

Figura 1.1: Exemplos de Veículos Não-Tripulados.

Do ponto de vista do software de controle destes veículos, estes podem ser considerados Sistemas de Tempo Real. Segundo Burns (2001), um sistema de tempo real é qualquer sistema de processamento de informações que deve responder a estímulos de entrada gerados externamente em um dado período de tempo finito e previamente especificado. Nestes sistemas, a corretude não depende apenas do resultado lógico, mas também do tempo em que ele foi obtido. Logo, um atraso na resposta pode ser tão ruim quanto uma resposta errada. Estes podem ser classificados em quatro categorias quanto sua tolerância à falhas:

Hard Real Time Systems: são sistemas onde é imprescindível que as respostas ocorram dentro dos limites de tempo pré-estabelecidos. Nesse caso, os erros precisam ser todos rastreados de forma a permitir uma ação imediata de tratamento dos mesmos. Um exemplo de sistema de tempo real crítico são os sistemas de controle de vôo;

Soft Real Time Systems: sistemas onde os limites de tempo são importantes, mas que ainda continuam funcionando caso algum destes limites seja rompido ocasionalmente. Um exemplo disso são sistemas de aquisição de dados;

Real Real Time Systems: são sistemas que são *hard real time* e que possuem tempos de resposta muito curtos. Um exemplo seria um sistema de controle de mísseis;

Firm Real Time Systems: o sistema de tempo real firme, ou *firm real time* é aquele no qual tarefas atrasadas não são aproveitáveis, mas ainda funcionam corretamente caso alguns prazos ou processos sejam ocasionalmente perdidos. Sistemas assim costumam ser utilizados em aplicações multimídia ou de realidade virtual que requerem características avançadas de sistemas operacionais.

Dada a classificação acima, a arquitetura de controle desenvolvida neste tra-

balho será considerada um sistema *soft real time*, uma vez que não serão feitas provas nem análises quanto a restrições temporais. Entretanto, tais aspectos farão parte do projeto.

O desenvolvimento de arquiteturas de controle para veículos submarinos é uma tarefa complexa. Além do fato destas se caracterizarem como sistemas de tempo real, existem ainda outras características que devem ser mencionadas que fazem parte de seu projeto, dificultando-o consideravelmente. A principal delas é a presença de concorrência. Os softwares de controle de robôs móveis frequentemente apresentam concorrência, onde o sistema de controle é composto por diversos processos sendo executados em paralelo, interagindo uns com os outros e disputando recursos de hardware como regiões de memória compartilhada e tempo de processamento. O aspecto crucial que torna os sistemas concorrentes diferentes dos seqüenciais é o fato de seus processos se comunicarem uns com os outros. O projeto de tais sistemas requer modos de se lidar com a complexidade inerente destas comunicações. Isso porque a concorrência por si só introduz fenômenos que não estão presentes em sistemas sequenciais, como (BELAPURKAR, 2005):

- *Deadlock*: é a condição onde um processo fica bloqueado indefinidamente por estar esperando que uma determinada condição se torne verdadeira, tal como um recurso tornar-se disponível, mas tal condição se tornar verdadeira depende deste mesmo processo que estava esperando fazer algo. Assim, dois ou mais processos ficam esperando os outros darem o primeiro passo, sendo que nenhum é capaz de fazer nada;

- *Livelock*: quando um programa entra em um *loop* infinito e não interage mais com o ambiente, este comportamento é denominado divergência. Sistemas paralelos apresentam *livelock*, que simboliza uma atividade interna que é executada infinitamente, impedindo que o sistema interaja com o ambiente. Do ponto de vista de um usuário, um programa em *livelock* é similar a um programa em *deadlock*. Porém, o primeiro pode ser pior, pois o usuário é capaz de observar alguma atividade interna, e em razão disso esperar eternamente por uma resposta do sistema(ROSCOE; HOARE; BIRD, 1997).

- *Disputa por recursos*: diz-se que existe uma disputa por recursos em um sistema quando existe um recurso compartilhado entre múltiplos processos e o vencedor determina o comportamento do sistema. Um problema resultante disso, onde a principal causa é a sincronização incorreta de acessos a

um recurso compartilhado, consiste na corrupção dos dados compartilhados, onde um processo faz a leitura do dado antes que um outro termine de atualizá-lo.

- *Starvation*: pode ocorrer em sistemas concorrentes em que os processos possuem prioridade de execução. O fenômeno consiste em um processo de baixa prioridade nunca ser executado devido à execução dos outros de maior prioridade.

Estes problemas não surgem a partir do comportamento dos componentes individuais, e sim através do modo como eles interagem, não aparecendo durante o desenvolvimento e nos testes individuais dos componentes de um sistema. Uma teoria de concorrência como o CSP (ROSCOE; HOARE; BIRD, 1997), que será apresentada posteriormente, oferece um modo de controlar tais problemas (SCHNEIDER, 1999). Outra característica relevante consiste na dificuldade na depuração de erros no sistema desenvolvido quando este já está embarcado no robô. Neste caso, a depuração de eventuais erros é uma tarefa difícil de ser realizada, uma vez que em geral não existem muitos recursos para se monitorar os estados do sistema já operando embarcado. Por exemplo, caso haja alguma falha no software que cause uma parada repentina do veículo, será muito difícil identificar a causa do problema para que o erro seja consertado.

Como uma tentativa de se minimizar as dificuldades apresentadas anteriormente, vêm sendo proposto o uso de métodos formais no desenvolvimento de software. Estes consistem basicamente no uso de linguagem matemática na especificação de software, e serão apresentados com maiores detalhes na seção 2.3. Seu uso é mais comum na área de sistemas críticos, mas também existem aplicações na indústria. Um exemplo disso é a utilização de métodos formais para especificação e validação de sistemas de tempo real apresentado por Sherif, Sampaio e Cavalcante (2001). Esta abordagem foi aplicada na especificação do computador de bordo do primeiro satélite para aplicações científicas (SACI-I), do Instituto Nacional de Pesquisas Espaciais - INPE (SHERIF; SAMPAIO; CAVALCANTE, 2003). Outro exemplo de aplicação de métodos formais na indústria é o trabalho de Lawrence (2005), onde é mostrada a aplicação de CSP em um sistema desenvolvido na empresa IBM. Neste caso, a parte especificada formalmente (em CSP) foi posteriormente validada no checador de modelos FDR (SYSTEMS, 2005), e depois exaustivamente testada tanto pelo fabricante quanto pelo cliente. Após a realização dos testes não foram encontrados erros, o que mostra a importância da utilização de especificações formais no desenvolvimento de software.

Apesar de existirem diversas aplicações de especificações formais, a linguagem de modelagem mais utilizada na indústria é o UML (*Unified Modeling Language*) (OMG, 2004). Esta se tornou um dos padrões para modelagem e projeto de sistemas de software mais utilizados devido principalmente a sua notação semi-formal ser relativamente fácil de se utilizar e ser bem suportada por ferramentas de desenvolvimento (AKHLAKI et al., 2006). Entretanto, existem diversos trabalhos que visam formalizar partes da linguagem, como é o caso, por exemplo, dos trabalhos de Akhlaki et al. (2006), Yeung et al. (2005) e Ng e Butler (2002), que utilizam regras de mapeamento de diagramas em UML para gerar especificações formais em diversas linguagens, como Z (LIGHTFOOT, 1991), OhCircus (BORGES; MOTA, 2007), CSP-OZ (FISCHER, 1997), entre outras. Uma outra abordagem desse tipo é a do trabalho de Moller et al. (2008), onde foi proposta uma metodologia utilizando modelagem em UML-RT, geração de especificações em CSP-OZ e implementação em Java.

No desenvolvimento de arquiteturas de controle de robôs móveis, foram encontrados apenas dois trabalhos que utilizam métodos formais. O primeiro deles é o de Medeiros, Chatila e Fleury (1996), que descreve a especificação de uma arquitetura de controle para robôs móveis autônomos, sendo verificados a existência de *deadlocks*, tempos de reação e detecção de inconsistências. O outro é o trabalho de Champeau et al. (2000), que utiliza métodos formais e orientação a objetos no desenvolvimento do software embarcado de AUVs. O objetivo é analisar as possibilidades e limitações de um desenvolvimento conjunto com orientação a objetos e especificações formais.

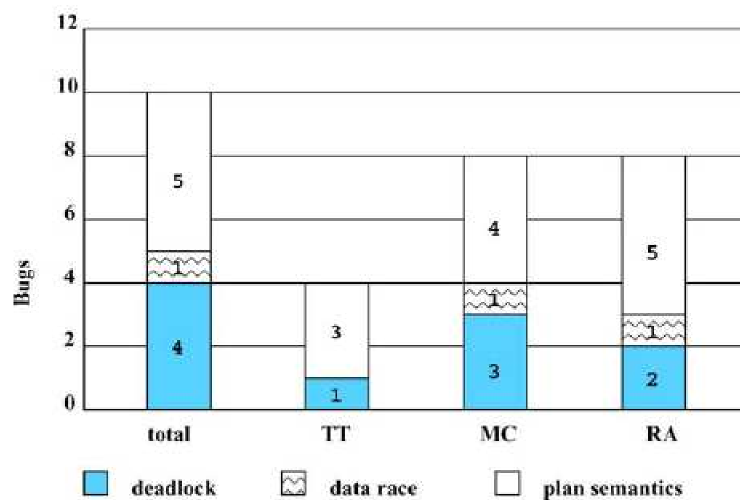


Figura 1.2: Desempenho entre os diferentes métodos de verificação de modelos (BRAT et al., 2003).

A principal vantagem do uso de especificações formais é a possibilidade de

checagem dos modelos antes de sua implementação. O trabalho de Brat et al. (2003) mostra um estudo experimental para determinar o grau de maturidade de três diferentes tecnologias de verificação e validação comparadas com o tradicional teste para encontrar erros em um exemplo representativo, que foi o software de um robô enviado ao planeta Marte pela NASA, a agência espacial americana. A figura 1.2 mostra um dos resultados da pesquisa, mostrando a quantidade de erros (*bugs*) encontrados com testes (TT), checagem de modelos (MC - *Model Checking*) e análise em tempo de execução (RT - *Run-Time Analysis*), para um total de dez falhas conhecidas inseridas no sistema. Podemos observar que de todos eles, os testes foram os menos efetivos na captura de erros no sistema, principalmente os relacionados aos aspectos de concorrência, que são os *deadlocks* e as disputas por recursos (*race conditions*). Neste trabalho o método proposto permite que sejam utilizadas as duas outras técnicas mais efetivas: o *Model Checking*, feito a partir do modelo em CSP e a análise em tempo de execução (*Run-Time Analysis*), podendo ser feita tanto com a ajuda da ferramenta da linguagem SPARK que será apresentada posteriormente ou então na fase de testes, com as ferramentas do sistema operacional de tempo real VxWorks.

1.1 Objetivos

Este trabalho tem por objetivo principal estabelecer um método para desenvolvimento de arquiteturas de controle para veículos submarinos utilizando especificações formais. O método deve apoiar o desenvolvimento da fase de especificação do projeto até a geração de código para sistemas embarcados.

1.2 Uma breve apresentação do método proposto

O ponto crucial de um possível método para a verificação formal de sistemas é a escolha da linguagem de especificação. Neste trabalho, opta-se por CSP-OZ (FISCHER, 1997). Nesta linguagem, a especificação do modelo pode ser dividida em duas partes: uma de processos (CSP) e outra de dados (Object-Z). Em CSP-OZ, ambas as partes estão interconectadas. A parte Object-Z é transformada em um processo CSP. Desta forma, eventos da parte CSP aparecem como chamadas de métodos da parte Object-Z. Pré condições atuam como guardas, i.e., evitando ou permitindo a operação específica. Pós-condições definem para qual estado o processo vai transitar. Uma conseqüência deste esquema é que tanto o refinamento da parte CSP quanto da parte Object-Z estão unificadas pela se-

mântica de falhas e divergências (FISCHER; WEHRHEIM, 2000). Esta particular configuração permite que se possa verificar separadamente a parte CSP da parte Object-Z. Neste trabalho não serão elaboradas pré e pós condições para os métodos, e portanto não serão feitas provas formais. As checagens na implementação serão feitas com base nos tipos de dados definidos nas especificações.

Deste modo foi proposto um método de desenvolvimento, cujo esquema pode ser visto na figura 1.3, e que será melhor descrito no capítulo 3. Tal método deve permitir a especificação dos diversos processos que compõem uma arquitetura de controle, além de suas estruturas de dados internas e modos de comunicação. CSP define os processos e seus modos de interação por meio de canais de comunicação síncronos e unidirecionais, e portanto o método proposto deve permitir a transformação tanto dos processos quanto dos canais de comunicação de CSP para código de um modo simples.

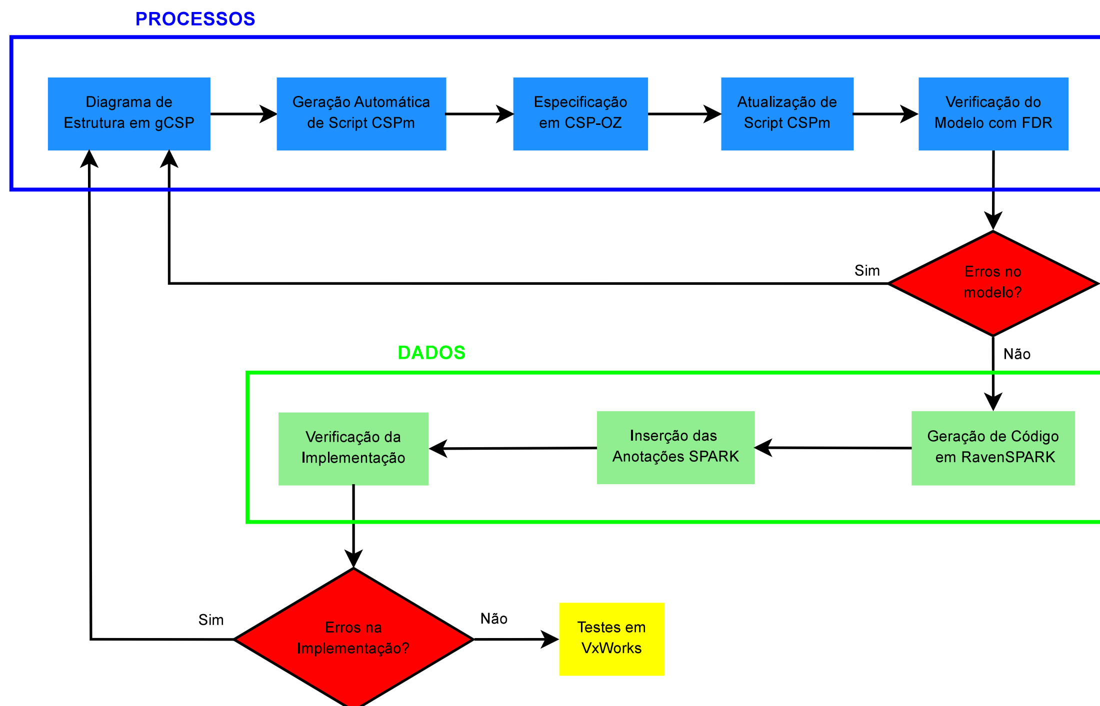


Figura 1.3: Esquema do Método Proposto.

De acordo com o esquema mostrado na figura 1.3, o desenvolvimento possui dois focos principais, que são cobertos em etapas separadas. Primeiramente o modelo é desenvolvido com foco na parte de processos, para posteriormente este passar para a parte de dados do sistema. Sendo assim, são realizadas duas checagens ao longo do desenvolvimento da arquitetura de controle: uma relacionada à parte de processos (CSP) e outra à parte de dados (Object-Z).

A primeira das checagens é feita no modelo em CSP com base em um *script* gerado automaticamente pela ferramenta gCSP (JOVANOVIC et al., 2004) e poste-

riormente editado manualmente a fim de levar em conta a especificação completa do sistema em CSP-OZ. A checagem do *script* é feita com a ferramenta FDR, da Formal Systems (SYSTEMS, 2005), identificando possíveis erros no modelo como presença de *deadlocks*, *livelocks* e indeterminismos. Assim, após feita a checagem do modelo, caso este não apresente erros, passa-se à sua implementação. Neste caso, tanto o código quanto as anotações SPARK (que são usadas na verificação da implementação) são gerados manualmente com base na parte Object-Z da especificação em CSP-OZ. Isso permite que seja feita a segunda das checagens utilizando o examinador da linguagem SPARK, denominado *SPARK examiner*, que verifica se as anotações estão de acordo com o código fonte da implementação, além verificar a existência de erros no fluxo de dados do programa.

Por fim, depois de checados o modelo e sua implementação, o código gerado pode então ser compilado e testado no VxWorks, onde são realizados os testes com o código em execução.

Como um exemplo real de aplicação do método proposto, foi desenvolvida uma arquitetura de controle para um veículo submarino operado remotamente (ROV), que será futuramente embarcada no robô descrito na seção 4.3.1. Como será detalhado na seção 2.4, esta foi desenvolvida com base no paradigma híbrido (MURPHY, 2000), possuindo tanto uma camada deliberativa que inclui as funções de comando do robô quanto uma camada reativa, onde estão encapsuladas as funções básicas de funcionamento do robô como leitura de sensores, comunicação e atuação. Apesar de ter sido colocado na arquitetura com o objetivo de implementações futuras, o modo autônomo de operação do robô não será implementado, e portanto o foco deste desenvolvimento se concentrou no modo remoto de operação. Um ponto importante a se ressaltar é que apesar de terem sido levados em conta na implementação da arquitetura, os aspectos temporais do sistema não foram levados em conta nas especificações em CSP-OZ.

De um modo geral, para esta arquitetura foram feitos:

- toda a modelagem do sistema utilizando a linguagem formal CSP-OZ, incluindo a estrutura do sistema com seus diversos processos e canais de comunicação. Foram especificadas também todos os tipos básicos de dados e também as estruturas de dados utilizadas pelos processos;

- checagem do modelo em CSP-OZ com o FDR a fim de se verificar presença de *deadlocks, livelocks* e determinismo do modelo;

- implementação do modelo em CSP-OZ utilizando a linguagem de progra-

mação SPARK Ada em conjunto com o padrão Ravenscar;

- checagem da implementação por meio do examinador da linguagem SPARK, a fim de verificar a consistência das anotações com o código fonte, além de verificar erros no fluxo de dados do programa;
- testes do sistema embarcado em VxWorks.

1.3 Contribuições

As seguintes contribuições são postuladas neste trabalho:

1. Utilização da linguagem CSP-OZ para especificação de arquiteturas de controle de robôs submarinos. Conforme afirmado anteriormente, não existem muitos trabalhos que utilizam métodos formais na área de robótica. Neste sentido este trabalho apresenta uma contribuição ao utilizar uma linguagem formal na modelagem do sistema. Esta linguagem permite que sejam modeladas tanto a parte de processos quanto a de dados das arquiteturas de controle.
2. Utilização combinada de ferramentas de checagem de modelos: FDR para as especificações da parte CSP e SPARK para as especificações da parte Object-Z, sendo estas últimas verificadas no nível de código.
3. Implementação em RavenSPARK dos canais de comunicação CSP. A implementação utilizada neste trabalho foi adaptada dos trabalhos de Atiya (2004) e Atiya e King (2005), uma vez que nestes trabalhos a implementação dos canais leva em conta apenas o padrão Ravenscar, sendo utilizadas construções proíbidas na linguagem SPARK como tipos de dados genéricos. Estes tiveram de ser removidos, sendo necessárias algumas adaptações no código gerado a fim de permitir o uso da linguagem SPARK, além de facilitar a verificação do código desenvolvido.
4. Utilização de padrões da linguagem Ada específicos para uso em sistemas críticos, que são a linguagem SPARK e o perfil Ravenscar. Isso faz com que o código gerado seja mais robusto e determinístico, uma vez que estes padrões restringem a utilização de diversos recursos da linguagem Ada que podem levar a erros de difícil detecção, principalmente os relacionados aos aspectos de concorrência no sistema.

1.4 Organização do Trabalho

No capítulo a seguir serão apresentados alguns conceitos básicos utilizados neste trabalho, como por exemplo uma breve descrição sobre métodos formais e a linguagem CSP-OZ.

O capítulo 3 apresenta uma descrição detalhada do método de desenvolvimento proposto, finalizando com um exemplo completo de sua aplicação por meio de um modelo simplificado de Produtor / Consumidor.

Em seguida, no capítulo 4 pode ser vista uma descrição completa do sistema de controle de ROVs em desenvolvimento, que inclui o veículo submarino e sua estação de controle.

Por fim, temos os resultados e as conclusões deste trabalho nos capítulos 6 e 7, respectivamente.

No Anexo A pode ser visto o *script* em CSP_{M} utilizado na verificação do modelo da arquitetura de controle na ferramenta FDR, enquanto que no Anexo B podem ser vistos todos os códigos fontes em RavenSPARK da implementação da arquitetura, que foram verificados utilizando o examinador da linguagem SPARK e posteriormente compilados e embarcados num computador PC104 executando o sistema operacional VxWorks.

Um resumo das notações das linguagens CSP e Object-Z pode ser visto no Apêndice A.

2 Conceitos Básicos

Neste capítulo serão apresentados alguns conceitos utilizados neste trabalho. O início da discussão estará relacionado à modelagem de software, onde serão brevemente descritas as linguagens de modelagem utilizadas incluindo CSP, Object-Z e sua combinação: CSP-OZ. Também será apresentada superficialmente a UML-RT, que atualmente é a linguagem mais utilizada na modelagem de software pela indústria. Por fim, será apresentado um breve resumo sobre arquiteturas de controle, de modo a elucidar algumas das escolhas de projeto adotadas na arquitetura desenvolvida no capítulo 5.

2.1 Modelagem de Software

No desenvolvimento de projetos de software, certamente a modelagem representa a parte mais importante. Isso porque ela auxilia no desenvolvimento da estrutura do sistema, o que possibilita a criação de projetos mais robustos, expansíveis e de fácil manutenção. Entretanto, em muitos projetos não são utilizados nenhuma técnica de modelagem ou de análise do sistema em desenvolvimento, cabendo ao programador ir construindo os componentes do sistema à medida que as necessidades vão aparecendo. Tal abordagem é conhecida na literatura como *"bottom-up"*. Neste trabalho será utilizada uma abordagem inversa, onde primeiramente são especificados todos os componentes do sistema, e também os seus modos de interação, para só depois ser feita sua implementação por meio de uma linguagem de programação. Portanto, será dada grande ênfase na fase de modelagem do software.

2.2 UML-RT

Atualmente, a UML (*Unified Modelling Language*) é a linguagem para especificação de modelos mais utilizada na indústria. UML-RT (UML for Real Time Systems) é uma extensão da UML projetada para descrever arquiteturas

de sistemas embarcados (AKHLAKI et al., 2006). Apesar do nome *Real Time* sugerir tempo, o foco desta extensão consiste na modelagem da estrutura de sistemas distribuídos, e não na inserção de facilidades para a inserção do conceito de tempo nas especificações.

UML-RT define três construções para a modelagem da estrutura de sistemas distribuídos, que são (FISCHER; OLDEROG; WEHRHEIM, 2001):

Cápsulas Descrevem componentes complexos do sistema que podem interagir com o ambiente. Elas podem ser estruturadas hierarquicamente, agrupando diversas subcápsulas que por sua vez também podem conter outras subcápsulas.

Portas A interação das cápsulas com o ambiente é feita por meio de portas. Estas geralmente estão associadas a *protocolos*, que determina o fluxo de informação através das portas.

Conectores Estes são usados para conectar duas ou mais portas de cápsulas, e assim descrever as relações de comunicação entre as cápsulas do sistema.

Com estes elementos, as diversas partes do sistema podem ser projetadas de forma independente, contanto que cada parte respeite as interfaces de comunicação existentes nos protocolos.

A utilização de UML-RT no desenvolvimento de sistemas embarcados apresenta algumas vantagens. A mais notável delas é a utilização de uma notação gráfica e amplamente difundida para a especificação de modelos de software. Entretanto, sua principal desvantagem é a falta de uma semântica precisa, o que abre espaço para ambiguidades na interpretação dos modelos por parte dos programadores, e também dificulta a elaboração de provas sobre o comportamento do sistema. Para sanar essa dificuldade foi desenvolvida a OCL (OMG, 2003), que permite a inserção de invariantes e pré e pós condições de métodos nas classes.

Além da utilização da OCL, pode ser feita uma combinação da UML com outras linguagens formais para a eliminação de ambiguidades. Segundo Borges e Mota (2007), existem diversos trabalhos na área de métodos formais relacionados à sua integração com UML. Um exemplo destes trabalhos é o de Polido (2007), que propõe regras de refinamento entre especificações feitas em UML-RT para especificações em CSP-OZ. Do ponto de vista da Engenharia de Software, a integração de métodos formais no processo de desenvolvimento propicia aos diagramas UML uma semântica precisa, o que possibilita sua verificação, e também

estabelece uma relação entre o modelo gráfico de mais alto nível e a implementação final. Isso porque as especificações formais permitem que sejam geradas condições de verificação das especificações do modelo antes de sua implementação. Do ponto de vista dos métodos formais, a vantagem de uma combinação com UML é a possibilidade de se especificar graficamente algumas da características comportamentais, estruturais e de orientação a objetos de um sistema em conjunto com as notações matemáticas. A especificação formal pode então ser obtida parcialmente com base nos diagramas UML por meio de regras de mapeamento, de modo a complementar a especificação do modelo desenvolvido. Isso poderia aumentar a aceitação do uso de uma linguagem de especificação formal (MOLLER et al., 2008) em áreas onde seu uso ainda é incomum, como por exemplo no desenvolvimento de software embarcado para robôs submarinos.

Neste trabalho não será utilizado esse tipo de abordagem. A modelagem do sistema será feita utilizando-se métodos formais, mais precisamente utilizando a linguagem CSP-OZ, cabendo à UML apenas os diagramas de casos de uso na elaboração dos requisitos do sistema.

2.3 Métodos Formais

Técnicas que utilizam princípios matemáticos para desenvolver sistemas computacionais são denominados de modo geral como *Métodos Formais*. A idéia de se especificar o que um sistema computacional deve fazer utilizando notação e técnicas de manipulação matemática é o que se denomina especificação formal (LIGHTFOOT, 1991). Uma especificação formal é uma descrição de um software ou hardware, que pode ser utilizada tanto para uma implementação quanto para a realização de testes. A especificação é diretamente focada no aspecto comportamental do programa, ou seja, no "o que" o programa faz, e não no aspecto operacional, que seria o "como" o programa é implementado. As expressões matemáticas têm a vantagem de serem precisas e não ambíguas, o que elimina confusões e a necessidade de discussões sobre o significado de uma especificação. Outra grande vantagem consiste na concisão das expressões matemáticas, o que se torna relevante na especificação de sistemas de maior tamanho e complexidade (LIGHTFOOT, 1991).

A idéia do uso de especificações formais é que seja garantida a corretude do sistema por construção, ou seja, que sejam encontrados e corrigidos erros mais comuns já na fase de projeto, sendo que esta corretude é garantida através de formalismo matemático. Isso minimizaria o problema de se depurar erros no

software que está em uso no sistema embarcado, uma vez que segundo Hall e Chapman (2002), a maior parte deles já seria identificada e resolvida no início do projeto. Esta identificação precoce de erros também implica em menores custos de projeto, uma vez que torna-se bem mais complicado corrigí-los à medida que o projeto vai avançando (BOWEN; HINCHEY, 2005; HALL; CHAPMAN, 2002; AMEY, 2002). Existem diversas linguagens de especificação formal, como Z, Object Z, VDM, B, entre outras. Neste trabalho será utilizada a linguagem CSP-OZ, que é uma união da álgebra de processos CSP com a linguagem Object Z, e que será apresentada na seção 2.3.3. Um aspecto considerável na utilização desta linguagem é o fato de Z e CSP serem dois dos formalismos mais utilizados na indústria (BORGES; MOTA, 2007).

A principal vantagem do uso de métodos formais na especificação de software consiste na capacidade de se analisar a especificação, com o objetivo de se assegurar da presença e/ou ausência de propriedades em seu comportamento (SHERIF; SAMPAIO; CAVALCANTE, 2001). Uma abordagem para se alcançar este objetivo é utilizar ferramentas para checagem de modelos[1], que permitem comparar automaticamente os comportamentos observados em um sistema com relação à sua especificação. No caso deste trabalho, a interação entre os processos que constituem o sistema será verificada formalmente com a utilização da ferramenta FDR (SYSTEMS, 2005), da Formal Systems.

Conforme mostrado nos trabalho de Brooks (1987) e Hall e Chapman (2002), devido às dificuldades inerentes ao processo de desenvolvimento de software, torna-se impossível garantir a ausência de erros em qualquer projeto por meio de testes. No caso de sistemas embarcados, este problema apresenta um agravante devido à dificuldade de realizar a detecção e depuração de erros no caso de uma falha. É neste sentido que vem sendo proposto o desenvolvimento de software tendo como ferramenta o uso de métodos formais, visando auxiliar na verificação e validação de modelos para a eliminação de erros na especificação do software, pois aproximadamente 80% dos erros de software são gerados nessa fase de projeto (BOWEN; HINCHEY, 2005).

Apesar de todas as vantagens apresentadas, o seu emprego apresenta um custo elevado. Uma das razões é que para sua utilização é necessário haver mão de obra altamente especializada, o que aumenta os custos de projeto. O desenvolvimento das especificações também aumenta consideravelmente a fase de modelagem do sistema, o que acaba atrasando as demais fases do projeto. Porém, com um uso correto, o emprego de métodos formais pode reduzir consideravelmente o período

[1] em inglês, essa técnica é conhecida como *Model Checking*.

de testes.

Já existe um consenso na comunidade de software de que a especificação é essencial. Porém, em qual grau? A resposta depende de quão crítico é o sistema (ATIYA, 2004). Na área de Sistemas Críticos, existem normas que visam a regulamentação do modo com que os softwares são desenvolvidos. Um exemplo destas é a DO-178B (RTCA, 1992), utilizada no desenvolvimento de softwares aviônicos. Segundo ela, para se atingir os níveis mais críticos de confiabilidade, é recomendado o uso de métodos formais no desenvolvimento do software. Já pela norma inglesa DEF STAN 00-55, o uso de métodos formais é mandatório tanto no projeto do sistema quanto como complemento aos testes e análises estáticas (BOWEN; HINCHEY, 2005).

2.3.1 CSP

CSP (*Communicating Sequential Process*) (ROSCOE; HOARE; BIRD, 1997) é uma álgebra de processos que pode ser utilizada como método formal de especificação de concorrência e dinâmica de sistemas. Ela apresenta um conjunto básico de operadores que permitem modelar diversos aspectos presentes em sistemas concorrentes como paralelismo, não determinismo, sincronização de processos, entre outras coisas. Segundo Schneider (1999), esta abordagem tem sido empregada na especificação, análise e verificação de sistemas concorrentes de tempo real, atuando como ferramenta para lidar com os problemas que podem surgir com a presença de concorrência.

O conceito básico adotado em CSP é considerar os processos como entidades independentes, com interfaces particulares sobre as quais eles podem interagir com o ambiente. Com isso é possível também formar composições, pois se dois processos são combinados para formar um sistema maior, novamente este sistema é considerado independente e com uma interface maior, ou seja, um processo maior. A interface de um processo é representada pelo conjunto de eventos que podem ser recebidos ou enviados. Tomemos como exemplo um processo P, cuja interface seria composta pelos eventos $\{a, b, c\}$. Sua especificação em CSP poderia ser $P = a \rightarrow b \rightarrow c \rightarrow P$, onde o processo ficaria em loop infinito. Sua máquina de estados pode ser vista na figura 2.1.

A principal vantagem do uso de CSP é a possibilidade de verificação do modelo desenvolvido por meio da ferramenta de checagem de modelos FDR (SYSTEMS, 2005). Segundo Lawrence (2005), a verificação de projetos com o FDR em certos casos específicos onde o sistema é pequeno pode resultar em um nível considerável

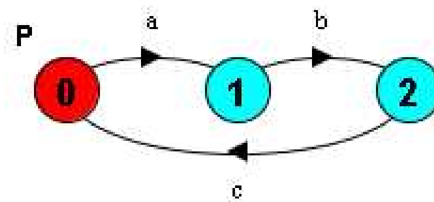


Figura 2.1: Máquina de estados da especificação CSP do processo P.

de confiança na corretude dos mesmos, mas não se pode provar essa corretude se o sistema aumenta muito em tamanho devido à limitações na capacidade de análise da ferramenta. Como neste trabalho não serão feitas análises na estrutura de dados do sistema com o FDR, este problema de explosão no número de estados não ocorrerá. Como serão feitas apenas checagens com relação à parte dinâmica do sistema, ou seja, na comunicação entre os processos, o número de estados é relativamente pequeno e perfeitamente tratável pela ferramenta.

2.3.2 Object-Z

Object Z é baseado na linguagem de especificação formal Z (LIGHTFOOT, 1991), e provê construções específicas para facilitar especificações em um estilo orientado a objetos (DUKE; ROSE, 2000). Nesta linguagem, cada classe pode ser examinada e entendida isoladamente, e classes complexas podem ser especificadas em termos de outras mais simples através do mecanismo de herança da programação orientada a objetos. O objetivo de sua utilização é a possibilidade de se especificar as estruturas de dados do sistema por meio de uma linguagem de alto nível, sem se preocupar inicialmente com detalhes de implementação.

A estrutura básica de uma classe em Object-Z é:

Nome

definição de tipos e constantes

esquema de estados

esquema de inicialização de estados

esquema de operações

Para ilustrar o uso da linguagem, será mostrada a especificação de uma pilha capaz de armazenar elementos do tipo genérico T, com capacidade máxima indicada pela constante *size* (especificação 2.3.2). Logo, esta pilha possui uma capacidade máxima de dez elementos. Representando os elementos da pilha, te-

$Stack[T]$

$size : \mathbb{N}$ [class constants]

$size = 10$

$elements : \text{seq}\ T$ [class attributes]

$\#elements \leq size$ [class invariants]

$\textit{INIT}$

$elements = \langle\, \rangle$ [inicial conditions]

$push$

$\Delta(elements)$ [Delta list]

$item? : T$ [method parameters]

$\#elements < size$

$elements' = \langle item? \rangle \frown elements$

pop

$\Delta(elements)$

$item! : T$

$elements \neq \langle\, \rangle$

$item! = head\ \ elements$

$elements' = tail\ \ elements$

Especificação 2.1: Especificação de uma Pilha em Object-Z.

mos o atributo *elements*, que consiste em uma sequência (ou seja, um conjunto ordenado) de elementos do tipo T, que no estado inicial da classe não possui nenhum elemento, ou seja, a pilha inicia-se vazia.

Para manipulação da mesma, podem ser utilizados os métodos *push* e *pop*, que retiram e adicionam elementos no início da pilha, respectivamente. No método *push*, para que seja possível adicionar o elemento *item*?, o número de elementos existentes na pilha deve ser menor do que o seu tamanho. Caso essa condição seja satisfeita, *item*? é adicionado no início da pilha. Assim como em CSP, os decoradores "?" e "!" indicam variável recebida do ambiente e variável retornada ao ambiente, respectivamente. Já o método *pop* retorna primeiro elemento da pilha (*item*!) caso esta não esteja vazia, removendo em seguida este elemento da pilha.

O exemplo acima serviu apenas para ilustrar superficialmente como seria uma especificação em Object-Z. Maiores detalhes sobre como construir especificações completas nessa linguagem podem ser encontradas em Duke e Rose (2000).

2.3.3 CSP-OZ

CSP-OZ (FISCHER, 1997; FISCHER; WEHRHEIM, 2000; FISCHER, 2000) é uma linguagem de especificação de alto nível criada para a descrição de sistemas distribuídos.

Conforme descrito acima, Object-Z é uma extensão da linguagem Z orientada a objetos destinada à especificação dos objetos de um sistema. Uma classe em Object-Z consiste na especificação de um espaço de estados e das operações neste espaço. A álgebra de processos CSP foi desenvolvida para descrever o comportamento dinâmico de um sistema. A idéia básica de CSP-OZ é definir a semântica de uma classe utilizando o modelo semântico de CSP. Portanto, todos os operadores de CSP podem ser utilizados para combinar objetos, e a sintaxe de CSP e Object-Z podem ser misturadas. A semântica de um processo em CSP é baseada em seu alfabeto, ou seja, no conjunto de eventos que este pode observar do ambiente. Já em Object-Z, os eventos observáveis de uma classe são seus métodos com seus respectivos parâmetros.

O objetivo principal de CSP-OZ é dar às classes em Object-Z a semântica de CSP. Como consequência, as classes são um modo alternativo de descrição de um processo de CSP, ou seja, classes em CSP-OZ são a versão orientada a objetos de um processo CSP (FISCHER, 2000). Essa visão mais voltada à orientação a

objetos facilita a implementação de uma maneira simples e transparente. Sendo assim, adiciona-se à estrutura básica de uma especificação em Object-Z a definição dos canais de comunicação da classe e a especificação de seu comportamento em CSP, que consiste na especificação da ordem em que estes métodos e canais são utilizados.

Descrições mais completas da linguagem podem ser encontradas em Fischer (1997) e Fischer (2000).

2.4 Arquiteturas de Controle

Por volta do final da década de 80, a tendência na área de robôs inteligentes era projetar e programar segundo o paradigma reativo. O paradigma reativo permitiu que robôs utilizando processadores de baixo custo e com pouca memória fossem capazes de operar em tempo real. Entretanto, o custo da reatividade era que o sistema eliminava qualquer tipo de planejamento ou funções que envolvessem um conhecimento do estado global do robô em relação ao seu ambiente. Ou seja, o robô não seria capaz de planejar trajetórias otimizadas (planejamento de trajetória), construir mapas, monitorar seu próprio desempenho ou mesmo selecionar os comportamentos necessários no cumprimento de uma dada tarefa. Segundo Murphy (2000), o novo desafio da inteligência artificial no início dos anos 90 era como colocar planejamento e deliberação de volta nos robôs, porém sem perder o sucesso obtido com o controle reativo. O consenso era que o controle comportamental era o modo "correto"de se fazer o controle de baixo nível, devido ao seu sucesso e elegância como teoria computacional tanto para inteligência biológica e de máquinas.

Arquiteturas que utilizam comportamentos reativos, mas que também incorporam planejamento, são conhecidas como parte do paradigma “Híbrido Deliberativo/Reativo”. Segundo Murphy (2000), as híbridas são a melhor solução geral de arquitetura por duas razões: primeiro, o uso de técnicas de processamento assíncrono (*multi-tasking, threads*, etc.) permitem que as tarefas deliberativas executem independentemente dos comportamentos reativos. Isso implica, por exemplo, que um planejador pode calcular a próxima trajetória do robô de uma forma mais lenta, enquanto que o robô continua reagindo a estímulos rapidamente, com os comportamentos trabalhando a uma frequência maior; segundo, uma boa modularidade de software permite que subsistemas ou objetos em uma arquitetura híbrida sejam misturados e/ou ativados para aplicações específicas. Com isso, aplicações em que se exige comportamento puramente reativo poderiam

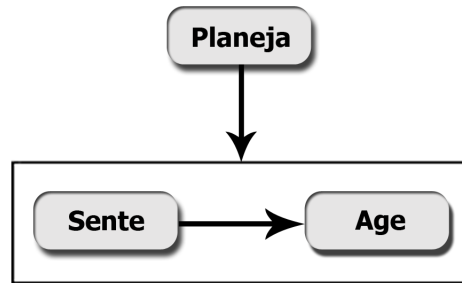


Figura 2.2: Esquema de funcionamento do paradigma Híbrido.

ser abordadas utilizando-se apenas uma parte da arquitetura híbrida, enquanto que em uma aplicação onde se exigiria um maior grau de inteligência do robô se utilizaria a arquitetura completa.

Conceitualmente, ela é dividida em duas partes: uma reativa e a outra deliberativa. A organização de um sistema híbrido pode ser descrita como: *PLANEJA*, e então *SENTE-AGE*, como pode ser visto na figura 2.2. O *PLANEJA* inclui toda a deliberação e modelagem do mundo, e não apenas planejamento de caminhos ou tarefas. Essa separação deve-se ao planejamento consumir muito tempo de processamento, e portanto ele deve ser desacoplado da execução de tempo real. Primeiramente, o robô planeja como executar a missão (utilizando um modelo do ambiente) ou uma tarefa. Em seguida, ele inicializa ou ativa um conjunto de comportamentos (*SENTE-AGE*) para executar o plano. Os comportamentos executarão até que o plano esteja completo, para que então o planejador gere um novo conjunto de comportamentos, e assim por diante.

A grande vantagem no desenvolvimento de uma arquitetura híbrida é a possibilidade da junção das vantagens presentes tanto em arquiteturas deliberativas quanto em reativas. Segundo Simpson, Jacobsen e Jadud (2006), uma abordagem puramente hierárquica para controle de robôs está focada principalmente no aspecto de planejamento de um ciclo comportamental do robô. Neste ciclo, primeiramente ele recebe informações do ambiente. Em seguida planeja sua próxima ação baseado nessas informações, para somente então executar alguma ação utilizando seu conjunto de atuadores. Em cada ciclo, o robô planeja explicitamente sua próxima ação a partir do conhecimento reunido do ambiente. A desvantagem deste tipo de abordagem é que ela não propicia uma separação de tarefas, e pode introduzir dependências entre camadas de funcionalidade, além de inserir um atraso relativamente alto entre o sensoriamento e a atuação se comparado ao paradigma comportamental.

Dentro do paradigma reativo, um dos principais modos é a arquitetura *Sub-*

sumption (BROOKS, 1986). Nessa arquitetura, a percepção do ambiente e o planejamento para a execução de um determinado comportamento interagem diretamente uns com os outros. A construção de sistemas de controle de robôs é feita por meio de níveis crescentes de competência, onde cada nível adicional é construído com base em um nível pré-existente, ou pelo menos interage com níveis pré-existentes, para que a cada novo nível sejam inseridas novas competências e que a junção de todos os níveis (ou camadas) determinem o sistema como um todo. Com o uso correto de supressores e inibidores (que são definidos em seu trabalho), o sistema pode variar entre vários modos de operação diferentes dependendo das entradas do ambiente, fazendo uma melhor re-utilização de módulos já escritos e bem testados. A idéia é remover estruturas de controle centralizadas. A partir de comportamentos simples e da interação entre eles e o ambiente, podem surgir comportamentos resultantes mais complexos. Isso traz a vantagem de tornar o projeto de sistemas robóticos mais simples e o código envolvido mais robusto, uma vez que o sistema é composto de vários componentes pequenos e simples. Entretanto, a principal critica à este paradigma de controle consiste na dificuldade de se definir os comportamentos necessários, e também de se desenvolver o sistema de um modo linear de modo a aproveitar a vantagem de desenvolvimento incremental.

No contexto deste trabalho, a utilização da arquitetura *Subsumption* apresenta grandes vantagens relacionadas à verificação e implementação dos modelos desenvolvidos para a arquiteturas de controle de veículos submarinos. A primeira delas já foi mencionada, que é a divisão do sistema em diversos processos menores, que por sua vez apresentam máquinas de estados menores, facilitando assim o processo de análise com o FDR. Outra vantagem consiste na facilidade de correlação entre os processos da arquitetura *Subsumption* com os processos de CSP. Isso porque assim como em CSP, na arquitetura *Subsumption* o sistema é composto por diversos processos que executam de maneira concorrente entre si, se comunicando exclusivamente por meio de canais.

Neste trabalho será desenvolvida uma arquitetura híbrida, porém sendo implementada apenas sua parte reativa. A camada deliberativa, cujo processo principal é o *Autonomous*, será implementada em trabalhos futuros. A camada reativa será projetada seguindo-se as idéias da arquitetura *Subsumption*, de modo que o veículo evite colisões ao se movimentar. Entretanto, essa característica não será efetivamente testada, pois não existem sensores embarcados no robô utilizado capazes de detectar obstáculos no ambiente.

2.5 Conclusões do Capítulo

Neste capítulo foram apresentados alguns dos conceitos básicos utilizados no trabalho relacionados a linguagens de modelagem de software (em especial às linguagens formais) e a arquiteturas de controle.

Foi visto que a UML é atualmente o padrão utilizado na indústria. Porém, a utilização desta linguagem pode levar a ambiguidades no modelo desenvolvido em função da falta de uma semântica precisa em alguns de seus diagramas. A fim de se amenizar este problema, foi criada a OCL, além de existirem diversos trabalhos que propõem uma união da UML com linguagens formais. Neste trabalho, para a modelagem da arquitetura de controle desenvolvida utilizou-se a linguagem formal CSP-OZ, possibilitando assim a especificação de seus processos e de toda a estrutura de dados utilizada, além de possibilitar que o modelo seja analisado com auxílio da ferramenta FDR a fim de se indentificar *deadlocks*, *livelocks* e checar seu determinismo.

Com relação à arquitetura de controle, esta foi desenvolvida segundo o paradigma híbrido. Assim, ela foi dividida em duas camadas conceituais: uma deliberativa, onde é estipulado o próximo movimento do robô; e uma reativa, onde estão presentes as funções básicas de sobreviência e funcionamento do robô. A vantagem dessa abordagem consiste na separação das atividades, e assim as funções de sobrevivência podem ser executadas a uma frequência maior, enquanto que as demais funções menos críticas podem levar mais tempo para serem processadas.

3 Método de Desenvolvimento

Neste trabalho está sendo proposto um método para o desenvolvimento robusto de software embarcado para veículos submarinos. Esse baseia-se no uso de métodos formais para a especificação do sistema, juntamente com ferramentas de checagem do modelo desenvolvido e de sua implementação. Conforme mostrado na figura 3.1, ele é composto de três fases, que podem ser divididas em etapas sequenciais.

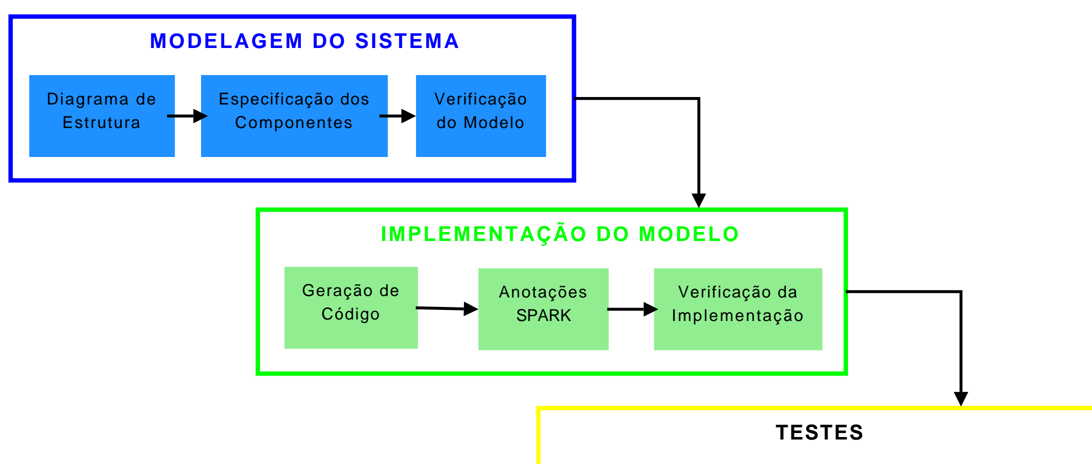


Figura 3.1: Método de Desenvolvimento Proposto.

O objetivo da utilização de métodos formais consiste na possibilidade de se fazer provas formais e checagens sobre a corretude do modelo desenvolvido. Além disso, seu uso permite ainda um maior detalhamento sobre as condições e limites no uso de métodos e variáveis do sistema. No caso deste trabalho, o foco se dará apenas no uso de ferramentas de checagem de modelos e suas implementações, não sendo elaboradas provas formais. A checagem de modelos permite que boa parte dos erros sejam descobertos antes da fase de implementação do sistema, o que implica em uma grande redução dos custos de desenvolvimento de software (AMEY, 2002). Dependendo do tamanho dos modelos, especialmente para os pequenos, há um aumento significativo em sua confiabilidade em razão de sua checagem com o FDR (LAWRENCE, 2005). Já a checagem da implementação permite encontrar e corrigir trechos de código propensos a gerar exceções.

Nas seções a seguir serão apresentadas cada uma das fases e etapas do método proposto. Um exemplo simples de sua aplicação por meio de um modelo Produtor / Consumidor, incluindo especificação e implementação, será mostrado na seção 3.4. Já a sua aplicação em um caso real será mostrada por meio do desenvolvimento de uma arquitetura de controle para um robô submarino do tipo ROV, que fará parte do sistema descrito com maiores detalhes no capítulo 4. A especificação completa em CSP-OZ da arquitetura, bem como o código fonte de sua implementação podem ser vistos no capítulo 5.

3.1 Modelagem do Sistema

A primeira fase do desenvolvimento consiste na elaboração do modelo do software. Este deve conter tanto uma especificação da estruturação de seus componentes quanto uma especificação individual mais detalhada de cada um deles, considerando suas estruturas de dados e comportamentos. Depois de finalizadas as especificações, estas devem ser checadas no FDR quanto à ausência de *deadlocks*, *livelocks*, divergências e determinismo.

A seguir serão apresentadas cada uma das etapas dessa fase.

3.1.1 Diagrama de Estrutura em CSP

Nesta etapa deve ser elaborada a estrutura do sistema, identificando todos os seus componentes e também o modo de interação entre eles. A especificação dessa estrutura será feita utilizando-se a álgebra de processos CSP. Assim, o sistema deve ser decomposto em processos independentes, que se comunicam de maneira síncrona através de canais unidirecionais. Os canais são o único meio de comunicação entre os processos, não sendo permitido por exemplo que um processo A acesse diretamente algum dado de um processo B, a menos que esse dado seja transmitido através de um canal que vá de B para A.

A divisão do sistema em vários processos menores apresenta vantagens e desvantagens. Dentre as vantagens podemos citar o aumento de sua modularidade, o que facilita o desenvolvimento e a realização de testes em cada módulo de maneira independente. Devido ao tamanho reduzido de cada processo, sua implementação em código torna-se muito mais simples e concisa, o que facilita o processo de verificação. Entretanto, a desvantagem de tal divisão consiste no aumento da complexidade do sistema devido ao aumento no número de componentes e consequentemente em seus modos de interação. O uso de ferramentas como o FDR tem

como objetivo facilitar a análise dos comportamentos dos modelos desenvolvidos, possibilitando a realização de análises automáticas, reduzindo assim os efeitos desse aumento de complexidade.

3.1.1.1 Definição dos Processos

Inicialmente, devem ser definidos todos os componentes do sistema. Um ponto a se reforçar é que estes componentes são processos, que executam em paralelo ou sequencialmente entre si e se comunicam somente por meio de canais. A idéia é que eles sejam coesos, ou seja, devem realizar apenas uma determinada função específica. Sendo assim, diferentes funcionalidades devem ser realizadas por diferentes processos.

Depois de definidos os processos, devem ser definidos os seus modos de execução entre si. Estes podem ser *paralelos* ou *sequenciais*. Processos em paralelo são executados ao mesmo tempo, ou seja, executam de maneira concorrente entre si, enquanto que os sequenciais executam sempre de acordo com a ordem estabelecida na especificação, sempre um após o outro.

3.1.1.2 Definição dos Canais de Comunicação

Depois de definidos os processos e seus modos de execução, devem ser definidos todos os canais de comunicação entre os processos do sistema. Estes servem como uma região tanto para troca de informações quanto para sincronização entre diferentes processos. Os canais de comunicação devem ser sempre unidirecionais. Caso seja necessária uma comunicação bidirecional entre dois processos, devem ser criados dois canais unidirecionais, cada um transmitindo informações em um sentido.

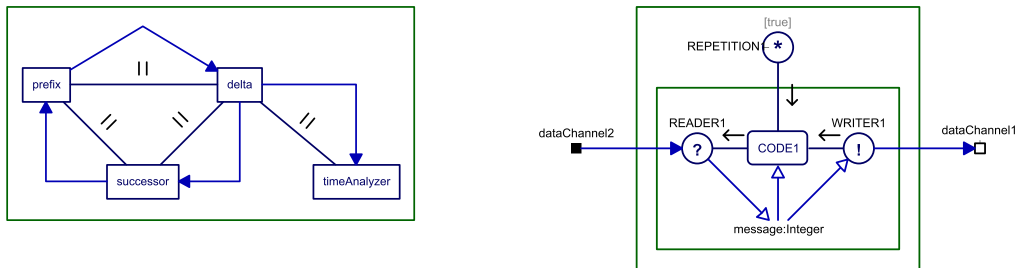
Com relação à nomenclatura, ela seguirá o seguinte padrão:

origem_destino:tipo

onde a primeira parte do nome representa o processo de origem e a segunda parte o processo de destino do canal, em letras minúsculas. Após o nome, deve ser especificado o tipo de dado transmitido pelo canal, separado pelo marcador ':'. Essa padronização nos nomes dos canais facilita o entendimento do código gerado na implementação do sistema.

3.1.1.3 Ferramenta Gráfica gCSP

De modo a auxiliar na especificação da estrutura em CSP, será utilizada uma ferramenta gráfica denominada gCSP (JOVANOVIC et al., 2004). Ela permite que sejam modelados sistemas completos por meio de canais e processos em CSP utilizando recursos gráficos. A figura 3.2 mostra um exemplo de modelo gráfico em gCSP, tanto da estrutura de um sistema como da especificação interna de um processo. Portanto, essa ferramenta permite a elaboração de especificações em diferentes níveis de abstração. No caso deste método, apenas a estrutura do sistema será especificada, sendo que a especificação do comportamento interno dos processos será feita manualmente, tendo como base as especificações em CSP-OZ.



(a) Estrutura de um Sistema em Processos

(b) Comportamento Interno de um Processo

Figura 3.2: Exemplo de Especificação em gCSP.

Com base nos diagramas desenvolvidos, a ferramenta oferece a possibilidade de geração automática de código em três diferentes linguagens: C++, Occam e CSP_M. Como a linguagem de programação escolhida foi Ada, apenas a geração de código em CSP_M (que é a utilizada no FDR) será aproveitada.

3.1.2 Especificação dos Componentes em CSP-OZ

Nesta etapa do desenvolvimento, o objetivo é fazer uma especificação mais detalhada de cada processo que compõe no sistema, englobando tanto sua estrutura de dados interna quanto seu comportamento dinâmico. Para isso, utiliza-se a linguagem formal CSP-OZ (FISCHER, 1997, 2000), uma combinação da álgebra de processos CSP com uma extensão da linguagem Z orientada a objetos, denominada Object-Z. Em CSP-OZ, os processos são descritos como classes (*Class*), possuindo duas partes: uma em CSP e uma em Object-Z, conforme descrito a seguir.

3.1.2.1 Parte CSP

A parte CSP de uma classe em CSP-OZ consiste nas especificações tanto de sua interface de comunicação quanto na de seu comportamento dinâmico. A interface compreende todos os canais utilizados pelo processo, além de todos os métodos descritos na parte em Object-Z para manipulação de sua estrutura de dados interna. Isso porque os métodos em Object-Z podem ser vistos em CSP como eventos internos do processo, o que permite que as chamadas dos métodos possam aparecer na especificação do comportamento do processo em CSP. A diferença entre um canal e um método na interface é destacada por meio do uso das palavras reservadas "`chan`", para canais, e "`method`", para os métodos implementados internamente no processo. Os nomes dos canais devem ser os mesmos utilizados nos diagramas feitos em gCSP.

O comportamento principal de cada processo é indicado pela palavra reservada "`main`". Isso porque para melhorar a legibilidade de especificações mais complexas, elas podem ser quebradas em diversos processos menores. No `main` (ou seja, na descrição do comportamento do processo) devem ser colocados sequencialmente todos os eventos que o processo pode aceitar, sendo que estes eventos são tanto as comunicações nos canais quanto as chamadas de métodos internos dos processos. Assim, um exemplo de comportamento básico com a execução de dois métodos ciclicamente seria:

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

3.1.2.2 Parte Object-Z

A parte Object-Z de uma classe em CSP-OZ consiste na especificação da estrutura de dados interna, juntamente com seus métodos de manipulação. Nessa parte devem ser especificadas as variáveis internas dos processos com seus valores iniciais, e também todos os métodos internos dos processos.

Além disso, cada canal de comunicação utilizado pelo processo deve ser especificado na classe como um método cujo nome possui o prefixo "`com_`" mais o nome do canal de acordo com o nome indicado no diagrama em gCSP. Este representa o efeito do uso do canal na classe, que geralmente consiste em uma passagem de informação. Assim, um canal de nome *theChannel* resultaria em um método de nome `com_`*theChannel* nos dois processos ligados pelo canal.

3.1.3 Checagem do Modelo

Depois de feita a especificação completa do modelo do sistema, ou seja, a especificação da estrutura no gCSP e as dos processos em CSP-OZ, esta deve ser checada a fim de se eliminar possíveis erros relacionados à concorrência dos processos. Este tipo de erro não aparece ao se testar individualmente os componentes de um sistema, existindo apenas como resultado de suas relações. É importante ressaltar que essa verificação de corretude é feita apenas no modelo desenvolvido. Nesta etapa ainda não é levado em conta nenhum aspecto da implementação. A verificação do modelo consiste em duas etapas, descritas a seguir.

3.1.3.1 Geração de Scripts em $\mathrm{CSP_M}$

Conforme dito anteriormente na seção 3.1.1.3, a ferramenta gCSP permite que seja gerado automaticamente um *script* em $\mathrm{CSP_M}$ que contém as especificações dos processos do modelo, juntamente com suas composições e ligações por meio dos canais declarados no modelo. $\mathrm{CSP_M}$ é a linguagem utilizada pelo FDR, e não se trata de CSP puro, mas sim de uma extensão de CSP com uma linguagem funcional básica (FISCHER; WEHRHEIM, 1999). Por meio dela, é possível fazer uma série de checagens no modelo, como a não existência de *deadlocks, livelocks*, determinismo dos processos, e também verificar relações de refinamento entre níveis de especificação que vão desde especificações de mais alto nível até especificações mais próximas da implementação.

Existem trabalhos que mostram como fazer a elaboração de *scripts* em $\mathrm{CSP_M}$ a partir de modelos em CSP-Z e CSP-OZ, como é o caso dos trabalhos de Fischer e Wehrheim (1999), Kassel e Smith (2001), e Mota, Farias e Sampaio (2001). Estes apresentam regras de mapeamento tanto para a parte dinâmica quanto para a estrutura de dados das classes. Como neste trabalho serão utilizadas as ferramentas de checagem do SPARK Ada para analisar as estruturas de dados, estas não serão analisadas no FDR. Sendo assim, as especificações em $\mathrm{CSP_M}$ deverão conter apenas os aspectos dinâmicos das classes em CSP-OZ, ou seja, a sua parte CSP.

Neste método proposto, utiliza-se como base o *script* gerado automaticamente pela ferramenta gCSP. Entretanto, este deve ser modificado manualmente a fim de se ajustar as especificações em $\mathrm{CSP_M}$ de acordo com as feitas em CSP-OZ. Como neste trabalho não foi feita a especificação do comportamento interno dos processos na ferramenta gCSP, esta gera por padrão um "comportamento nulo" para cada processo, que é o processo "`Skip`" de CSP. Além disso, os métodos

inseridos na especificação em CSP-OZ não aparecem no diagrama em gCSP, e portanto também não aparecem no arquivo gerado automaticamente. A fim de se contornar isso, foi estabelecido um pequeno conjunto de regras:

1. Todos os métodos declarados na interface das classes em CSP-OZ, ou seja, que foram adicionados na parte Object-Z das especificações e não fazem parte do diagrama em gCSP, devem ser inseridos manualmente em $\mathrm{CSP_M}$ como canais.

2. Nas especificações dos processos em $\mathrm{CSP_M}$, o processo `Skip` gerado automaticamente pela ferramenta deve ser substituído pela especificação dos respectivos processos `main` da parte CSP de cada classe em CSP-OZ.

3. Os nomes dos canais no diagrama gCSP são usados como os próprios canais em $\mathrm{CSP_M}$. Estes canais são vistos para os processos como eventos, ou seja, não aparece na especificação em $\mathrm{CSP_M}$ nenhum dado passando pelos canais, embora isso seja o que acontece na prática.

3.1.3.2 Análise com FDR

Nesta etapa, o objetivo é verificar se o modelo desenvolvido apresenta erros relacionados aos aspectos de concorrência do sistema. Para isso utiliza-se a ferramenta de checagem automática de modelos, o FDR, da Formal Systems (SYSTEMS, 2005). Com ela é possível verificar a existência de *deadlocks, livelocks* e não-determinismos nos processos. Além destas verificações, é possível checar relações de refinamento entre especificações de diferentes níveis de abstração, ou seja, pode-se partir de uma especificação mais alto nível e ir refinando-se as especificações, acrescentando-se progressivamente mais detalhes de seu comportamento. A cada passo de refinamento, o FDR verifica se as especificações são consistentes entre si.

Antes de se iniciar a implementação de qualquer *software*, é preciso saber se o modelo desenvolvido está correto, de acordo com o que se deseja do mesmo. Isso porque um modelo incorreto torna impossível uma implementação de acordo com os requisitos estabelecidos, mesmo que se disponha dos melhores programadores, uma vez que a implementação deve sempre ser baseada no modelo desenvolvido.

Segundo Lawrence (2005), existem algumas técnicas que ajudam a validar uma especificação de *software*:

- Inspeção cuidadosa, prestando muita atenção nos aspectos de sincronização

e ocultamento (*hiding*), que são fontes comuns de erro;

- Uso de uma ferramenta tal como o ProBe[1] para explorar possíveis comportamentos da especificação;

- Formulação das propriedades esperadas da especificação, e então utilizar o FDR para checá-las.

Neste trabalho serão utilizadas todas essas técnicas no desenvolvimento da arquitetura de controle do ROV, numa tentativa de se minimizar os erros de especificação. Entretanto, o foco principal se dará no último item, que é a checagem da especificação do comportamento dinâmico do sistema em CSP com o FDR.

3.2 Implementação do Modelo

Após a fase de modelagem do sistema, deve ser feita sua implementação em código utilizando uma dada linguagem de programação. Neste método, a linguagem de programação escolhida foi Ada. Segundo Ruiz (2006), Ada possui uma longa história de sucesso na sua utilização em sistemas críticos. Entre as vantagens de sua utilização estão sua alta legibilidade, existência de recursos básicos para programação de sistemas de tempo real como criação e manipulação de processos e utilização de recursos temporais já incluídos na linguagem (ou seja, não são implementadas como uma biblioteca à parte), uso de tipagem forte (*strong typing*) e existência de padrões e subconjuntos da linguagem Ada para sistemas críticos, como o Ravenscar e o SPARK (GOLDSACK, 1985; BARNES, 1989; RUIZ, 2006). Estes últimos visam o desenvolvimento de sistemas determinísticos e verificáveis, e por isso serão utilizados neste método. A utilização do SPARK em conjunto com o Ravenscar também é conhecida como RavenSPARK (TEAM, 2006), que consiste na utilização da linguagem SPARK para trechos sequenciais e de um subconjunto restrito das *Tasks* de Ada de acordo com o Ravenscar para programação concorrente. Uma descrição mais detalhada das implicações da utilização da linguagem RavenSPARK neste método será mostrada a seguir.

No desenvolvimento de softwares críticos, não é desejável utilizar todas as facilidades de uma linguagem complexa, uma vez que complexidade excessiva pode prejudicar a confiabilidade. Ao invés disso, poderia ser utilizado um subconjunto reduzido da linguagem, o que implicaria em um sistema embarcado mais compacto e eficiente. Uma outra vantagem de se reduzir o escopo da linguagem é a

[1]Ferramenta para a inspeção do espaço de estados de um processo CSP, também da Formal Systems.

redução de sua complexidade, o que facilita a geração de provas de corretude, previsibilidade, confiabilidade e análise de cobertura de código, se necessário (RUIZ, 2006). Essa é a abordagem empregada em Ada pelo Ravenscar (BURNS; DOBBING; VARDANEGA, 2004) e o SPARK (BARNES, 2006). Ambos utilizam essa idéia de se restringir os comandos da linguagem, de modo a se obter uma implementação mais eficiente, confiável e determinística. Em outras linguagens também se observa este tipo de abordagem, como a tentativa de se definir um subconjunto *"safe C"* para a linguagem C, denominado MISRA C. O mesmo observa-se em C++, com o MISRA C++.

3.2.1 Geração de Código

Neste trabalho, a transformação da especificação formal em um programa será feita manualmente. Entretanto, existem ferramentas que são capazes de fazer essa transformação de forma automática. Com relação ao processo manual, um exemplo pode ser visto no trabalho de Lawrence (2005), onde foi feita a passagem da especificação de uma parte de um sistema da IBM feita em CSP para a linguagem Java. Com relação à geração automática com base em modelos, um exemplo destas seria a gCSP (JOVANOVIC et al., 2004), que permite a geração de código em C++ e Occam a partir de modelos em CSP. Outro exemplo seria o Rhapsody, da IBM (TELELOGIC, 2009). Este permite a geração de código em C, C++, Java e Ada a partir de modelos em UML-RT.

Inicialmente houve uma tentativa de se utilizar o Rhapsody como ferramenta de geração automática de código neste trabalho. Porém, esta abordagem não foi bem sucedida em razão de a ferramenta não gerar um código compatível com os padrões da linguagem Ada utilizados neste trabalho, que são o Ravenscar e o SPARK. Assim, realiza-se manualmente a conversão das especificações formais para código em SPARK através de regras de mapeamento que serão apresentadas ao longo deste capítulo.

3.2.1.1 Restrições da Linguagem

A linguagem SPARK em conjunto com o perfil Ravenscar impõem uma série de restrições com relação ao uso de recursos da linguagem Ada, e que podem ser vistos com maiores detalhes em Team (2006), Burns, Dobbing e Vardanega (2004), Amey e Dobbing (2003). Dentre eles, as que afetaram diretamente esta implementação foram:

- Proibido o uso de alocação dinâmica, ponteiros e tipos genéricos;
- Comunicação entre *Tasks* restrita a objetos protegidos ou atômicos;
- Proibido o uso de *select* e *abort* e *delays* relativos nas *Tasks*;

3.2.1.2 Implementação dos Processos

O mapeamento dos processos em CSP-OZ para RavenSPARK foi feito de acordo com as limitações mencionadas acima. Sendo assim, foram definidas algumas regras para a elaboração do código, e que se repetem para todos os processos da especificação em CSP-OZ:

- Cada processo foi implementado como um *package* separado, que contém os métodos internos dos processos e uma *Task* de Ada, que efetivamente atua como o processo em CSP. Esta possui o mesmo nome do *package* acrescido do sufixo "_task".
- Conforme foi dito, os métodos internos de cada processo foram criados dentro do *package*, e não da *Task*. Isso facilita as verificações com o SPARK e também reduz o tamanho e a complexidade do código interno das *Tasks*.
- As variáveis internas dos processos estão localizadas dentro da *Task*, e não do *package*. Assim, elas estão encapsuladas na *Task*, não sendo acessadas diretamente por outras *Tasks*, como ocorre em CSP.
- Todos os processos executam em um *loop* infinito, ou seja, são executados ciclicamente sem nunca terminar. Isso porque segundo o Ravenscar, a finalização de uma *Task* é considerada um erro no programa.

3.2.1.3 Implementação dos Canais de Comunicação

A implementação de canais CSP em RavenSPARK desenvolvida teve como base o método de implementação de canais CSP em Ada proposto nos trabalhos de Atiya e King (2005) e Atiya (2004), que utiliza o padrão Ravenscar. Nestes foi provada a corretude da implementação, mostrando que a semântica destes canais é equivalente à dos canais *one-to-one* da biblioteca JCSP para Java, apresentada no trabalho de Welch e Martin (2000). Como neste trabalho será utilizado o Ravenscar além do SPARK, foram necessárias algumas modificações devido às restrições impostas pelo SPARK e também de modo a se facilitar a verificação do código desenvolvido.

Do mesmo modo que nas implementações originais, os canais em RavenSPARK possuem dois objetos protegidos: *Data*, que armazena o dado transmitido no canal e bloqueia o processo que está irá receber o dado enviado pelo canal; e *Sync*, que bloqueia o processo que está enviando o dado no canal até que o dado seja recebido pelo outro processo. Baseado nisso, foram estabelecidas as seguintes regras de mapeamento dos canais em CSP para RavenSPARK:

- Para cada tipo de canal foi criado um *package* diferente. O tipo de canal é definido pelo tipo de dado transferido por ele, ou seja, se o dado transferido for um inteiro, o canal será do tipo *IntegerChannel*. Se for transmitir mensagens, será do tipo *MessageChannel*, e assim por diante.
- Cada *package* possui dois tipos de objetos protegidos: *Data* e *Sync*, conforme mencionados acima. O dado interno armazenado pelo objeto protegido *Data* possui o mesmo tipo de dado do canal.
- Cada canal presente no diagrama em gCSP deve ser criado (ou instanciado) dentro do *package* que representa o seu tipo de canal. Assim, todos os canais de mesmo tipo são criados dentro do mesmo *package*. Isso ocorre devido à restrições impostas pelo RavenSPARK e também facilita o processo de verificação da implementação.
- A nomenclatura das instâncias dos canais obedece à seguinte regra: o prefixo consiste no mesmo nome presente no diagrama gCSP,e o sufixo difere para os dois componentes do canal: "_d" para o objeto do tipo *Data* e "_s" para o do tipo *Sync*. Assim, por exemplo, para um canal de inteiros cujo nome seja *myChannel* em CSP, sua implementação em RavenSPARK consistirá na criação de dois objetos: *myChannel_d* e *myChannel_s*, ambos criados dentro do *package IntegerChannel*.
- O envio e recepção de dados pelos canais se dará por meio da chamada dos métodos *put*, *get*, *stay* e *proceed*, da seguinte forma:
  - Na *Task* que envia o dado *someData*, o código para uso do canal seria, por exemplo:

    Código Fonte 3.1: Código para envio de dados pelo canal.

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

  - Na *Task* que recebe o dado do canal e o armazena na variável *myData*, o código para uso do canal seria, por exemplo:

Código Fonte 3.2: Código para recepção de dados pelo canal.

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

3.2.2 Anotações SPARK

Com base nas especificações em CSP-OZ, além de gerado todo o código que implementa o sistema, são geradas as anotações SPARK para todos os componentes que serão utilizadas na próxima etapa da metodologia, que é a verificação do código implementado.

As anotações são inseridas no meio do código na forma de comentários que se iniciam com os símbolos "–#". Estas servem como base para as análises feitas com o examinador da linguagem SPARK. O trecho de código 3.3 mostra como exemplo a especificação de um método com suas anotações SPARK. Dentre os tipos de comentários inseridos estão pré e pós condições de variáveis na execução de métodos, prioridades para *Tasks* e objetos protegidos e declaração modos de uso de variáveis (leitura ou escrita) ao longo do programa. Os detalhes de quais anotações devem ser inseridas e seu modo de uso podem ser vistos em (BARNES, 2006) e (TEAM, 2006).

Código Fonte 3.3: Exemplo de código com anotações SPARK

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

A principal vantagem da utilização de CSP-OZ em conjunto com SPARK consiste na possibilidade da elaboração de pré e pós condições para a execução de métodos, bem como da especificação dos intervalos de valores válidos para todas as variáveis utilizadas na implementação durante a modelagem do sistema. Assim, as restrições feitas em CSP-OZ seriam mapeadas diretamente para anotações em RavenSPARK, reduzindo assim o tempo de elaboração das mesmas durante a fase de implementação.

Outra vantagem da utilização desta linguagem formal consiste na elaboração de tipos de dados em Ada com base na especificação de tipos de dados em CSP-OZ, ou seja, utilizar a tipagem forte da linguagem Ada para implementar os tipos básicos definidos em CSP-OZ. Esses tipos de dados em Ada, que possuem intervalos válidos pré-definidos, são então utilizados nas análises das variáveis

criadas na implementação a fim de se encontrar trechos de código que apresentem possíveis erros.

3.2.3 Checagem da Implementação

Com base nas anotações SPARK inseridas, o examinador da linguagem, o SPARK *examiner*, é capaz de fazer certas checagens no código. Este possui duas funções básicas:

- Checar a conformidade do código com as regras da linguagem;
- Checar a consistência entre o código e suas anotações, realizando análises nos dados e no fluxo dos mesmos.

A análise executada pelo examinador baseia-se fortemente na análise das interfaces entre os componentes, assegurando também que as implementações em todos os componentes estão de acordo com as especificações indicadas em suas respectivas interfaces. Deste modo, a linguagem SPARK juntamente com as suas anotações asseguram que um programa não pode conter certos erros relacionados ao fluxo de informação. Isso porque o examinador detecta o uso de variáveis não inicializadas e a sobrescrita de variáveis antes que elas sejam usadas. Já o comportamento dinâmico é verificado por meio das anotações de pré e pós condições, permitindo que o analisador gere teoremas que portanto devem ser provados a fim de verificar que o programa não apresenta erros.

Neste trabalho não serão geradas pré e pós condições para as variáveis e métodos da especificação. Sendo assim, estas também não serão feitas nas anotações SPARK, e portanto não serão elaborados teoremas ou provas formais. Será intensivamente explorado o conceito de tipagem forte, onde todos os tipos de dados utilizados na implementação serão pré-definidos e especificados com seus intervalos de validade. Estes é que serão utilizados pelo examinador em suas análises, de forma automática.

3.3 Testes

Depois de finalizadas as etapas de modelagem e implementação, cada uma com sua etapa final de checagem, são feitos os testes no sistema embarcado. Estes são os testes comuns realizados em qualquer desenvolvimento de software, podendo ser utilizadas as técnicas mais apropriadas para cada tipo de aplicação.

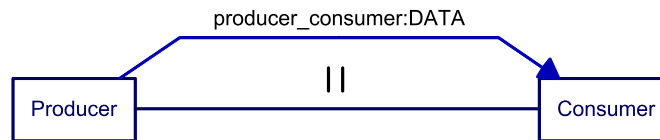


Figura 3.3: Modelo Produtor / Consumidor em gCSP.

Como neste trabalho está sendo utilizado o sistema operacional de tempo real VxWorks, serão utilizadas suas ferramentas de análise do código em tempo de execução nesta etapa de testes. Este possui um simulador onde pode ser embarcado o código desenvolvido, com possibilidade de configuração para diversos tipos de hardware. Depois de configurado o hardware desejado, que consiste basicamente de tipo de processador e quantidade de memória utilizada, é possível verificar em tempo de execução quais processos estão sendo executados, quanto cada um consome em termos de processamento e de memória, entre outros.

3.4 Exemplo de Aplicação: modelo Produtor/Consumidor

Nesta seção será mostrada a aplicação do método de desenvolvimento proposto no modelo clássico de Produtor / Consumidor. O objetivo é mostrar uma aplicação prática de cada uma das fases, sem termos de nos preocupar com detalhes de um modelo complexo.

O modelo consiste de dois processos, que são executados em paralelo: um *produtor*, que deve produzir e enviar números inteiros, sequenciais, e que variam no intervalo de zero a dez; um *consumidor*, que deve receber estes dados e os imprimir na tela. Um requisito adicional é que não deve ser produzido um dado novo até que este seja consumido.

3.4.1 Modelagem

3.4.1.1 Estrutura

A primeira etapa da modelagem consiste na elaboração do diagrama de estrutura do sistema. Este deve ser feito em CSP, com o auxílio da ferramenta gráfica gCSP. Assim, foram definidos os dois processos presentes no sistema, cujo diagrama pode ser visto na figura 3.3: *Producer* e *Consumer*, que executam em paralelo entre si, se comunicando por meio do canal *producer_consumer*, que vai de *Producer* para *Consumer* transportando uma variável do tipo *DATA*.

3.4.1.2 Componentes

A próxima etapa consiste na elaboração das especificações em CSP-OZ do sistema. Antes de especificar os processos, devem ser especificados todos os tipos primitivos e estruturas de dados utilizadas ao longo da especificação. No caso deste modelo simples, temos apenas o seguinte tipo:

$$DATA ::= \{x : \mathbb{N} \mid 0 \leq x \leq 10\}$$

A especificação do processo produtor pode ser vista na especificação 3.1. Primeiramente, temos a declaração de sua interface formada pelo canal *producer_consumer* e o método interno *produceData*, precedidos pelas palavras reservadas `chan` e `method`, respectivamente. Logo abaixo da interface temos a especificação do comportamento dinâmico do processo, representado pela palavra reservada `main`.

Producer

`chan` $producer_consumer : [producerData : DATA]$
`method` $produceData$

$$\texttt{main} = produceData \rightarrow producer_consumer \rightarrow \texttt{main}$$

$myData : DATA$

$0 \leq myData \leq 10$ [this is a redundant specification]

INIT

$myData = 0$

$\texttt{com}_producer_consumer$

$\Delta()$
$producerData! : DATA$

$producerData! = myData$

produceData

$\Delta(myData)$

$myData = 10 \wedge myData' = 0$
$\vee$
$myData \leq 10 \wedge myData' = myData + 1$

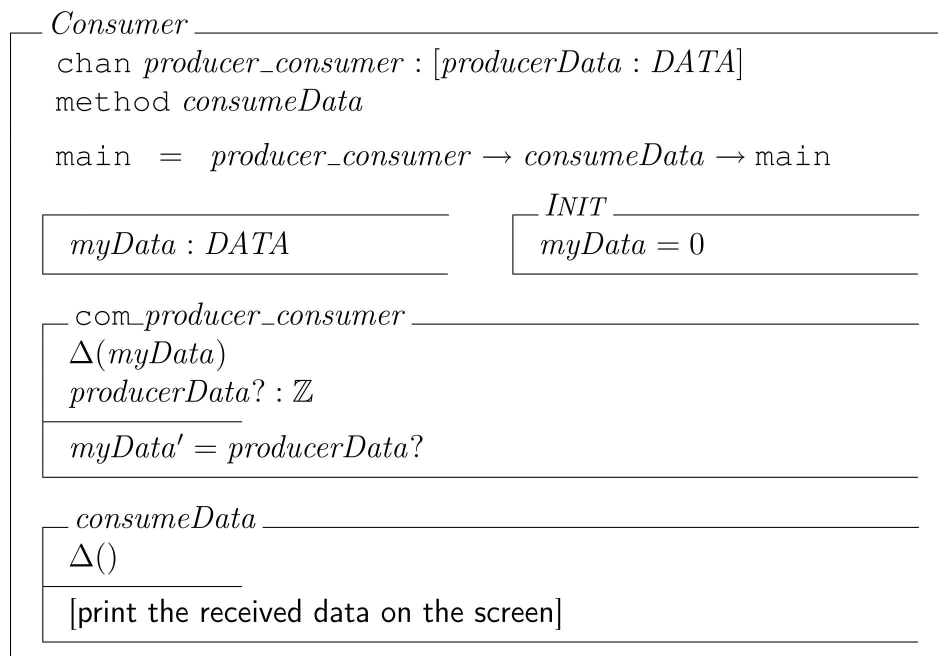
Especificação 3.1: Processo Produtor.

Este possui o atributo *myData* do tipo DATA, ou seja, um dado inteiro que possui um intervalo de valores válidos de zero a dez. Como o intervalo de valores válidos já foi especificado na declaração do tipo *DATA*, este não precisa ser repe-

tido ao longo das especificações como consta na declaração da variável *myData*, tornando-se redundante. Portanto, sempre que os intervalos de valores válidos de um método ou variável coincidir com o da declaração do tipo básico da variável utilizada, estes serão sempre ocultados. Isso pode ser visto na declaração da variável *myData* do processo *Consumer*.

Com relação ao canal *producer_consumer*, o efeito de sua utilização pelo processo está representado pelo método cujo nome é o mesmo do usado no canal acrescido do prefixo com_, ou seja, com_*producer_consumer*. Como o efeito dos canais no método de desenvolvimento proposto é a troca de informações entre processos, o método com_*producer_consumer* apenas efetua essa troca de informações, armazenando na variável *myData* a variável *producerData*, que é o dado transmitido pelo processo *Producer*.

De maneira semelhante à mostrada acima, foi elaborada a especificação em CSP-OZ do processo consumidor (Especificação 3.4.1.2).



Especificação 3.2: Processo Consumidor.

3.4.1.3 Checagem

Depois de finalizadas as especificações dos diagramas de estrutura em gCSP e CSP-OZ, deve ser gerado automaticamente pela ferramenta gCSP um *script* em CSP_M. Este deve ser modificado manualmente para incluir os métodos internos implementados pelos processos sempre que necessário, e também o com-

portamento dinâmico de cada processo. Isso porque a especificação interna dos processos não será feita na ferramenta gCSP, e portanto ela não aparece na geração automática do *script*. Uma composição englobando todos os processos especificados, seus modos de execução e seus canais de comunicação é gerada automaticamente. Este é o processo que representa o sistema como um todo, resultado da interação de todos os processos presentes na especificação. Neste caso, foi dado a ele o nome de *SYSTEM*. Após todas as modificações manuais, temos:

Código Fonte 3.4: Script CSP_M do modelo Produtor / Consumidor

```
-- MODELO PRODUTOR-CONSUMIDOR EM CSP_M

-- canal entre os processos
channel producer_consumer

-- metodos implementados nos processos
channel produceData, consumeData

-- Especificacao do Sistema
SYSTEM = Producer [| {| producer_consumer |} |] Consumer
         \ {| produceData, consumeData |}

Producer = produceData -> producer_consumer -> Producer
Consumer = producer_consumer -> consumeData -> Consumer

-- Checagens automaticas (deadlock do sistema)
assert SYSTEM :[deadlock free [FD]]
```

Depois de gerado o *script* acima, este foi analisado no FDR para verificar a presença de *deadlocks, livelocks* e determinismo, conforme mostrado na figura 3.4. Pode-se observar que todos os testes realizados foram bem sucedidos.

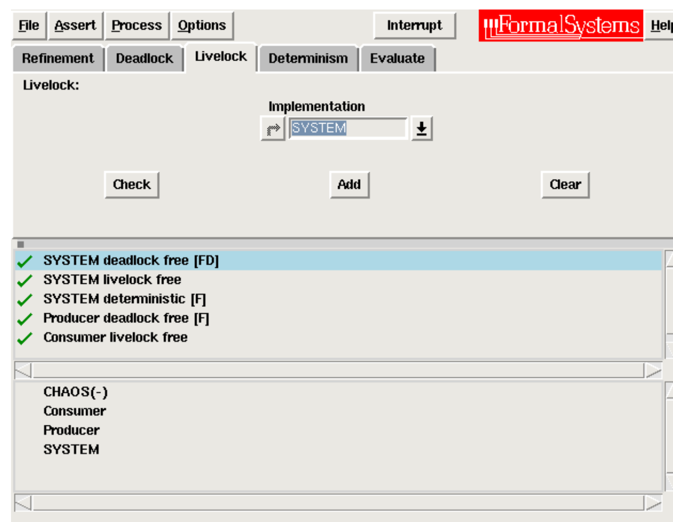


Figura 3.4: Verificação do modelo Produtor - Consumidor no FDR.

3.4.2 Implementação

Depois de verificado o modelo, deve ser feita sua transformação para código em RavenSPARK. Esta é uma passagem manual, seguindo as regras apresentadas no capitulo 3.

3.4.2.1 Codificação

A primeira transformação realizada consiste nas estruturas de dados e tipos básicos utilizados. Estes devem ser todos agrupados em um único *package* a fim de se facilitar futuras modificações no código, denominado *DataTypes*. Neste caso, este *package* contém apenas um único tipo de dado (*DATA*), juntamente com um único valor padrão (*defaultData*). Existe um valor padrão para cada tipo de dado ou estrutura de dados, que são utilizados na inicialização de variáveis ao longo da implementação. Abaixo segue o seu código fonte.

Código Fonte 3.5: Código Fonte DataTypes.ads.

```
package DataTypes is
   ----------------------------
   -- Data Type declarations --
   ----------------------------
   subtype DATA is Integer range  0 .. 10;

   ----------------------------
   -- Default Initial Values --
   ----------------------------
   defaultData: constant DATA := 0;

end DataTypes;
```

Vale ressaltar que os limites do tipo *DATA* acima foram obtidos à partir de sua especificação em CSP-OZ.

Conforme mencionado no capítulo 3, cada processo foi implementado por meio de um pacote da linguagem Ada. Em Ada, cada *package* pode ser dividido em dois arquivos: um com extensão ".ads", que contém as declarações de entidades internas do *package* como variáveis, *Tasks*, métodos e objetos protegidos; e outro com extensão ".adb", que contém todas as implementações destas entidades. Sendo assim, as implementações do produtor, consumidor e do canal de dados foram divididas em dois arquivos, que seguem abaixo:

Código Fonte 3.6: Código Fonte Producer.ads.

```
with DataTypes;
--# inherit DataTypes,
--#         DataChannel;

package Producer
--# own task producerTask : Producer_task;
```

```
is

   -------------------------------
   -- Task that do the real job --
   -------------------------------
   task type Producer_task
   --# global in out DataChannel.producer_consumer_d;
   --#           out DataChannel.producer_consumer_s;
   --# derives DataChannel.producer_consumer_d,
   --#         DataChannel.producer_consumer_s from  &
   --#         null                            from DataChannel.
      producer_consumer_d;
   --# declare Suspends => DataChannel.producer_consumer_s;
   is
      pragma Priority(10);
   end Producer_task;

   ----------------------------
   -- Methods of the package --
   ----------------------------
   procedure produceData(myData: in out DataTypes.DATA);
   --# derives myData from myData;

   ------------------------
   -- Task instantiation --
   ------------------------
   producerTask: Producer_task;

end Producer;
```

Código Fonte 3.7: Código Fonte Producer.adb.

```
with DataChannel;

package body Producer is

   task body Producer_task is
      -- control data to be transformed
      myData: DataTypes.DATA := DataTypes.defaultData;
   begin
      loop
         -- produce the data
         produceData(myData);
         -- send data through the output channel
         DataChannel.producer_consumer_d.put(myData);
         DataChannel.producer_consumer_s.stay;
      end loop;
   end Producer_task;

   procedure produceData(myData: in out DataTypes.DATA) is
      pragma Inline(produceData);
   begin
      if myData = 10 then
            myData := 0;
         else
            myData := myData + 1;
      end if;
   end produceData;

end Producer;
```

Código Fonte 3.8: Código Fonte Consumer.ads.

```
with DataTypes, DataChannel;
```

```
--# inherit DataTypes,
--#         DataChannel;

package Consumer
--# own task consumerTask : Consumer_task;
is

   -------------------------------
   -- Task that do the real job --
   -------------------------------
   task type Consumer_task
   --# global in out DataChannel.producer_consumer_d;
   --#           out DataChannel.producer_consumer_s;
   --# derives DataChannel.producer_consumer_d,
   --#         DataChannel.producer_consumer_s from  &
   --#         null                             from DataChannel.
      producer_consumer_d;
   --# declare Suspends => DataChannel.producer_consumer_d;
   is
      pragma Priority(10);
   end Consumer_task;

   ----------------------------
   -- Methods of the package --
   ----------------------------
   procedure consumeData(myData: in DataTypes.DATA);
   --# derives null from myData;

   ------------------------
   -- Task instantiation --
   ------------------------
   consumerTask: Consumer_task;

end Consumer;
```

Código Fonte 3.9: Código Fonte Consumer.adb.

```
with Ada.Text_IO, Ada.Integer_Text_IO;

package body Consumer is

   task body Consumer_task is
      -- internal attribute of the process
      myData: DataTypes.DATA := DataTypes.defaultData;
   begin
      loop
         -- receive data from the input channel
         DataChannel.producer_consumer_d.get(myData);
         DataChannel.producer_consumer_s.proceed;
         -- consume the received data
         consumeData(myData);
      end loop;
   end Consumer_task;

   procedure consumeData(myData: in DataTypes.DATA) is
      pragma Inline(consumeData);
   begin
       -- Print the received data on the screen
       Ada.Text_IO.Put("Received: ");
       Ada.Integer_Text_IO.Put(myData);
       Ada.Text_IO.New_Line;
   end consumeData;

end Consumer;
```

Podemos observar os métodos presentes na especificação em CSP-OZ *produceData* e *consumeData* implementados como métodos do *package* em RavenSPARK, e as variáveis *myData* de ambos processos implementadas como variáveis internas à *Task*, conforme as regras estabelecidas. Outro ponto importante consiste nas nomenclaturas adotadas para as *tasks* internas dos *packages*, que também seguem o padrão estabelecido.

Na implementação do canal de comunicação foram utilizados dois tipos de objetos protegidos: *Data* e *Sync*, sendo que o canal descrito no diagrama em gCSP consiste de uma instância de cada um destes objetos. Conforme pode ser visto no código fonte abaixo, os objetos que compõem o canal também são nomeados de acordo com as regras estabelecidas, com o nome do canal como sufixo mais os prefixos "_d" e "_s" para os tipos *Data* e *Sync*, respectivamente.

Código Fonte 3.10: Código Fonte DataChannel.ads.

```
with DataTypes;
--# inherit DataTypes;

package DataChannel
--# own protected producer_consumer_d : Data (priority => 10,
   suspendable);
--#     protected producer_consumer_s : Sync (priority => 10,
   suspendable);
is

   protected type Data is
      pragma Priority(10);

      entry get(someData : out DataTypes.DATA);
      --# global in out Data;
      --# derives Data,
      --#         someData from Data;

      procedure put(someData: in DataTypes.DATA);
      --# global in out Data;
      --# derives Data from *,
      --#                   someData;

   private
      chData : DataTypes.DATA := DataTypes.defaultData;
      readyToRead : Boolean := False; -- Initially, there is no data
         to read
   end Data;


   protected type Sync is
      pragma Priority(10);

      entry stay;
      --# global in out Sync;
      --# derives Sync from *;

      procedure proceed;
      --# global in out Sync;
      --# derives Sync from *;

   private
```

```
      hasRead : Boolean := False;
   end Sync;

   ----------------------------
   -- Channel Instantiations --
   ----------------------------
   producer_consumer_d: Data;
   producer_consumer_s: Sync;

end DataChannel;
```

Código Fonte 3.11: Código Fonte DataChannel.adb.

```
package body DataChannel is

   protected body Data is
      entry get(someData : out DataTypes.DATA) when readyToRead
      --# global in     chData;
      --#          out readyToRead;
      --# derives readyToRead from  &
      --#         someData    from chData;
      is
      begin
         -- Read the encapsulated data
         someData := chData;
         -- Block successive readings
         readyToRead := False;
      end get;

      procedure put(someData : in DataTypes.DATA)
      --# global out readyToRead;
      --#        out chData;
      --# derives readyToRead from  &
      --#         chData      from someData;
      is
      begin
         -- Update the encapsulated data
         chData := someData;
         -- Permit for the next writing operation
         readyToRead := True;
      end put;
   end Data;

   protected body Sync is
      entry stay when hasRead
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Stay for the data to be read
         hasRead := False;
      end stay;

      procedure proceed
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Declare the completion of a reading operation
         hasRead := True;
      end proceed;
   end Sync;

end DataChannel;
```

O programa final resulta da composição dos diversos *packages* do sistema, que representam os processos, os canais e demais estruturas utilizadas no sistema. Cada *package* que contém uma *Task* deve ser indicado em um método, que em outras linguagens de programação como C e Java equivale ao *main*. Além dos *packages* que contém *Tasks*, ele deve conter anotações SPARK referentes a entidades analisadas pelo examinador no programa. Abaixo segue o código fonte.

Código Fonte 3.12: Código Fonte Main.adb.

```
pragma Profile(Ravenscar); -- Enables the Ravenscar Profile

with Producer,
     Consumer;
--# inherit Producer,
--#         Consumer,
--#         DataChannel;
--# main_program;
--# global in out DataChannel.producer_consumer_d;
--#        in out DataChannel.producer_consumer_s;
--# derives DataChannel.producer_consumer_d,
--#         DataChannel.producer_consumer_s from DataChannel.
   producer_consumer_d,
--#                                           DataChannel.
   producer_consumer_s;
procedure Main
--# derives ;
is
   Pragma Priority(10); -- a priority is mandatory and must appear
      here
begin
   null; -- all the real work is done somewhere else
end Main;
```

3.4.2.2 Anotações

Ao longo da implementação devem ser inseridas uma série de anotações da linguagem RavenSPARK para que possam ser feitas análises no código pelo examinador. Estas anotações podem ser vistas nos trechos de código acima, sendo sempre iniciadas pelo símbolo "–#", e referem-se à declaração de variáveis, *Tasks* e objetos protegidos. Com isso, o examinador é capaz de identificar erros relacionados ao fluxo de informações ao longo da implementação, violação de prioridades, entre outros. Neste trabalho serão apenas verificadas apenas estas duas propriedades.

3.4.2.3 Checagem

Com base nas anotações adicionadas, o examinador pode executar uma série de análises na implementação. Neste modelo de Produtor / Consumidor foram analisados os fluxos de dados e de informações, produzindo o relatório abaixo,

não indicando a presença de erros:

Código Fonte 3.13: Relatório do examinador SPARK.

```
            *******************************************************
                           Report of SPARK Examination
        SPARK95 Examiner with VC and RTC Generator Release 7.6 / 06.08
                              Demonstration Version
            *******************************************************


                        DATE : 08-APR-2009 18:30:01.82

Options:
    default switch file used
    index_file=rov_index.idx
    warning_file=rov_warning.wrn
    notarget_compiler_data
    config_file=rov_config.cfg
    source_extension=ada
    listing_extension=lst
    nodictionary
    report_file=RELATORIO.rep
    html
    nostatistics
    fdl_identifiers
    flow_analysis=information
    ada95
    annotation_character=#
    profile=ravenscar
    rules=none
    error_explanations=off
    justification_option=full

Selected files:
   Main.adb

Index Filename(s) used were:
   C:\Producer_Consumer\rov_index.idx

No Meta Files used

Summary warning reporting selected for:
   Hidden parts
   Private types lacking method of initialization
   Notes
   Pragmas: Priority, Inline, Profile

Target configuration file:
Line
   1  -- 32-bit System (Page 196 of the Spark Book)
   2
   3  package Standard is
   4
   5     type Short_Short_Integer is range -2**7 ..  2**7-1;
   6     type Short_Integer is range -2**15 .. 2**15-1;
   7     type Integer is range -2**31 .. 2**31-1;
   8     type Long_Integer is range -2**31 .. 2**31-1;
   9     type Long_Long_Integer is range -2**63 .. 2**63-1;
  10     type Float is digits 6 range -3.40282E+38 ..  3.40282E+38;
  11
  12  end Standard;
  13
  14  package System is
  15
  16     --  System-Dependent Named Numbers
  17
  18     Min_Int               : constant := -2**63;
  19     Max_Int               : constant :=  2**63-1;
  20
  21     Max_Binary_Modulus    : constant := 2**64;
  22     Max_Mantissa          : constant := 63;
  23
  24     --  Storage-related Declarations
  25
  26     type Address is private;
  27
  28     Storage_Unit : constant := 8;
  29     Word_Size    : constant := 32;
  30
  31     --  Priority-related Declarations (RM D.1)
  32
  33     subtype Any_Priority       is Integer      range  0 .. 31;
  34     subtype Priority           is Any_Priority range  0 .. 30;
  35     subtype Interrupt_Priority is Any_Priority range 31 .. 31;
  36
  37  end System;
2 summarized warning(s), comprising:
     2 note(s)


Source Filename(s) used were:
   C:\Producer_Consumer\Main.adb
   C:\Producer_Consumer\DataChannel.ads
   C:\Producer_Consumer\Consumer.ads
   C:\Producer_Consumer\Producer.ads
   C:\Producer_Consumer\DataTypes.ads


Source Filename:   C:\Producer_Consumer\DataChannel.ads
No Listing File

   Unit name:  DataChannel
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Producer_Consumer\Consumer.ads
No Listing File

   Unit name:  Consumer
```

```
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Producer_Consumer\Producer.ads
No Listing File

   Unit name:  Producer
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Producer_Consumer\DataTypes.ads
No Listing File

   Unit name:  DataTypes
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Producer_Consumer\Main.adb
Listing Filename:  C:\Producer_Consumer\SPARK\Main.lst

   Unit name:
   Unit type:  main program
   Unit has been analysed, any errors are listed below.

No errors found

1 summarized warning(s), comprising:
     1 pragma(s)*
(*Note: the above warnings may affect the validity of the analysis.)


--End of file--------------------------------------------------
```

3.4.3 Testes

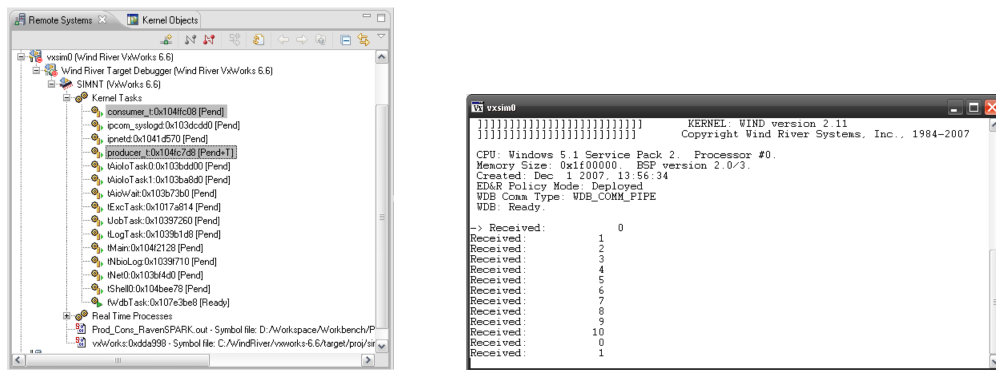
Depois de feitas as análises no FDR e no examinador do SPARK, o código desenvolvido deve ser analisado em tempo de execução utilizando-se as ferramentas do sistema operacional VxWorks.

Neste método, utiliza-se para os testes o simulador de ambientes embarcados presente no *Workbench*, o ambiente de desenvolvimento do VxWorks. Este simulador pode ser configurado para atuar como diversos tipos de hardware. Neste caso, ele foi configurado como um processador Intel Pentium III. A figura 3.5 mostra o simulador em execução, juntamente com todos os processos carregados no *kernel*. Os processos criados no modelo aparecem selecionados.

Como podemos observar, os dois processos executam de forma correta, pois o consumidor recebe os dados do produtor de forma sequencial em um intervalo de zero a dez, conforme especificado.

3.5 Conclusões do Capítulo

Neste capítulo foi apresentado um método de desenvolvimento de arquiteturas de controle de veículos submarinos. Tal método baseia-se no uso de métodos formais para a especificação do sistema, além de ferramentas para checagem tanto



(a) Tasks carregadas kernel do simulador

(b) Simulador em execução

Figura 3.5: Execução do modelo Produtor / Consumidor no VxWorks.

do modelo desenvolvido na linguagem CSP-OZ quanto de sua implementação em código na linguagem RavenSPARK.

A utilização de ferramentas de checagem de modelos permite que eventuais erros cometidos durante o desenvolvimento do sistema sejam descobertos no início do projeto, o que torna mais simples as devidas correções. Além da verificação do comportamento dinâmico da arquitetura (feito através da parte CSP das especificações com a ferramenta FDR), também é checada a implementação do modelo com relação a uma série de erros comuns durante o desenvolvimento de software. Estas checagens são feitas por meio das ferramentas da linguagem SPARK, o que resulta em um código mais robusto e eficiente.

Também foi visto neste capítulo uma forma de se implementar em código os canais de CSP. Deste modo, a parte dinâmica de modelos em CSP-OZ que foram verificados no FDR podem ser implementados sem grandes dificuldades.

4 Sistema para Controle de ROVs

Neste capítulo será apresentado o sistema para operação de veículos submarinos do tipo ROV em desenvolvimento, incluindo os requisitos de projeto e seus componentes. Como será visto a seguir, este sistema é basicamente regido por dois softwares: um para a estação base de controle do veículo e um embarcado no mesmo, que consiste em sua arquitetura de controle. Uma descrição mais detalhada desta arquitetura poderá ser vista em sua especificação formal em CSP-OZ no capítulo 5. Já para a estação base, como esta não faz parte do escopo deste trabalho, os detalhes de sua modelagem e implementação em código não serão apresentados. O mesmo vale para o módulo de controle dos propulsores, que será implementado em trabalhos futuros.

4.1 Descrição Geral

Basicamente, o sistema para operação de ROVs é composto pelo veículo submarino em conjunto com uma estação base localizada na superfície, sendo a comunicação entre os dois feita por meio de um cabo umbilical. É por meio da estação base que o operador do veículo recebe as informações do veículo e envia seus comandos de atuação. A figura 4.1 mostra um esquema de funcionamento deste sistema, com um navio de apoio onde estão localizados a estação base e o operador do veículo, e também o veículo em funcionamento inspecionando um oleoduto. Esta configuração é muito utilizada em missões de inspeção visual na indústria do petróleo, mas não é a única. Como a estação base é portátil, o veículo pode ser controlado de qualquer lugar na superfície em inspeções de barragens, lagos, etc.

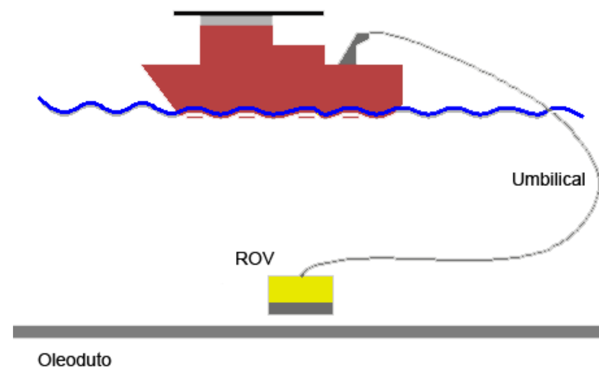


Figura 4.1: Esquema de monitoramento com ROVs.

4.2 Requisitos

A fim de se definir os limites e funcionalidades a serem implementadas no projeto, foram estabelecidos requisitos relacionados ao modo de operação e movimentação do veículo, bem como requisitos temporais e de desempenho do sistema, que serão descritos a seguir.

4.2.1 Modos de Operação

A arquitetura de controle deve permitir que o robô seja controlado de acordo com dois modos de operação:

- *Remoto*: O veículo é operado por um operador, que monitora o seu estado e envia sinais de atuação por meio de uma estação de controle. Apesar de estar sendo controlado pelo operador, o robô possui um comportamento reativo. Isso porque ele é capaz de ignorar estes sinais de comando de acordo com decisões tomadas internamente baseadas nas informações dos sensores embarcados, principalmente dos sensores que detectam obstáculos como sonares, *lasers*, etc. Essa reatividade é necessária por questões de segurança, a fim de evitar colisões com objetos do ambiente durante a operação;

- *Autônomo*: O veículo opera sem nenhum tipo de intervenção do operador, utilizando apenas uma missão previamente programada e as leituras de seus sensores embarcados para identificar características do ambiente e tomar as decisões necessárias para o cumprimento da missão. Assim como no modo remoto, ele deve ser capaz de navegar no ambiente autonomamente evitando colisões.

A arquitetura de controle será desenvolvida de modo a permitir estes dois modos de operação. Entretanto, neste trabalho só será implementado o modo Remoto, sendo que o bloco responsável pelo modo Autônomo será apenas posicionado no modelo da arquitetura de modo a possibilitar sua implementação em trabalhos futuros.

4.2.2 Modos de Movimentação

Apesar do veículo ser do tipo holonômico[1], ele terá alguns de seus movimentos restritos, operando com apenas quatro graus de liberdade. O objetivo disso é tornar mais fácil sua operação por meio de um *joystick* comum, utilizado em simuladores de vôo. Sendo assim, sua movimentação se restringe a:

- Movimentos no plano XY;
- Movimentos no eixo Z;
- Rotação em torno do eixo Z;

Portanto, a estação base deverá ser capaz de enviar apenas comandos de velocidade linear nos eixos X, Y e Z, além do comando de velocidade de rotação em torno do eixo Z.

4.2.3 Requisitos de Desempenho

Com relação aos aspectos de desempenho, temos os seguintes requisitos:

- O robô deve ser capaz de permanecer parado, mesmo na presença de correnteza.
- O robô sempre deve permanecer estável durante a sua movimentação, ou seja, deve ter atitude e altitude controladas;
- A profundidade de operação deverá ser de até 300 metros;
- O tempo de duração da missão será indeterminado, uma vez que este opera por meio de um cabo umbilical transmitindo energia e portanto não é necessário utilizar baterias.

[1] Robôs holonômicos são capazes de se movimentar em qualquer direção, não apresentando restrições de movimentação. Um exemplo de veículo não-holonômico é um automóvel, que não pode executar movimentos laterais, por exemplo.

4.2.4 Requisitos Temporais

No trabalho de Amianti (2008), foi feita uma análise do tempo de resposta do ser humano para avaliar os atrasos admissíveis no sistema de controle de VANTs[2]. Os requisitos temporais para operação deste tipo de veículo são muito mais críticos do que os para ROVs, uma vez que a dinâmica destes últimos é bem mais lenta que os de um VANT. Entretanto, estes requisitos temporais serão os adotados neste trabalho, sendo eles:

- Frequência de atualização dos sensores: 10 Hz;
- Frequência de atualização da estação base: 20 Hz;
- Delay máximo no sistema: 175 ms;

Estes requisitos serão utilizados apenas na implementação da arquitetura de controle, não aparecendo nas especificações em CSP-OZ. Isso porque o CSP utilizado neste trabalho não leva em conta aspectos temporais. Para que isso pudesse ser feito, deveria ser utilizada uma variação de CSP, denominada *Timed-CSP* (SCHNEIDER, 1999). Assim, aspectos temporais também não serão tratados nas análises feitas com o FDR.

4.3 Componentes do Sistema

4.3.1 O Robô

A arquitetura de controle desenvolvida deverá ser aplicada futuramente no robô submarino desenvolvido no laboratório de Sistemas Embarcados do departamento de Engenharia Mecatrônica da POLI-USP(figura 4.2), cujo objetivo será atuar em missões de inspeção visual em baixas profundidades.

Ele possui uma série de sensores embarcados, cujos nomes juntamente com as variáveis medidas, taxas de amostragem e precisão dos mesmos são apresentados na tabela 4.1. Posteriormente será adicionado a este conjunto uma câmera para a realização de inspeções visuais.

Como neste trabalho só será implementado o modo Remoto de operação, serão utilizados apenas dois destes sensores embarcados: o Altímetro e a Bússola. Estes compõem o conjunto mínimo necessário para fornecer as informações utilizadas neste modo de operação. Os demais sensores deverão ser adicionados na

[2] Sigla para Veículo Aéreo Não-Tripulado

Figura 4.2: Robô utilizado no trabalho.

Tabela 4.1: Conjunto de sensores embarcados no ROV.

Variável	**Sensor (Fabricante)**	**Precisão**	**Atualização**
Direção	Bússola TCM2 (PNI)	1°	13 Hz
Roll e Pitch	Tilt Series 757 (Applied Geomechanics)	2°	13 Hz
Profundidade	Sensor de Pressão MPX5100DP (Motorola)	3,5 cm	20 Hz
Taxa de giro	Giro de Fibra Ótica E-Core 2000 (KVH)	Bias $< 2^o$/h	10 Hz
Altura	Altímetro PA-200 (Tritech)	1 mm	10 Hz
Aceleração Linear	IMU VG700A (Crossbow)	Bias $<$ 12 mg	100 Hz
Aceleração Angular	IMU VG700A (Crossbow)	Bias $< 20^o$/h	100 Hz

arquitetura em uma implementação futura, juntamente com o modo Autônomo de operação.

4.3.2 Estação Base

A fim de se implementar o modo de operação remota, foi desenvolvida uma estação base para controle do ROV. É por meio desta que o operador recebe e envia todas as informações necessárias para controlar o robô remotamente, ou seja, à distância. Basicamente, ela é composta por um *joystick* conectado a um computador, que permite que o operador forneça os comandos de atuação no robô. Além deste, ela também possui uma interface gráfica que contém informações referentes aos sensores embarcados no robô. A comunicação entre eles é feita por meio de um cabo umbilical, que transmite tanto estes dados de telemetria e vídeo quanto energia.

No diagrama de hardware mostrado na figura 4.5, a estação base corresponde ao *BaseStation_Module*. Para a implementação deste módulo utilizou-se a linguagem Java. A escolha por esta linguagem foi feita por uma questão de praticidade. Isso porque ela apresenta diversos recursos necessários a essa aplicação ou já implementados na própria linguagem ou então por meio de bibliotecas, como é o caso

da aquisição dos sinais de vídeo e do *joystick*, manipulação de imagens, criação de *sockets* para comunicação UDP/IP, entre outras. Além disso, Java implementa os recursos básicos para programação de sistemas de tempo real, como criação e manipulação de *Threads* e leituras de tempo no sistema.

A principal desvantagem do uso de Java em sistemas de tempo real é a sua falta de determinismo nos tempos de execução, principalmente devido ao *garbage collector* (SCHOEBERL, 2004). Este é um programa executado pela máquina virtual Java responsável pelo gerenciamento do uso de memória, que identifica regiões não mais utilizadas pelo programa e possibilita seu reuso. Durante sua execução, os programas em execução entram em espera, retornando apenas após o término das tarefas do *garbage collector*. O problema é que sua execução não é determinística, não sendo possível programar os instantes de sua execução e nem o tempo gasto neste processo. A fim de se evitar este problema, sempre que possível foram eliminadas alocações dinâmicas de objetos. Isso reduz significativamente a interferência do *garbage collector* no sistema.

4.3.2.1 Interface Gráfica

A interface gráfica da estação de controle deve exibir todas as informações necessárias à operação do veículo. Estas informações podem ser de vários tipos, como imagem da câmera de navegação, nível das baterias, direção do veículo, profundidade, entre muitas outras, dependendo apenas dos sensores embarcados. Neste trabalho a interface será otimizada para o modo de operação remota, e por isso serão mostradas nela as seguintes informações:

- *Depth*: Profundidade de operação, em metros;
- *Direction*: Direção do veículo, em graus;
- *Power*: Potência aplicada nos motores, em porcentagem;
- Data e tempo decorrido da missão;

A figura 4.3 mostra a interface gráfica [3] desenvolvida para a estação de controle. A exibição das imagens de vídeo recebidas de uma eventual câmera embarcada já está implementada.

[3] A imagem de fundo na interface é meramente ilustrativa. Esta será substituída pelas imagens da câmera embarcada do ROV, quando esta for instalada.

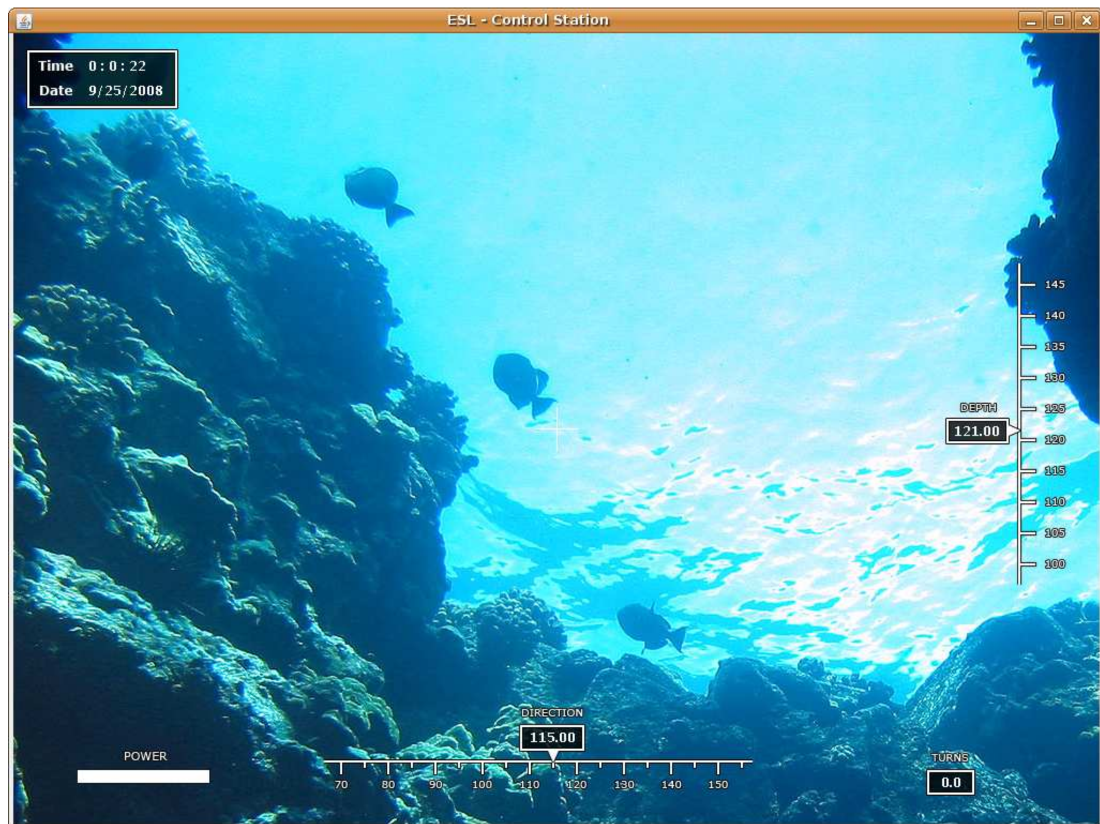


Figura 4.3: Interface gráfica de controle do ROV.

4.3.2.2 Comandos do *Joystick*

No modo de operação remota, o operador gera os comandos de atuação do robô por meio de um *joystick*. Estes comandos são representados na forma de velocidade nos graus de liberdade permitidos. Portanto, de acordo com os requisitos de movimentação definidos na seção 4.2, estas velocidades são as lineares nos eixos X, Y e Z bem como a velocidade de rotação em torno de Z. A figura 4.4 apresenta o modo de inserção de comandos com o joystick *Microsoft Sidewinder Force Feedback* que será utilizado neste trabalho.

O operador é capaz de controlar a potência dos movimentos do veículo através da alavanca *Power*. Assim, os valores das velocidades enviadas podem ser limitadas dentro de um intervalo entre 0 e 100% dos valores lidos no *joystick*.

4.3.2.3 Troca de Mensagens

A troca de mensagens entre a estação base e o ROV foi implementada utilizando-se o protocolo de rede UDP/IP, sendo que as mensagens enviadas apresentam um formato e uma semântica pré-definidos. Ambos enviam suas informações a uma frequência fixa de 10 Hz, de acordo com os requisitos temporais estabelecidos na seção 4.2.4.

Figura 4.4: Comandos de atuação do ROV.

Com relação ao formato, as mensagens trocadas entre a estação de controle e o robô são do tipo *$IDvalorIDvalorIDvalorIDvalor...$* , onde:

- *$*: Caractere de início e fim de mensagem
- *ID*: Caractere Identificador da variável;
- *valor*: Valor da variável, com precisão de duas casas decimais.

Um exemplo de mensagem válida transmitida do robô para a estação base poderia ser *$d170.00h20.00$*, significando que o robô está a cem metros de profundidade, orientado a vinte graus. Uma mensagem válida da estação base para o veículo poderia ser por exemplo *$x1.00y0.50z0.23t-0.80$*.

Esse formato permite que as mensagens sejam expansíveis, de tamanho variável e que os dados possam ser enviados em qualquer ordem. Isso possibilita a inserção de novos sensores ou outros tipos de informações na mensagem com muita facilidade, bastando atribuir às novas variáveis um identificador válido. Considera-se como identificador válido qualquer caractere com exceção dos caracteres numéricos, do ponto ('.'), da vírgula (','), dos sinais de mais e menos ('+' e '-') e das letras 'E' e 'e', que são reservados para uso na descrição de números em notação científica.

A semântica de comunicação adotada neste trabalho pode ser vista na tabela 4.2, que contém os caracteres identificadores de cada variável usada nesta implementação. Trata-se de uma semântica muito simples, que contém apenas seis elementos. Porém, em uma implementação futura desta arquitetura com o modo Autônomo e a utilização de todos os sensores embarcados no robô, ela pode

Tabela 4.2: Semântica de Comunicação.

ID	Variável
d	direção
h	profundidade
x	velocidade em X
y	velocidade em Y
z	velocidade em Z
t	velocidade de rotação em Z

facilmente ser expandida, bastando apenas estabelecer um caractere identificador para cada nova variável transmitida na mensagem.

4.4 Arquitetura de Hardware

Do ponto de vista de hardware, o sistema pode ser dividido entre os componentes presentes na superfície e no sistema embarcado. Este último será desenvolvido utilizando uma arquitetura distribuída composta por três módulos de processamento: um computador com maior poder de processamento do tipo PC104, onde será executada a arquitetura de controle do ROV desenvolvida neste trabalho; dois microcontroladores do tipo ARM7, cada um responsável pelo controle de um conjunto de quatro propulsores do veículo.

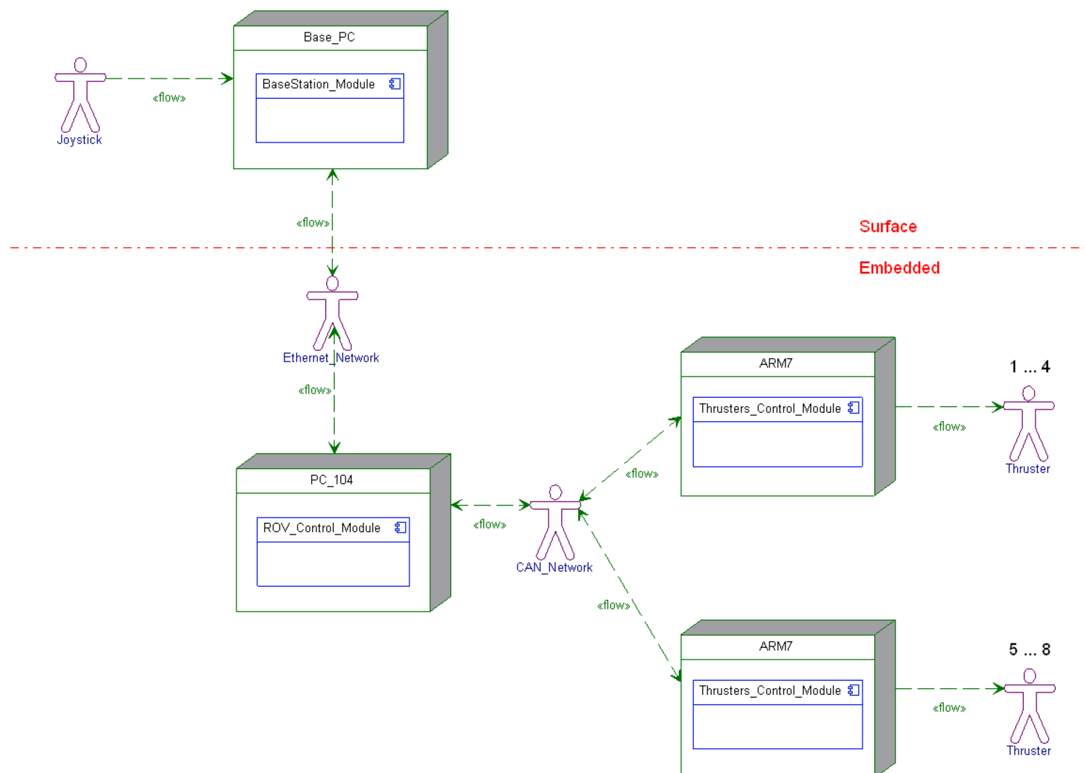


Figura 4.5: Módulos de Software presentes no Sistema.

A figura 4.5 mostra esta arquitetura, juntamente com seus modos de comunicação e os módulos de software que são executados em cada componente. Tanto neste diagrama como no da figura 4.6, os atores representam elementos de hardware presentes no sistema que fornecem, transmitem ou recebem informações, e portanto englobam os sensores embarcados no robô e o controle utilizado pelo operador, as redes CAN e Ethernet e os propulsores do robô, respectivamente.

Conforme mostrado na figura 4.5, em cada um dos componentes de hardware do sistema deve ser executado um módulo de software. Do ponto de vista de software, o sistema foi dividido em três módulos: um para a estação base (*BaseStation_Module*), um para controle do veículo (*ROV_Control_Module*) e outro para o controle de seus propulsores (*Thrusters_Control_Module*). A figura 4.6 mostra os requisitos de cada um dos módulos bem como seus meios de comunicação por meio de um diagrama UML de casos de uso.

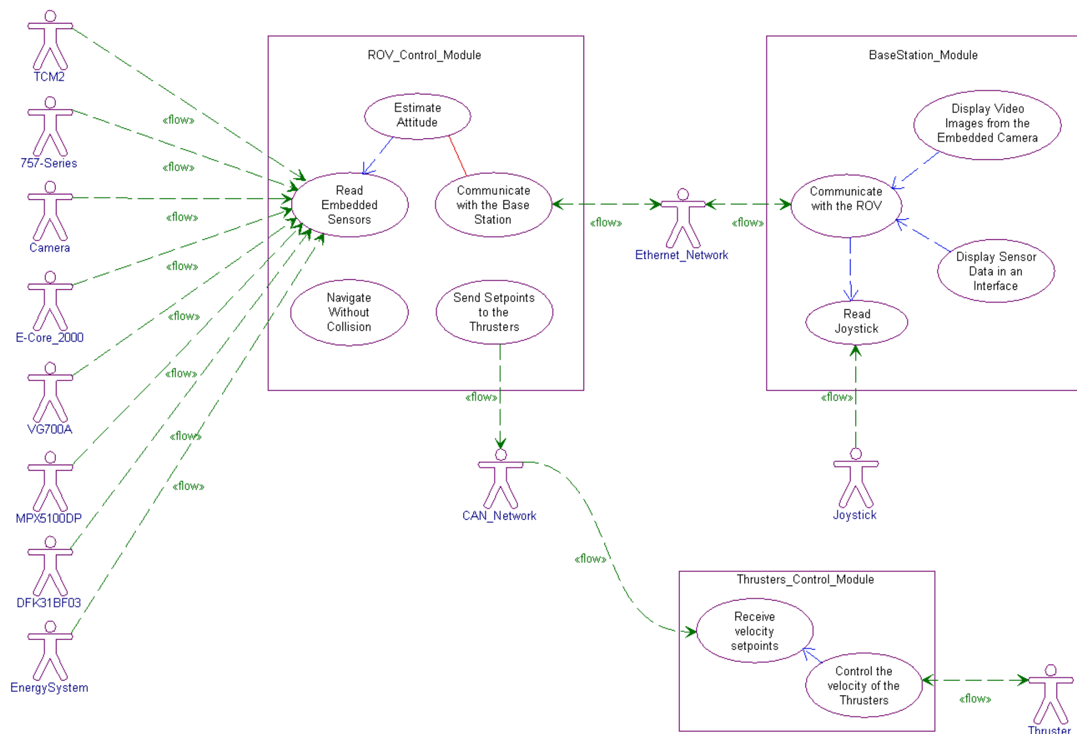


Figura 4.6: Casos de Uso dos Módulos de Software do Sistema.

Um ponto importante a se destacar consiste no escopo do trabalho. Como o objetivo é desenvolver uma metodologia que permita a checagem de modelos de arquiteturas de controle para ROVs, apenas o módulo *ROV_Control_Module* será especificado formalmente e analisado segundo a metodologia proposta. Os demais módulos não serão nem especificados e nem analisados segundo esta abordagem, e por isso serão implementados utilizando linguagens e ferramentas diferentes das utilizadas na metodologia.

4.5 Conclusões do Capítulo

Neste capítulo foi apresentado o sistema de controle de veículos submarinos atualmente em desenvolvimento no Laboratório de Sistemas Embarcados da Escola Politécnica da Universidade de São Paulo. Neste trabalho está sendo desenvolvida a parte de software da estação base e a arquitetura de controle embarcada no veículo.

A comunicação entre o sistema embarcado e a estação base é feita por meio de uma semântica de comunicação definida exclusivamente para esta implementação. Entretanto, esta pode ser facilmente expandida de modo a acomodar a inserção de novos sensores no sistema, bem como possibilitar o envio de outros tipos de informações. Além de sua extensibilidade, sua codificação e decodificação é facilmente implementável. Neste caso, para este trabalho foram escritos dois codificadores/decodificadores: um em Java (para a estação base) e um em RavenSPARK (para o sistema embarcado).

5 Especificação da Arquitetura do Software Embarcado

Neste capítulo será apresentada a especificação formal da arquitetura de controle do veículo submarino apresentado na seção 4.3.1. Ela consiste em uma combinação de textos em linguagem natural com especificações formais em CSP-OZ. O objetivo dessa combinação é facilitar a compreensão das especificações em CSP-OZ por meio de explicações sempre que pertinente utilizando linguagem natural, ou seja, textos em português. Todas as especificações em CSP-OZ foram feitas em LaTeX.

5.1 Introdução

O desenvolvimento da arquitetura foi feito segundo o paradigma híbrido (MURPHY, 2000), sendo portanto dividida em dois níveis conceituais (figura 5.1): uma camada *deliberativa*, onde se encontram as funções relacionadas às tomadas de decisão do robô, e portanto é a responsável por indicar ao robô qual o seu próximo movimento, e; uma camada *reativa*, onde estão encapsuladas as funções de proteção e movimentação do robô, juntamente com a interação com os sensores embarcados.

A figura 5.1 mostra uma representação gráfica da especificação em CSP da arquitetura desenvolvida, feita na ferramenta gCSP (JOVANOVIC et al., 2004). Neste diagrama, todos os elementos representados por um retângulo são processos, que executam de maneira independente entre si. O único ponto de interação entre eles é feito por meio de canais de comunicação unidirecionais, representados por meio das setas no diagrama. Nos canais, as setas representam a direção da passagem dos dados. Os nomes também indicam a direção da passagem de dados e o tipo de dado transmitido, sendo sempre nomeados da seguinte forma: *origem_destino:tipo*. Pode-se notar que os processos *Compass*, *Altimeter* e *Sonar* não possuem canais de comunicação. Isso porque eles não se comunicam diretamente com os demais processos da arquitetura. Eles captam as informações

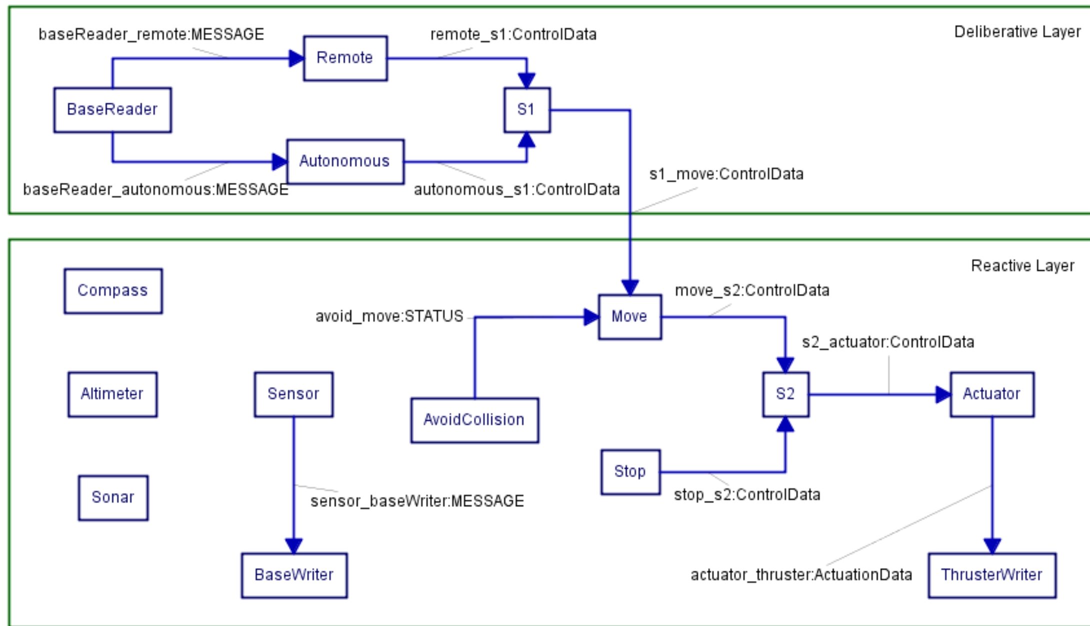


Figura 5.1: Arquitetura de Software - Canais e Estruturas de Dados.

de seus respectivos sensores e armazenam em uma estrutura de dados denominada *BlackBoard*, que será apresentada a seguir. Isso permite que os dados do *BlackBoard* sejam usados para leitura e escrita de maneira assíncrona, ou seja, o *BlackBoard* é utilizado como uma região de memória para transferência assíncrona de dados entre diferentes processos.

A seguir serão descritos todos os processos presentes na arquitetura, juntamente com os tipos básicos e as estruturas de dados definidos e utilizados ao longo da implementação.

5.2 Definições Gerais

Antes de elaborar as especificações dos processos presentes na arquitetura, foram definidos alguns elementos que serão utilizados ao longo das especificações como tipos básicos e estruturas de dados, além de algumas definições sobre processos periódicos, que serão apresentados a seguir.

5.2.1 Tipos Básicos de Dados

Para esta implementação foram definidos alguns tipos de dados, de modo a facilitar a elaboração das especificações dos processos que compõem a arquitetura. Para cada tipo de dado, está associado a ele um intervalo de valores válidos, de modo que tal intervalo não precise ser descrito novamente ao longo das especificações. Neste trabalho, essa definição de tipos de dados será posteriormente

transformada em tipos de dados da linguagem Ada, o que permitirá o aproveitamento das vantagens da programação utilizando-se tipagem forte, como por exemplo checagens automáticas de condições de existência de variáveis ao longo do programa. Assim, foram definidos os seguintes tipos básicos:

- *COMPASS_TYPE* $::= \{x : \mathbb{R} \mid 0 \leq x \leq 360\}$
  Representa as leituras da bússola, em graus. Seu intervalo de valores possíveis vai de zero a trezentos e sessenta.

- *ALTIMETER_TYPE* $::= \{x : \mathbb{R} \mid \ 0 \leq x \leq 300\}$
  Representa as leituras do altímetro, em metros. Para esta implementação, o valor máximo de profundidade foi limitado a trezentos metros, de acordo com os requisitos estabelecidos no capítulo 4.2.

- *SPEED_TYPE* $::= \{x : \mathbb{R} \mid -1 \leq x \leq 1\}$
  Representa uma velocidade, tanto linear quanto angular. Seu valor é normalizado, variando de menos um a um (velocidades máximas em dois sentidos). Assim, este tipo de dado pode ser utilizado para representar velocidades tanto lineares quanto angulares.

- *TORQUE_TYPE* $::= \{x : \mathbb{R} \mid -100 \leq x \leq 100\}$
  Representa a referência de torque (em N*m) de um propulsor a ser enviada via rede para o controlador dos propulsores. De posse desta referência, o controlador obtém a rotação final do mesmo.

- *STATUS* $::=$ *normal* | *alert*
  Indica o estado do robô com relação à presença de obstáculos no ambiente. Caso não haja nenhum obstáculo detectado ou o robô esteja abaixo do limite de proximidade de algum destes, o estado indicado seria *normal*, enquanto que caso o robô esteja no limite de proximidade ou acima dele, o estado indicado seria *alert*.

Além destes, temos ainda um tipo de dado para representar o conjunto completo de sonares do robô e as mensagens trocadas entre este e a estação base. No caso do sonar, o valor de cada leitura individual está normalizado, variando de zero (ausência de obstáculos) a um (robô no limiar de proximidade de um obstáculo). Como o robô utilizado neste trabalho não possui nenhum sonar, apenas para fins de demonstração será adotado nesta especificação um conjunto composto por quatro sonares, localizados nas posições esquerda, frontal, direita e inferior do veículo, nesta ordem. Portanto temos:

SONAR_TYPE

$readings : \text{seq}\,\mathbb{R}$	[set of all embedded sonar readings]
$\forall\, i \in 0 \,..\, 3 \bullet 0 \leq readings_i \leq 1$	[restrict to four elements]

Por fim, as mensagens trocadas entre a estação base e o robô são representadas pelo seguinte tipo básico:

[MESSAGE]: Conjunto de todas as mensagens válidas possíveis de serem enviadas entre a estação base e o sistema embarcado, de acordo com a semântica estabelecida previamente na seção 4.3.2.3. Sendo assim, todas as mensagens apresentam a seguinte forma:

$$\langle \$ \rangle \frown \langle id \rangle \frown \langle value \rangle \frown \langle id \rangle \frown \langle value \rangle \frown \cdots \frown \langle \$ \rangle.$$

5.2.2 Estruturas de Dados

Com base nos tipos básicos definidos acima foram criadas três estruturas de dados, que por sua vez também são outros tipos de dados, porém mais complexos, uma vez que representam um conjunto de um ou mais destes tipos básicos.

A primeira delas é a estrutura utilizada para armazenar os dados dos sensores embarcados, definida pelo tipo *SensorData*. Nela estarão presentes as leituras da bússola, (variável de estado *direction*), do altímetro (variável *depth*) e do sonar. Vale lembrar que este último sensor não está presente no robô utilizado, sendo que sua presença na arquitetura serve apenas para ilustrar o modelo de reatividade implementado.

SensorData

$direction : COMPASS_TYPE$
$depth : ALTIMETER_TYPE$
$sonar : SONAR_TYPE$

INIT

$direction = 0$
$depth = 0$
$\forall\, i \in 0 \,..\, 3 \bullet sonar_i = 0$

Outra estrutura de dados importante é a *ControlData*, utilizada para armazenar as informações de controle do veículo. Ela é composta de quatro atributos que correspondem às velocidades de translação nos três eixos mais a rotação em torno do eixo Z.

ControlData

	Init
$vx : SPEED_TYPE$	$vx = 0$
$vy : SPEED_TYPE$	$vy = 0$
$vz : SPEED_TYPE$	$vz = 0$
$vTheta : SPEED_TYPE$	$vTheta = 0$

Por fim, temos a estrutura de dados utilizada para armazenar o torque a ser aplicado em cada um dos oito propulsores do veículo, representada pela classe *ActuationData*.

ActuationData

	Init
$thruster1 : TORQUE_TYPE$	$thruster1 = 0$
$thruster2 : TORQUE_TYPE$	$thruster2 = 0$
$thruster3 : TORQUE_TYPE$	$thruster3 = 0$
$thruster4 : TORQUE_TYPE$	$thruster4 = 0$
$thruster5 : TORQUE_TYPE$	$thruster5 = 0$
$thruster6 : TORQUE_TYPE$	$thruster6 = 0$
$thruster7 : TORQUE_TYPE$	$thruster7 = 0$
$thruster8 : TORQUE_TYPE$	$thruster8 = 0$

5.2.3 Processos Periódicos

Com relação ao modo de execução, os processos podem ser periódicos ou esporádicos. Os periódicos têm sua execução repetida em intervalos de tempo pré-definidos, enquanto que os esporádicos são executados sempre que solicitado. Estes últimos serão sincronizados através dos canais de comunicação em que estão conectados, como ocorre em CSP. Nesta arquitetura temos os dois tipos de processos, sendo que os periódicos são o *Compass*, *Altimeter*, *Sonar*, *Sensor*, *AvoidCollision* e *Stop*, enquanto que os demais são esporádicos.

Nos processos periódicos, a leitura do *clock* do computador será realizada a partir do método *Clock*. O *clock* é um valor sempre crescente, utilizado para se estimar a passagem de tempo. Portanto, essa passagem de tempo no sistema pode ser estimada a partir da diferença nas leituras do *clock* depois e antes da realização de determinada atividade. Assim, definimos o seguinte método:

Clock

$clock! : \mathbb{N}$

[get the computer clock in miliseconds]

Outro aspecto importante das tarefas periódicas consiste no modo em que o tempo de espera, ou *delay*, é utilizado. Existem dois modos:absoluto e relativo. Esse tempo de espera é utilizado para liberar tempo de processamento para que os demais processos do sistema possam ser executados.

O *delay* relativo consiste em um tempo de espera fixo, que é iniciado após o término de um ciclo de execução de um processo. A utilização deste modo resulta em uma frequência de execução variável, uma vez que o tempo total entre as execuções é a soma do tempo estipulado para o *delay* mais o tempo gasto na execução do processo, sendo que este último em geral não é conhecido e também pode variar em cada ciclo de execução. A fim de se obter uma execução periódica, deve ser utilizado o *delay* absoluto. Neste modo, o tempo de espera é obtido com base apenas no período desejado, descontando-se o tempo de execução do processo. Para as especificações em CSP-OZ, será utilizada a instrução $Delayuntil(time)$, onde $time$ representa o instante no tempo que o processo irá iniciar novamente seu ciclo de execução.

Neste trabalho, todos os processos periódicos utilizarão *delays* absolutos. A fim de encapsular todas as características de tais processos, será definida a classe $Periodic$. Nela temos a constante $PERIOD$, que representa o período de execução desejado, e a variável $nextExecution$, que representa o instante de tempo da próxima execução. O cálculo deste valor é feito no método $getNextExecution$, que simplesmente soma o período desejado para o processo com o instante de tempo do início do processamento. Deste modo o tempo gasto na execução do processo não interfere no início do próximo ciclo de processamento, como ocorre com *delays* relativos.

Periodic

This class encapsulates all the attributes and methods
necessary for use in periodic processes

$PERIOD : \mathbb{N}$ [desired period, in miliseconds]

$PERIOD = 100$

$nextExecution : \mathbb{N}$ [time of the next execution]

$nextExecution > 0$

INIT

$nextExecution = Clock$ [starts as soon as possible]

getNextExecution

$\Delta(nextExecution)$

[calculates the next execution as absolute delay]
$nextExecution' = nextExecution + PERIOD$

Aqui temos dois pontos importantes a destacar. O primeiro deles consiste na análise temporal do sistema desenvolvido. Isso não será feito neste trabalho, sendo que a discussão sobre a periodicidade dos processos e a construção *Delayuntil*(*time*) servem apenas como orientação para a implementação em RavenSPARK, e não para as análises feitas no modelo. Portanto, não serão considerados aspectos temporais de CSP, e as análises feitas no FDR levarão em conta apenas os eventos ocorridos, desconsiderando o momento de suas ocorrências.

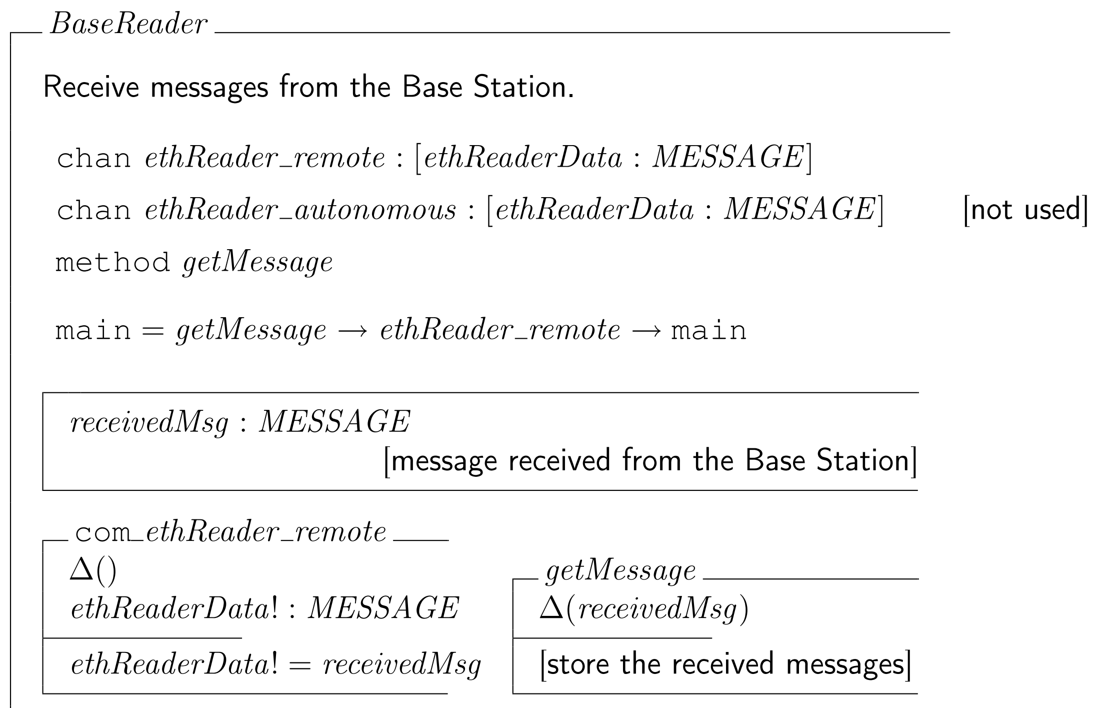
O segundo consiste na repetição de nomes de canais nos processos. Em CSP, canais (ou eventos) de mesmo nome em processos diferentes atuam como pontos de sincronização entre os processos. No caso do método *getNextExecution*, este não é o comportamento desejado, pois esse é um evento interno na execução dos processos periódicos. Assim, no *script* em CSP_M, este deve possuir um nome diferente em cada processo a fim se serem feitas corretamente as análises no FDR. Sempre que isso ocorrer, o canal com nome modificado será indicado com um comentário na frente de seu nome no *script* em CSP_M.

5.3 Camada Deliberativa

A camada deliberativa é a responsável por estabelecer o próximo movimento do robô a cada iteração do ciclo de controle. Ela é composta de quatro processos, todos esporádicos, e que serão descritos a seguir.

5.3.1 BaseReader

Este é o processo responsável pela recepção de mensagens vindas da estação base. Neste caso, as mensagens são transmitidas através de uma rede ethernet pelo protocolo UDP/IP, e portanto este processo possui um *socket* para recepção de mensagens neste protocolo. Porém, esta recepção será abstraída no modelo, e portanto não fará parte da especificação em CSP-OZ. Logo, o método *getMessage* é tratado apenas como uma caixa preta que armazena na variável *receivedMsg* a mensagem recebida pela rede.



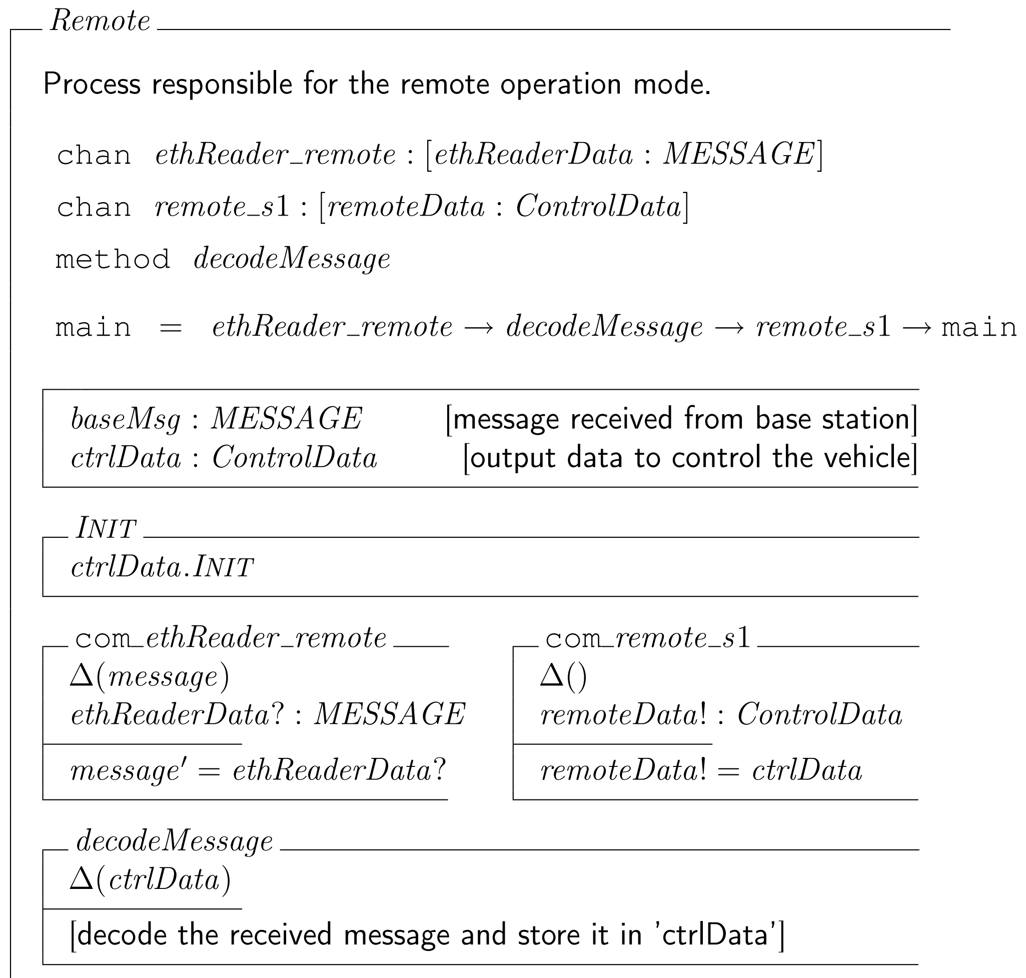
Assim como todos os processos neste trabalho, ele executa em um ciclo infinito, com seu comportamento indicado pelo processo `main`. Ele tem sua execução sincronizada com a recepção de mensagens da rede ethernet. Isso porque método *receiveMessage* é bloqueante, ou seja, o processo fica inativo aguardando uma mensagem, e só reinicia sua execução com a chegada de uma nova mensagem da rede.

Como não será implementado o modo autônomo neste trabalho, o processo *EthernetReader* envia diretamente as mensagens recebidas ao processo *Remote*, e portanto o canal de comunicação com o processo *Autonomous* não será utilizado, tendo sido colocado na especificação apenas visando sua implementação em trabalhos futuros.

5.3.2 Remote

Este é o processo responsável pelo modo de operação remota do veículo. Neste modo, o comportamento básico é ficar aguardando novas mensagens da estação base. Assim que chega uma mensagem (que contém as informações de velocidade a serem aplicadas no do veículo), esta é armazenada na variável *baseMsg*, para então ser decodificada e armazenada na estrutura de dados de controle *ctrlData*.

Depois de gerada a informação de controle do veículo, esta é enviada para a camada reativa por meio do supressor *S1*.



5.3.3 Autonomous

Como o modo autônomo não será implementado, a especificação deste processo será composta apenas de sua interface, com comportamento nulo. Esta interface servirá apenas como base para implementações futuras.

Autonomous

Process responsible for the autonomous operation mode, not implemented in this work.

$\texttt{chan}\ \ ethReader_autonomous : [msg : MESSAGE]$
$\texttt{chan}\ \ sensor_autonomous : [sensrData : SensorData]$
$\texttt{chan}\ \ autonomous_s1 : [ctrlData : ControlData]$
$\texttt{main} = \texttt{Skip}$

5.3.4 S1

Este é um processo supressor como descrito no trabalho de Brooks (1986), que faz a seleção das entradas dos processos *Remote* e *Autonomous* de modo que apenas um dos dois seja capaz de atuar na camada reativa. É ele quem controla qual dos dois modos de operação irá atuar no robô em cada ciclo de execução.

S1

This is the supressor process between the Remote and Autonomous processes

$\texttt{chan}\ remote_s1 : [remoteData : ControlData]$
$\texttt{chan}\ autonomous_s1 : [autonomousData : ControlData]$
$\texttt{chan}\ s1_moveToGoal : [s1Data : ControlData]$

$$\texttt{main} = (remote_s1 \rightarrow s1_moveToGoal \rightarrow \texttt{main}) \ \Box \ (autonomous_s1 \rightarrow s1_moveToGoal \rightarrow \texttt{main})$$

$controlData : ControlData$ [to store the transmitted data]

INIT
$controlData.INIT$

$\texttt{com_}remote_s1$
$\Delta(controlData)$
$remoteData? : ControlData$
$controlData' = remoteData?$

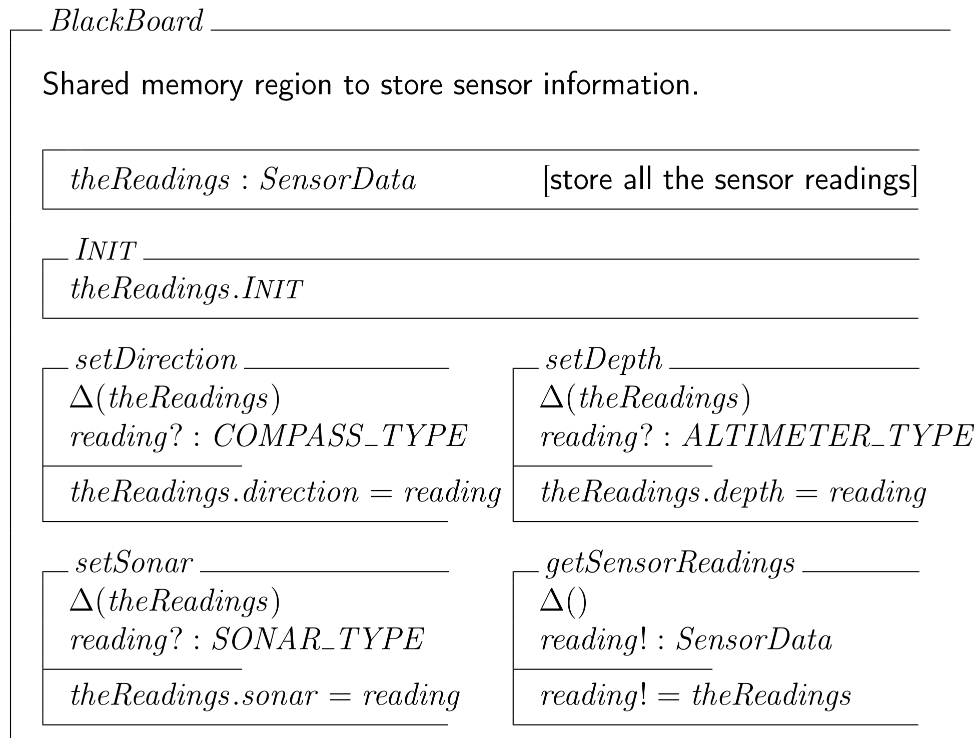
$\texttt{com_}autonomous_s1$
$\Delta(controlData)$
$remoteData? : ControlData$
$controlData' = remoteData?$

$\texttt{com_}s1_moveToGoal$
$\Delta()$
$s1Data! : ControlData$
$s1Data! = controlData$

5.4 Camada Reativa

Nesta camada estão presentes os processos responsáveis pelo sensoriamento, movimentação e proteção do robô, além do envio de dados para a estação base. Assim, alguns destes processos fazem interface direta com o hardware do veículo. Essa interface de leitura de hardware, seja ela dos sensores embarcados ou da rede, será abstraída no modelo e não será especificada formalmente, sendo apenas listado o efeito final de cada operação deste tipo na forma de um comentário, como foi feito no processo *BaseReader*. Um exemplo dessa abordagem pode ser observado no método *getDirection* do processo *Compass*, que apresenta apenas um comentário sobre o efeito da aplicação do método e mostra a variável *direction* em sua lista delta, o que nos leva a concluir que seu efeito final consiste na leitura da bússola sendo armazenada nesta variável. O mesmo vale para todos os métodos desse tipo nos processos *Compass*, *Altimeter*, *Sonar*, *BaseReader*, *BaseWriter* e *ThrusterWriter*, que são os que fazem interface direta com o hardware do veículo.

Uma classe muito importante da arquitetura, mas que não aparece no diagrama da figura 5.1 por não ser um processo, é a *BlackBoard*. Esta consiste em uma estrutura de dados compartilhada entre os processos *Compass*, *Altimeter*, *Sonar* e *Sensor*, que possui acesso protegido com relação à leitura e escrita de seus dados. Isso garante que estes processos possam atualizar e ler as informações dos sensores de maneira assíncrona de forma segura. Assim, os processos *Compass*, *Altimeter* e *Sonar* podem ser executados em frequências diferentes, de acordo com seus respectivos hardwares, enquanto que o processo *Sensor* faz a leitura e envio dos dados para a estação base a uma frequência controlada de 10 Hz. Nesta implementação todos eles trabalharão nesta mesma frequência.



A especificação completa de todos os processos desta camada será apresentada a seguir.

5.4.1 Compass

Processo responsável por realizar as leituras da bússola embarcada no veículo e salvá-las no *BlackBoard*.

Compass

Perform a periodic reading of the compass.

`inherit` *Periodic* [this is a periodic process]

`method` *getDirection*

`method` *saveDirection*

$$\texttt{main} = \texttt{Delay until}(\mathit{nextExecution}) \rightarrow \mathit{getDirection} \rightarrow \mathit{saveDirection} \rightarrow \mathit{getNextExecution} \rightarrow \texttt{main}$$

$\mathit{direction} : \mathit{COMPASS_TYPE}$	[store the compass readings]
$\mathit{blckBoard} : \mathit{Blackboard}$	[reference to the BlackBoard]

INIT

$\mathit{direction}.\mathit{INIT}$

getDirection

$\Delta(\mathit{direction})$

[get the compass reading]

saveDirection

$\Delta(\mathit{blckBoard})$

$\mathit{blckBoard}.\mathit{putDirection}(\mathit{direction})$

5.4.2 Altimeter

Este é o processo responsável pela leitura do altímetro. Assim como nos processos *Compass* e *Sonar*, este possui um método *get* e um *set* do sensor embarcado (neste caso o altímetro), responsáveis por ler o valor do hardware do sensor e por salvá-no no *BlackBoard*, respectivamente.

Altimeter

Perform a periodic reading of the altimeter.

`inherit` *Periodic* [this is a periodic process]

`method` *getDepth*

`method` *saveDepth*

$\mathtt{main} = \mathtt{Delay\ until}(\mathit{nextExecution}) \rightarrow \mathit{getDepth} \rightarrow \mathit{saveDepth} \rightarrow \mathit{getNextExecution} \rightarrow \mathtt{main}$

$depth : ALTIMETER_TYPE$	[store the altimeter readings]
$blckBoard : BlackBoard$	[reference to the BlackBoard]

INIT

$depth.INIT$

getDepth

$\Delta(depth)$

[get the altimeter reading]

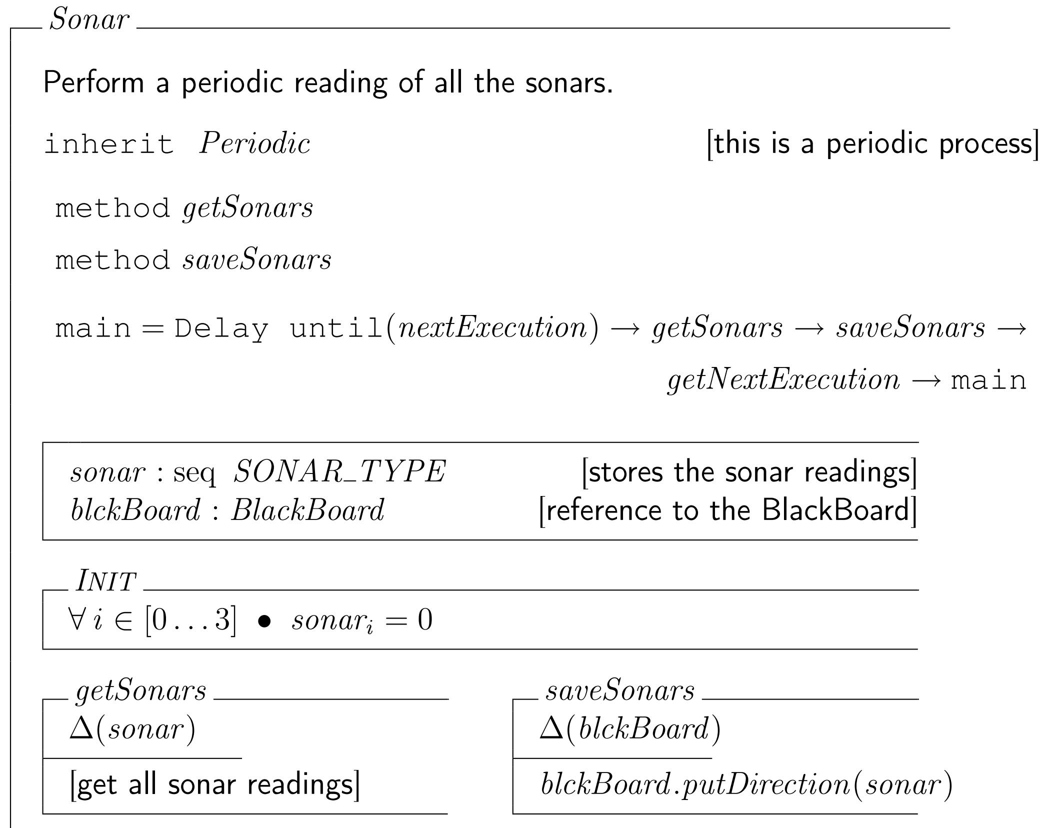
saveDepth

$\Delta(blckBoard)$

$blckBoard.saveDepth(depth)$

5.4.3 Sonar

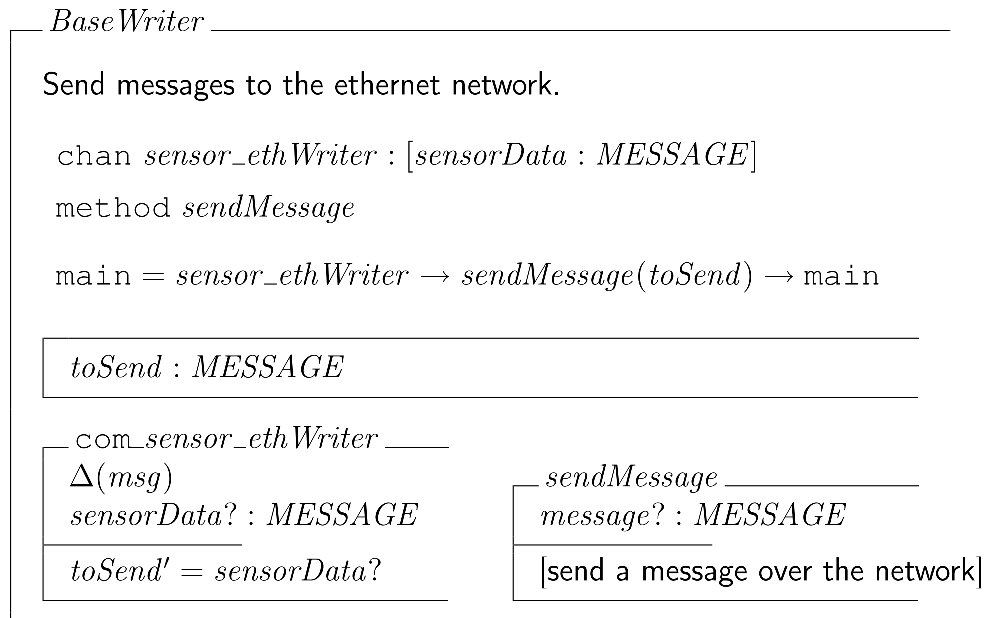
Conforme mencionado anteriormente, será adotado um conjunto de sonares composto por quatro elementos, cobrindo a parte da frente, laterais esquerda e direita, e a parte de baixo do veículo. Isto serve apenas para demonstrar o modelo de reatividade desenvolvido. O processo responsável pela leitura e envio de dados dos sonares é o *Sonar*.



5.4.4 BaseWriter

Este é o responsável por enviar mensagens do veículo para a estação base. Para isso, este processo deve ser capaz de enviar mensagens na rede ethernet por meio do protocolo UDP/IP. Assim como no processo *BaseReader*, essa comunicação em baixo nível com a rede não será especificada, e portanto o método *sendMessage* será tratado como uma caixa preta capaz de realizar o envio da mensagem.

Um ponto importante a se destacar consiste no nome deste método no *script* CSP_M para análise o FDR. Como o processo *ThrusterWriter* também possui um método com este nome, no *script* eles estarão representados com nomes diferentes, apenas para não serem considerados pelo FDR como eventos sicronizadores entre os processos *BaseWriter* e *ThrusterWriter*, o que não deve acontecer pois estes são eventos internos e independentes de cada processo.



5.4.5 ThrusterWriter

Este é o processo responsável por enviar as informações de atuação ao controlador dos propulsores. Na implementação final do sistema, a comunicação entre esse processo e o controlador será feita por meio de uma rede CAN. Porém, como ainda não existe uma rede CAN implementada no robô, neste trabalho não será implementado o envio das mensagens para o controlador. Portanto, em sua especificação abstrata o método *sendMessage* irá enviar a mensagem pela rede CAN, mas em sua implementação ele irá apenas mostrar na tela a mensagem a ser enviada.

Para a comunicação com o controlador foram definidos oito códigos identificadores dos propulsores do veículo, indo do caractere "i" ao "p", sendo que o primeiro caractere se refere ao propulsor de número um, e o último caractere ao de número oito, respectivamente. Assim, o método *generateMessage* monta uma mensagem de acordo com o padrão estabelecido para o tipo MESSAGE especificado anteriormente.

ThrusterWriter

Send messages to the Thrusters controller.

$\mathtt{chan}\ actuator_thruster : [actuatorData : ActuationData]$

$\mathtt{method}\ generateMessage$

$\mathtt{method}\ sendMessage$

$$\mathtt{main} = actuator_thruster \rightarrow generateMessage(setpoints) \rightarrow sendMessage(controllerMsg) \rightarrow \mathtt{main}$$

$setpoints : ActuationData$	[store the setpoints to the thrusters]
$controllerMsg : MESSAGE$	[message to the controller]

INIT

$actuationData.INIT$

$\mathtt{com}_actuator_thruster$

$\Delta(setpoints)$

$actuatorData? : ActuationData$

$setpoints' = actuatorData?$

sendMessage

$\Delta()$

$controllerMsg? : MESSAGE$

[send the controllerMsg]

generateMessage

$\Delta(controllerMsg)$

$$\begin{aligned} controllerMsg' = \langle \$ \rangle ^\frown \langle i \rangle ^\frown \langle setpoints.thruster1 \rangle ^\frown \\ \langle j \rangle ^\frown \langle setpoints.thruster2 \rangle ^\frown \\ \langle k \rangle ^\frown \langle setpoints.thruster3 \rangle ^\frown \\ \langle l \rangle ^\frown \langle setpoints.thruster4 \rangle ^\frown \\ \langle m \rangle ^\frown \langle setpoints.thruster5 \rangle ^\frown \\ \langle n \rangle ^\frown \langle setpoints.thruster6 \rangle ^\frown \\ \langle o \rangle ^\frown \langle setpoints.thruster7 \rangle ^\frown \\ \langle p \rangle ^\frown \langle setpoints.thruster8 \rangle ^\frown \langle \$ \rangle \end{aligned}$$

5.4.6 Move

Este é o comportamento responsável pela movimentação do veículo, uma vez que ele passa para o *Actuator* suas referências de velocidade. Ele leva em conta as informações recebidas do processo *AvoidCollision*, que envia um indicador relacionado à presença de obstáculos, e também as referências vindas da camada deliberativa.

Move

Behaviour that make the vehicle to move according to the commands received from the Deliberative Layer and AvoidCollision.

$\texttt{chan}\ s1_move : [s1Data : ControlData]$
$\texttt{chan}\ avoid_move : [avoidCollData : STATUS]$
$\texttt{chan}\ move_s2 : [moveData : ControlData]$
$\texttt{method}\ getNextMovement$

$\texttt{main} = s1_move \rightarrow avoid_move \rightarrow getNextMovement \rightarrow move_s2 \rightarrow \texttt{main}$

$dangerLevel : STATUS$ [indicates the presence of obstacles]
$speedData : ControlData$ [command from the Deliberative Layer]
$avoidData : ControlData$ [last command used in the vehicle]
$movement : ControlData$ [final command to be applied]

INIT
$speedData.INIT$
$avoidData.INIT$
$movement.INIT$

$\texttt{com_}s1_move$
$\Delta(speedData)$
$s1Data? : ControlData$

$speedData' = s1Data?$

$\texttt{com_}avoid_move$
$\Delta(avoidData)$
$avoidCollData? : STATUS$

$dangerLevel' = avoidCollData?$

$\texttt{com_}move_s2$
$\Delta()$
$moveData! : ControlData$

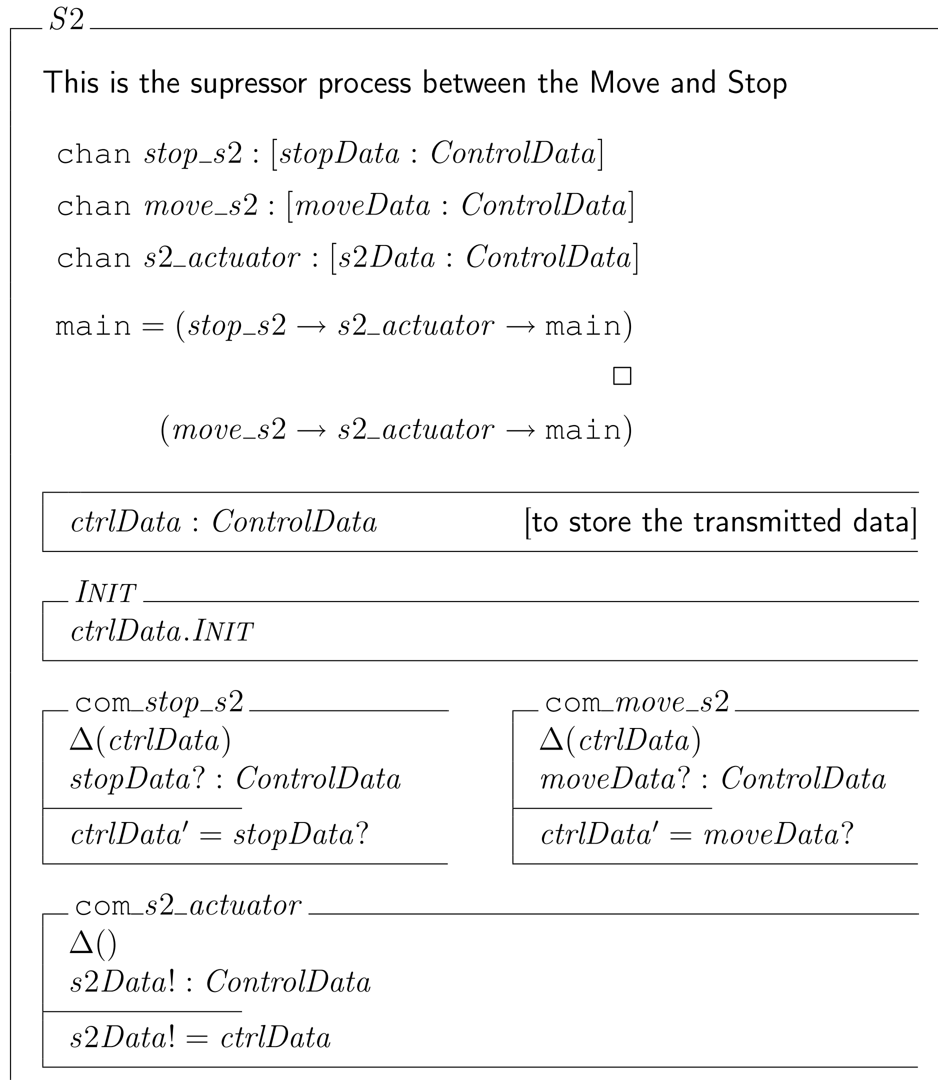
$moveData! = movement$

getNextMovement
$\Delta(movement, avoidData)$
---
if $(dangerLevel = normal)$
then $movement' = speedData$
$avoidData' = -speedData$
else $movement' = avoidData$

A saída deste processo irá depender da entrada vinda do processo *AvoidCollision*. A última informação de velocidade enviada ao atuador quando não foram detectados obstáculos é armazenada com sinal invertido na variável *avoidData*. Assim, esta é utilizada para freiar e retroceder o veículo na iminência de colisões, quando o processo *AvoidCollision* envia um sinal de alerta.

5.4.7 S2

Este processo atua de maneira similar ao processo $S1$, porém com as entradas do processos *Move* e *Stop*.



$S2$

This is the supressor process between the Move and Stop

chan $stop_s2 : [stopData : ControlData]$
chan $move_s2 : [moveData : ControlData]$
chan $s2_actuator : [s2Data : ControlData]$

$$\mathtt{main} = (stop_s2 \rightarrow s2_actuator \rightarrow \mathtt{main}) \;\square\; (move_s2 \rightarrow s2_actuator \rightarrow \mathtt{main})$$

$ctrlData : ControlData$ [to store the transmitted data]

$INIT$
$ctrlData.INIT$

com_$stop_s2$
$\Delta(ctrlData)$
$stopData? : ControlData$
$ctrlData' = stopData?$

com_$move_s2$
$\Delta(ctrlData)$
$moveData? : ControlData$
$ctrlData' = moveData?$

com_$s2_actuator$
$\Delta()$
$s2Data! : ControlData$
$s2Data! = ctrlData$

5.4.8 Stop

O processo *Stop* atua como um comportamento de segurança, que visa proteger o veículo no caso de perda de comunicação com a estação base. Para tanto, ele checa periodicamente quando foi recebida a última mensagem, e caso tenha se passado um intervalo de tempo maior que um limite tolerável, representado pela variável *TIMEOUT*, o processo envia um comando para parar o veículo. Este comando consiste em referências de velocidade nulas, armazenadas no atributo *stop*.

Stop

Behaviour that stop the vehicle if there is no message reception.

inherit *Periodic* [this is a periodic process]

chan $stop_s2 : [stopData : ControlData]$

method *checkLastReception*

$$\texttt{main} = \texttt{Delay until}(nextExecution) \rightarrow checkLastReception \rightarrow (\textbf{if}(Clock - lastReception > TIMEOUT) \textbf{ then } stop_s2) \rightarrow getNextExecution \rightarrow \texttt{main}$$

$TIMEOUT : \mathbb{N}$ [maximum delay between received messages]

$TIMEOUT = 1000$ [in miliseconds]

$lastReception : \mathbb{N}$ [time of the last received message]
$stop : ControlData$ [data with null speeds]

$lastReception > 0$

INIT

$lastReception = Clock$
$stop.INIT$ [initializes with null speeds]

checkLastReception

$\Delta(lastReception)$

[get the time of the last message reception]

com_*stop_s2*

$\Delta()$
$stopData! : ControlData$

$stopData! = stop$

5.4.9 Sensor

O *Sensor* é o responsável por enviar periodicamente as informações armazenadas no *BlackBoard* para a estação base. Assim, as informações são retiradas através do método *getSensors*, transformadas em mensagem no método *structureToMessage* e então enviadas ao *BaseWriter*.

Sensor

Joint all sensor informations and pass it to the Base Station.

inherit *Periodic* [this is a periodic process]

chan $sensor_baseWriter : [sensrData : MESSAGE]$

method *structureToMessage*

method *getSensors*

$$\texttt{main} = \texttt{Delay until}(nextExecution) \rightarrow getSensors \rightarrow structureToMessage \rightarrow sensor_baseWriter \rightarrow getNextExecution \rightarrow \texttt{main}$$

$sensors : SensorData$ [to store the sensor readings]
$blckBoard : BlackBoard$ [reference to the BlackBoard]

INIT

$sensors.INIT$

com_*sensor_baseWriter*

$\Delta()$
$sensrData! : MESSAGE$

$sensrData! = structureToMessage()$

getSensors

$\Delta(sensors)$
$sensors' = blckBoard.getSensorReadings$

structureToMessage

$\Delta()$
[the sonar data is not sent to the Base Station]
$message! : MESSAGE$

[joint the sensor values with their respective id's]
$message! = \langle \$ \rangle \frown \langle d \rangle \frown sensors.direction \frown \langle h \rangle \frown sensors.depth \frown \langle \$ \rangle$

5.4.10 AvoidCollision

O processo *AvoidCollision* é o responsável por tratar as informações dos sonares, identificando a presença de obstáculos próximos ao robô. Caso haja algum obstáculo a uma distância inferior ao limite representado pela variável *LIMIT*, o processo envia um sinal de alerta. Caso contrário, indica condição normal de operação.

AvoidCollision

Identifies obstacles with the sonar readings

inherit *Periodic* [this is a periodic process]

chan *avoid_move* : [*avoidCollisionData* : *ControlData*]
method *generateAvoid*

$$\texttt{main} = \texttt{Delay until}(\mathit{nextExecution}) \rightarrow \mathit{generateAvoid} \rightarrow \mathit{avoid_move} \rightarrow \mathit{getNextExecution} \rightarrow \texttt{main}$$

$\mathit{avoidData} : \mathit{STATUS}$ [to indicate the status of the robot]
$\mathit{sonars} : \mathit{SONAR}$ [readings of the sonars]

INIT

$\mathit{avoidData}.\mathit{INIT}$
$\forall\, i \in [0 \ldots 3] \bullet \mathit{sonarData}_i = 0$

generateAvoid

$\Delta(\mathit{avoidData})$

$\forall\, i \in [0 \ldots 3] \bullet \textbf{if } (\mathit{sonars}_i > \mathit{LIMIT}) \textbf{ then } \mathit{avoidData}' = \mathit{alert}$ $\textbf{else } \mathit{avoidData}' = \mathit{normal}$

com_*avoid_move*

$\Delta()$
$\mathit{avoidCollisionData}! : \mathit{ControlData}$

$\mathit{avoidCollisionData}! = \mathit{avoidData}$

5.4.11 Actuator

Por fim, temos o processo *Actuator*, responsável por transformar as referências de velocidade do veículo recebidos do supressor *S2* em referências de torque para os atuadores. Neste trabalho essa transformação não está levando em conta a planta do veículo. Para isso, deve ser utilizada uma matriz de alocação de empuxo, que permite transformar as referências de velocidade recebidas em referências de torque a ser empregados em cada um dos propulsores do veículo.

Como o foco do trabalho consiste nos aspectos de verificação do modelo e sua implementação, será feita uma implementação simbólica do método *generateSetpoints*, onde este irá sempre gerar valores constantes para as referências de torque nos propulsores. Entretanto, em implementações futuras, a matriz de alocação de empuxo do veículo deverá ser implementada neste método, juntamente com os demais parâmetros de controle a serem adotados.

Actuator

Generate the setpoints to the thrusters.

$\mathtt{chan}\ s2_actuator : [s2Data : ControlData]$

$\mathtt{chan}\ actuator_thruster : [actuatorData : ActuationData]$

$\mathtt{chan}\ generateSetpoints$

$$\mathtt{main} = s2_actuator \rightarrow generateSetpoints \rightarrow actuator_thruster \rightarrow \mathtt{main}$$

$ctrlData : ControlData$ [control data to transform]
$setpoints : ActuationData$ [setpoints to the thrusters]

INIT

$ctrlData.\mathrm{INIT}$
$setpoints.\mathrm{INIT}$

com_s2_actuator

$\Delta(ctrlData)$
$s2Data? : ControlData$

$ctrlData' = s2Data?$

com_actuator_thruster

$\Delta()$
$actuatorData! : ActuationData$

$actuatorData! = setpoints$

generateSetpoints

$\Delta(setpoints)$

[generate the setpoints of the thrusters (in a simplified way)]
$setpoints'.thruster1 = 10.0$
$setpoints'.thruster2 = 20.0$
$setpoints'.thruster3 = 30.0$
$setpoints'.thruster4 = 40.0$
$setpoints'.thruster5 = 50.0$
$setpoints'.thruster6 = 60.0$
$setpoints'.thruster7 = 70.0$
$setpoints'.thruster8 = 80.0$

5.5 Conclusões do Capítulo

Neste capítulo foi apresentada a especificação completa da arquitetura de controle de um veículo submarino na linguagem CSP-OZ. Através desta torna-se possível especificar tanto a parte estática do sistema (por meio da parte Object-Z) quanto o seu comportamento dinâmico (parte CSP). Entretanto, a principal vantagem da utilização desta linguagem consiste na possibilidade de verificação do modelo com a ferramenta FDR, eliminando assim eventuais *deadlocks*, *livelocks* ou construções não-determinísticas do modelo em desenvolvimento.

A arquitetura foi desenvolvida de acordo com o paradigma híbrido, possuindo

assim uma camada reativa e uma deliberativa. Deste modo, funções menos críticas e que demandam mais tempo de processamento como planejamento de trajetórias podem ser separadas das funções mais críticas, como leitura de sensores e detecção de obstáculos. Assim, tarefas com frequências menores não interferem diretamente em tarefas que devem executar em frequências maiores.

A utilização de CSP resultou na decomposição do sistema em diversos processos que executam em paralelo, se comunicando sincronamente por meio de canais unidirecionais. Essa divisão implica numa maior modularização do sistema, facilitando assim sua implementação. Isso porque os diversos componentes podem ser implementados separadamente, e por existirem diversos processos "pequenos", estes possuem uma implementação mais simples. Esta modularidade também possibilita que futuras expansões na arquitetura, como por exemplo a implementação do modo de operação autônomo, sejam feitas sem grandes dificuldades.

6 Resultados

O objetivo deste trabalho consiste na elaboração de um método de desenvolvimento robusto de software embarcado para veículos submarinos. Tal desenvolvimento robusto inclui a utilização de técnicas de modelagem e verificação tanto de modelos quanto da própria implementação em código, a fim de se obter a corretude do sistema por construção. Tal objetivo foi alcançado, uma vez que foi criado um método de desenvolvimento com base na utilização da linguagem formal CSP-OZ, em conjunto com a ferramenta de checagem de modelos FDR e a linguagem de programação RavenSPARK, que possui ferramentas para verificação da implementação em código.

O método proposto foi aplicado em dois sistemas de diferentes graus de complexidade: primeiramente em um modelo simplificado do tipo Produtor / Consumidor, a fim de se estudar a aplicação do método e aprimorar suas regras; e por fim em uma aplicação real, no desenvolvimento de uma arquitetura de controle para um veículo submarino do tipo ROV, que será embarcada no robô apresentado na seção 4.3.1. Destas duas aplicações, foram apresentados apenas os resultados do modelo Produtor / Consumidor. Os resultados da aplicação do método no desenvolvimento da arquitetura de controle serão apresentados a seguir.

6.1 Verificação do Modelo

O script $\mathrm{CSP_M}$ mostrado no anexo A foi analisado no FDR com o objetivo de se encontrar *deadlocks*, *livelocks* e verificar se o sistema é determinístico. Estes testes podem ser feitos tanto no sistema como um todo, representado pelo processo *ROV_Control_Module* como também em cada um dos processos individuais que o compõem.

Como pode ser visto na figura 6.1, o processo *ROV_Control_Module* foi testado com relação à existência de *deadlocks*, *livelocks* e determinismo, sendo bem sucedido em todos eles segundo representado pelas marcações em verde ao lado do nome do processo. Com relação ao desempenho, o FDR realizou as três análi-

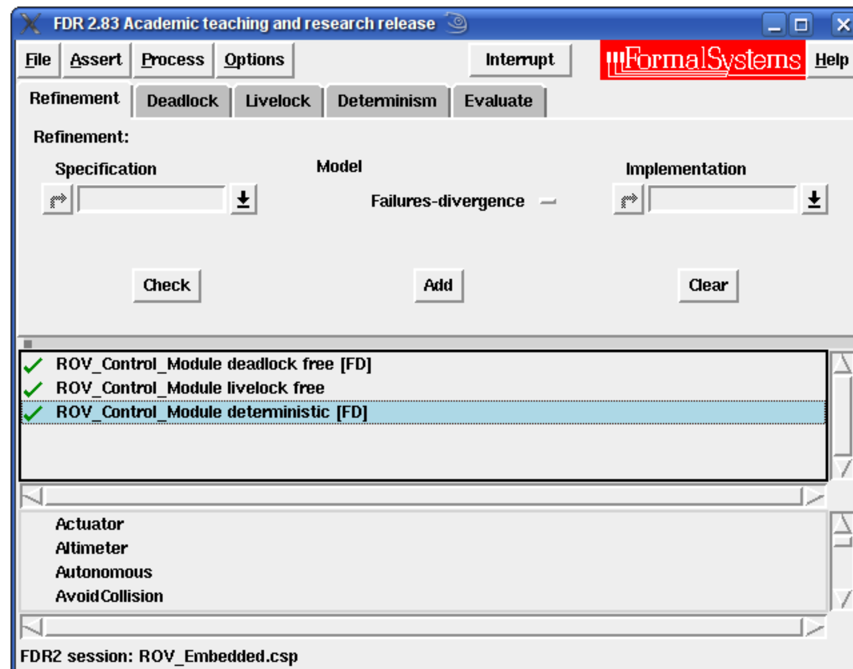


Figura 6.1: Análise da Arquitetura no FDR.

Tabela 6.1: Tempo de Análise da Arquitetura no FDR.

Análise Efetuada	**Tempo Gasto [s]**
deadlock	82
livelock	82
determinismo	3301

ses programadas sem grandes problemas. Segundo o relatório obtido pelo FDR, para o sistema analisado temos as seguintes informações:

- Número de Estados: 2.519.424
- Número de Transições: 23.549.616

Pode-se observar que o número de estados do sistema é bastante elevado, o que torna impossível uma análise manual de sua máquina de estados. Mesmo com esse número elevado de estados, conforme pode ser observado na tabela 6.1, o tempo gasto nas análises é relativamente curto. A análise de determinismo do sistema é a mais longa, levando pouco menos de uma hora. Já as demais levam menos de dois minutos para serem concluídas[1].

[1]Estas análises foram feitas em um notebook comum, com processador AMD Semprom de 1.8 GHz com memória de 1 GB. Assim, pode-se concluir que não é preciso grandes recursos de hardware para executar as análises propostas no método de desenvolvimento.

6.2 Verificação da Implementação

Todo o código desenvolvido em RavenSPARK foi verificado utilizando-se o *Examiner* do SPARK. Assim, foi checada a consistência do código com relação ao fluxo de dados no programa, uso das variáveis, *overflows*, entre outros. Pode-se observar pelo relatório abaixo, fornecido automaticamente pela ferramenta, que a implementação encontra-se livre destes erros.

Código Fonte 6.1: Relatório da Análise da Implementação da Arquitetura de Controle.

```
           *******************************************************
                          Report of SPARK Examination
         SPARK95 Examiner with VC and RTC Generator Release 7.6 / 06.08
                             Demonstration Version
           *******************************************************


                       DATE : 23-APR-2009 11:38:28.55

Options:
    default switch file used
    index_file=rov_index.idx
    warning_file=rov_warning.wrn
    notarget_compiler_data
    config_file=rov_config.cfg
    source_extension=ada
    listing_extension=lst
    nodictionary
    report_file=RELATORIO.rep
    html
    exp_checks
    rtc
    vcs
    nest
    nostatistics
    fdl_identifiers
    flow_analysis=information
    ada95
    annotation_character=#
    profile=ravenscar
    rules=none
    error_explanations=off
    justification_option=full

Selected files:
   Main_ROV_Embedded.adb


Index Filename(s) used were:
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\rov_index.idx


No Meta Files used


Summary warning reporting selected for:
   Hidden parts
   Private types lacking method of initialization
   Notes
   Pragmas: Priority, Inline, Profile


Target configuration file:
Line
   1  -- 32-bit System (Page 196 of the Spark Book)
   2
   3  package Standard is
   4
   5     type Short_Short_Integer is range -2**7 .. 2**7-1;
   6     type Short_Integer is range -2**15 .. 2**15-1;
   7     type Integer is range -2**31 .. 2**31-1;
   8     type Long_Integer is range -2**31 .. 2**31-1;
   9     type Long_Long_Integer is range -2**63 .. 2**63-1;
  10     type Float is digits 6 range -3.40282E+38 .. 3.40282E+38;
  11
  12  end Standard;
  13
  14  package System is
  15
  16     --  System-Dependent Named Numbers
  17
  18     Min_Int                : constant := -2**63;
  19     Max_Int                : constant :=  2**63-1;
  20
  21     Max_Binary_Modulus     : constant := 2**64;
  22     Max_Mantissa           : constant := 63;
  23
  24     --  Storage-related Declarations
  25
  26     type Address is private;
  27
  28     Storage_Unit : constant := 8;
  29     Word_Size    : constant := 32;
  30
  31     --  Priority-related Declarations (RM D.1)
  32
  33     subtype Any_Priority       is Integer      range  0 .. 31;
  34     subtype Priority           is Any_Priority range  0 .. 30;
  35     subtype Interrupt_Priority is Any_Priority range 31 .. 31;
```

```
  36
  37  end System;
2 summarized warning(s), comprising:
     2 note(s)



Source Filename(s) used were:
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Main_ROV_Embedded.adb
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\StatusChannel.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\ActuationDataChannel.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\ThrusterWriter.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Stop.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Actuator.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\AvoidCollision.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Move.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\S2.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\S1.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\BaseWriter.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Sensor.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Sonar.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Altimeter.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Compass.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\BlackBoard.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\ControlDataChannel.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\MsgChannel.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Remote.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\BaseReader.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\UDP_Sockets.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Constants.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\DataTypes.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Shadows.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Decoder.ads
   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Encoder.ads



Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\StatusChannel.ads
No Listing File

   Unit name:  StatusChannel
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\
     ActuationDataChannel.ads
No Listing File

   Unit name:  ActuationDataChannel
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\ThrusterWriter.ads
No Listing File

   Unit name:  ThrusterWriter
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Stop.ads
No Listing File

   Unit name:  Stop
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Actuator.ads
No Listing File

   Unit name:  Actuator
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\AvoidCollision.ads
No Listing File

   Unit name:  AvoidCollision
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Move.ads
No Listing File

   Unit name:  Move
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.
```

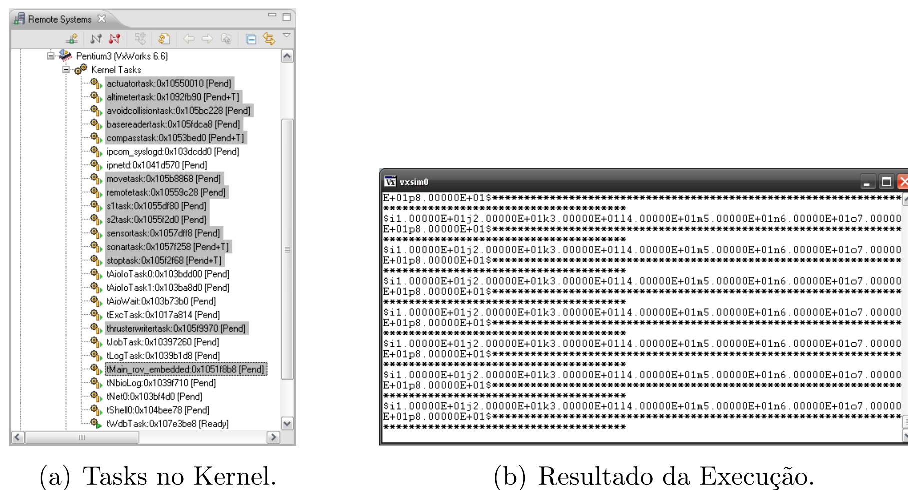
```
No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\S2.ads
No Listing File

   Unit name:  S2
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\S1.ads
No Listing File

   Unit name:  S1
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\BaseWriter.ads
No Listing File

   Unit name:  BaseWriter
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Sensor.ads
No Listing File

   Unit name:  Sensor
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Sonar.ads
No Listing File

   Unit name:  Sonar
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Altimeter.ads
No Listing File

   Unit name:  Altimeter
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Compass.ads
No Listing File

   Unit name:  Compass
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\BlackBoard.ads
No Listing File

   Unit name:  BlackBoard
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\ControlDataChannel.
    ads
No Listing File

   Unit name:  ControlDataChannel
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\MsgChannel.ads
No Listing File
```

```
   Unit name:  MsgChannel
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Remote.ads
No Listing File

   Unit name:  Remote
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\BaseReader.ads
No Listing File

   Unit name:  BaseReader
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\UDP_Sockets.ads
No Listing File

   Unit name:  UDP_Sockets
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

2 summarized warning(s), comprising:
     1 hidden part(s)*
     1 private type(s) lacking method of initialization
(*Note: the above warnings may affect the validity of the analysis.)


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Constants.ads
No Listing File

   Unit name:  Constants
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\DataTypes.ads
No Listing File

   Unit name:  DataTypes
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Shadows.ads
No Listing File

   Unit name:  GNAT
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

   Unit name:  Ada.Streams
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

   Unit name:  GNAT.Sockets
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Decoder.ads
No Listing File

   Unit name:  Decoder
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings


Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Encoder.ads
No Listing File

   Unit name:  Encoder
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

No errors found

No summarized warnings
```

```
Source Filename:   C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\Main_ROV_Embedded.
    adb
Listing Filename:  C:\Documents and Settings\fhassis\Desktop\ROV_Embedded\SPARK\
    Main_ROV_Embedded.lst

   Unit name:
   Unit type:  main program
   Unit has been analysed, any errors are listed below.

No errors found

1 summarized warning(s), comprising:
     1 pragma(s)*
(*Note: the above warnings may affect the validity of the analysis.)

--End of file--------------------------------------------------
```

6.3 Testes no Sistema Embarcado

O sistema foi testado em bancada utilizando o ambiente de desenvolvimento do VxWorks, o Workbench. Ele foi embarcado em um PC104 semelhante ao existente no ROV apresentado na seção 4.3.1, a fim de se testar a troca de mensagens com a estação base.



(a) Tasks no Kernel. (b) Resultado da Execução.

Figura 6.2: Execução da Arquitetura de Controle no VxWorks.

A figura 6.2(a) mostra os processos sendo executados no kernel embarcado no PC104. Como a rede CAN ainda não foi implementada no robô, o processo *ThrusterWriter* não pode enviar as mensagens via rede para o controlador dos propulsores. Assim, as mensagens que seriam enviadas são impressas na tela a fim de se mostrar o funcionamento do sistema, como mostrado na figura 6.2(b).

Com o sistema embarcado, podem ser feitos diversos testes como análises de memória, consumo de processamento, tempos de resposta das diversas tarefas do sistema, entre outros. Como tais análises são bastante complexas e específicas, pois dependem de uma estratégia pré estabelecida para a realização das mesmas de acordo com o que se deseja observar do comportamento do sistema, estas não serão feitas neste trabalho. Foram feitos apenas testes do tipo *blackbox* no

sistema, observando suas saídas a partir de determinadas entradas conhecidas. Para estes casos, o sistema se comportou da forma esperada.

7 Conclusões

O desenvolvimento de software com base no uso de especificações formais é um meio de se buscar a corretude dos programas por construção, conceito difícil de ser aplicado por meio das metodologias de projeto que não utilizam linguagens formais devido a dificuldades na verificação dos modelos desenvolvidos. Como solução para especificação formal do sistema foi utilizado o CSP-OZ, que permite modelar e analisar o comportamento dinâmico do sistema através da especificação dos processos que o compõem e de seus modos de interação, além de permitir a especificação das estruturas de dados internas dos processos incluindo definições de tipos básicos de dados e suas condições de existência.

Com base nas especificações em CSP-OZ é possível checar tanto o modelo desenvolvido quanto sua implementação em código. A checagem do modelo é feita pelo checador de modelos FDR através de um *script* na linguagem $\mathrm{CSP_M}$, feito com base na parte CSP da especificação em CSP-OZ. Já a checagem da implementação é feita pelo examinador da linguagem SPARK com base em anotações inseridas no código na forma de comentários. Grande parte destas anotações são geradas a partir das especificações das estruturas de dados da parte Object-Z dos processos. Neste trabalho não foram criadas anotações relacionadas a pré e pós condições da estrutura de dados, e portanto não foram realizadas provas formais com o examinador. Este tipo de abordagem deverá ser usada em trabalhos futuros.

A utilização de uma linguagem formal como CSP-OZ facilita a checagem do modelo desenvolvido e de sua implementação, uma vez que tanto a geração de *scripts* $\mathrm{CSP_M}$ quanto a elaboração das anotações SPARK relacionadas às pré e pós condições das variáveis e métodos, além da especificação dos intervalos de valores válidos para os tipos de dados utilizados na implementação em RavenSPARK são todos feitos com base no modelo em CSP-OZ. Isso implica na diminuição do tempo gasto na elaboração tanto dos *scripts* quanto das anotações. Outro aspecto importante é que as especificações formais por si só representam uma boa documentação do software, não sendo necessário ficar atualizando a documentação à

medida que o modelo vai sendo modificado, pois um representa o outro.

A implementação do modelo utilizando-se subconjuntos restritos da linguagem Ada empregados no desenvolvimento de softwares críticos como o Ravenscar e o SPARK resulta em uma implementação mais robusta, confiável e determinística, uma vez que o código gerado segundo estes padrões está livre de uma série de características da linguagem Ada como ponteiros, alocação dinâmica e diversos outros recursos relacionados à programação concorrente e que dificultam seu projeto e análise. No caso do SPARK, a geração de anotações e a verificação do código está totalmente de acordo com os objetivos do trabalho, que é buscar a corretude do software por construção, dando grande ênfase no desenvolvimento preliminar do sistema, ou seja, em sua modelagem.

Outra grande vantagem da utilização do RavenSPARK consiste na facilidade na transformação dos modelos em CSP-OZ para código, uma vez que a linguagem de programação adotada possui os recursos necessários para uma implementação direta e segura, como *tasks*, objetos protegidos, mecanismos de sincronização de *tasks*, encapsulamento em *packages*, entre outros. Isso possibilitou a elaboração das regras de criação de canais e processos utilizadas no método de desenvolvimento proposto. A criação de canais foi feita com base nos trabalhos de Atiya (2004), Atiya e King (2005), onde foram feitas provas formais em CSP sobre a corretude da implementação dos canais em Ada utilizando o padrão Ravenscar.

Com o modelo e o código checados, o sistema pode ser embarcado e analisado em tempo de execução por meio das ferramentas de desenvolvimento do VxWorks, podendo assim serem realizados os testes e diversas análises de desempenho relacionadas ao consumo de processamento e de memória. Depois de feitas todas as checagens na arquitetura desenvolvida, nesta etapa não foram encontrados erros. Entretanto, isso não garante que a implementação esteja livre de erros, nem que o método desenvolvido leve a uma implementação livre dos mesmos. O que sempre é possível verificar é se o sistema apresenta falhas para um determinado conjunto de testes.

Com relação à técnica de checagem de modelos utilizada neste trabalho, observou-se que esta apresentou um desempenho satisfatório, pois todas as análises realizadas com o FDR foram finalizadas em um tempo relativamente curto. Isso porque não foram analisadas as estruturas de dados internas dos processos no FDR, sendo estas feitas com o examinador do SPARK, que também efetuou todas as análises programadas em um tempo bastante curto.

Referências

AKHLAKI, K. B.; TUNON, M. I. C.; TERRIZA, J. A. H.; MORALES, L. E. M. Formal Specification of Real-Time Systems by Transformation of UML-RT Design Models. In: BARJIS, J.; ULTES-NITSCHE, U.; AUGUSTO, J. C. (Ed.). *MSVVEIS*. [S.l.]: INSTICC Press, 2006. p. 16–25. ISBN 978-972-8865-49-8.

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

AMEY, P.; DOBBING, B. High Integrity Ravenscar. In: *Ada-Europe*. [S.l.: s.n.], 2003. p. 68–79.

AMIANTI, G. *Arquitetura de Software Aviônico com Requisitos de Homologação*. Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo, 2008.

ATIYA, D.-A.; KING, S. Extending Ravenscar with CSP Channels. In: *Ada-Europe*. [S.l.: s.n.], 2005. p. 79–90.

ATIYA, D. M. *Verification of Concurrent Safety-critical Systems: The Compliance Notation Approach*. Tese (Doutorado) — University of York, October 2004.

BARNES, J. *High Integrity Software - The SPARK Approach to Safety and Security*. [S.l.]: Addison-Wesley, 2006.

BARNES, J. G. *Programming in Ada*. 3. ed. Wokingham [u.a.]: Addison-Wesley, 1989. ISBN 0-201-17566-5.

BELAPURKAR, A. *CSP for Java Programmers*. [S.l.], Jun 2005. Disponível em: <www.ibm.com/developerworks/java/library/j-csp1.html>.

BORGES, R. M.; MOTA, A. C. Integrating UML and Formal Methods. *Electron. Notes Theor. Comput. Sci.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 184, p. 97–112, 2007. ISSN 1571-0661.

BOWEN, J. P.; HINCHEY, M. G. Ten Commandments Revisited: A Ten-Year Perspective on the Industrial Application of Formal Methods. In: *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*. New York, NY, USA: ACM Press, 2005. p. 8–16. ISBN 1-59593-148-1.

BRAT, G.; GIANNAKOPOULOU, D.; GOLDBERG, A.; HAVELUND, K.; LOWRY, M.; PASAREANU, C.; VENET, A.; VISSER, W. *Experimental Evaluation of Verification and Validation Tools on Martian Rover Software*. 2003. Disponível em: <citeseer.ist.psu.edu/brat03experimental.html>.

BROOKS, F. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer Magazine*, v. 4, n. 4, p. 10 – 19, abr. 1987.

BROOKS, R. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, v. 2, n. 1, p. 14–23, 1986. ISSN 0882-4967.

BURNS, A. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. 3rd. ed. [S.l.]: Addison-Wesley, 2001. 738 p.

BURNS, A.; DOBBING, B.; VARDANEGA, T. Guide for the use of the Ada Ravenscar Profile in High Integrity Systems. *Ada Lett.*, ACM, New York, NY, USA, XXIV, n. 2, p. 1–74, 2004. ISSN 1094-3641.

CHAMPEAU, J.; DHAUSSY, P.; MOITIE, R.; PRIGENT, A. Object Oriented and Formal Methods for AUV development. *OCEANS 2000 MTS/IEEE Conference and Exhibition*, v. 1, p. 73–78 vol.1, 2000.

DUKE, R.; ROSE, G. *Formal object-oriented specification using object-z*. [S.l.]: Macmillan, 2000. 229 p. (cornestones of computing).

FISCHER, C. 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. p. 423–438. ISBN 0-412-82040-4.

FISCHER, C. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. Tese (Doutorado) — Universidade de Oldenburg, january 2000.

FISCHER, C.; OLDEROG, E.-R.; WEHRHEIM, H. A CSP View on UML-RT Structure Diagrams. In: *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*. London, UK: Springer-Verlag, 2001. p. 91–108. ISBN 3-540-41863-6.

FISCHER, C.; WEHRHEIM, H. 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. p. 315–334. ISBN 1-85233-107-0.

FISCHER, C.; WEHRHEIM, H. Failure-divergence Semantics as a Formal Basis for an Object-Oriented Integrated Formal Method. *Bulletin of the European Association for Theoretical Computer Science*, v. 71, p. 92–, 2000. Disponível em: <citeseer.ist.psu.edu/452845.html>.

GOLDSACK, S. J. *Ada for Specification: Possibilities and Limitations*. New York, NY, USA: Cambridge University Press, 1985. ISBN 0521308534.

HALL, A.; CHAPMAN, R. Correctness by Construction: Developing a Commercial Secure System. *Software, IEEE*, v. 19, n. 1, p. 18–25, 2002. ISSN 0740-7459.

JOVANOVIC, D. S.; ORLIC, B.; LIET, G. K.; BROENINK, J. F. gCSP: A Graphical Tool for Designing CSP Systems. *Communicating Process Architectures 2004*, 2004. Disponível em: <http://www.djov.net/DT/JovanovicCPA2004.pdf>.

KASSEL, G.; SMITH, G. Model Checking Object-Z Classes: Some experiments with FDR. In: *Proc. Eighth Asia-Pacific Software Engineering Conference APSEC 2001*. [S.l.: s.n.], 2001. p. 445–452.

LAWRENCE, J. Practical Application of CSP and FDR to Software Design. In: . [s.n.], 2005. p. 151–174. Disponível em: <http://dx.doi.org/10.1007/11423348_9>.

LIGHTFOOT, D. *Formal Specification using Z*. [S.l.]: Macmillan, 1991. ISBN 0-333-54408-0.

MEDEIROS, A. de; CHATILA, R.; FLEURY, S. 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*, v. 1, p. 162–169 vol.1, Nov 1996.

MOBILE. *Home page da empresa Mobile Robots, desenvolvedora de robôs terrestres*. 2009.

MOLLER, M.; OLDEROG, E.-R.; RASCH, H.; WEHRHEIM, H. Integrating a Formal Method into a Software Engineering Process with UML and Java. *Form. Asp. Comput.*, Springer-Verlag, London, UK, v. 20, n. 2, p. 161–204, 2008. ISSN 0934-5043.

MOTA, A.; FARIAS, A.; SAMPAIO, A. De CSPz para CSPm: Uma ferramenta transformacional Java. In: Workshop de Métodos Formais, p. 1–10, 2001.

MURPHY, R. R. *Introduction to AI Robotics*. Cambridge, MA, USA: MIT Press, 2000. ISBN 0262133830.

NG, M. Y.; BUTLER, M. Tool Support for Visualizing CSP in UML. In: GEORGE, C.; MIAO, H. (Ed.). *International Conference on Formal Engineering Methods(ICFEM)*. Springer Verlag, 2002. p. 287–298. Disponível em: <http://eprints.ecs.soton.ac.uk/6908/>.

OMG. *OMG UML 2.0 OCL specification*. October 2003. Disponível em: <http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf>.

OMG. *Unified Modeling Language: Superstructure. Version 2.0*. 2004.

POLIDO, M. F. *Um método de refinamento para desenvolvimento de software embarcado: Uma abordagem baseada em UML-RT e especificações formais*. Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo, 2007.

ROSCOE, A. W.; HOARE, C. A. R.; BIRD, R. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN 0136744095.

RTCA. *DO-178B - Software Consideration in Airborne Systems and Equipment Certification*. December 1992.

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

SAAB. *Home Page da empresa Saab Seaeye, desenvolvedora de veículos submarinos*. 2009. Disponível em: <www.seaeye.com>.

SCHNEIDER, S. A. *Concurrent and Real Time Systems: The CSP Approach.* New York, NY, USA: John Wiley & Sons, Inc., 1999. ISBN 0471623733.

SCHOEBERL, M. Restrictions of Java for Embedded Real-Time Systems. *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, p. 93–100, 2004.

SHERIF, A.; SAMPAIO, A.; CAVALCANTE, S. An Integrated Approach to Specification and Validation of Real-Time Systems. In: *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity.* London, UK: Springer-Verlag, 2001. p. 278–299. ISBN 3-540-41791-5.

SHERIF, A.; SAMPAIO, A.; CAVALCANTE, S. Specification and Validation of the SACI-1 On-Board Computer Using Timed-CSP-Z and Petri Nets. In: . [s.n.], 2003. p. 161–180. Disponível em: <http://dx.doi.org/10.1007/3-540-44919-1_13>.

SIMPSON, J.; JACOBSEN, C. L.; JADUD, M. C. Mobile Robot Control: The Subsumption Architecture and occam-pi. In: BARNES, F. R. M.; KERRIDGE, J. M.; WELCH, P. H. (Ed.). *Communicating Process Architectures 2006.* IOS Press, 2006. p. 225–236. ISBN 1-58603-671-8. Disponível em: <http://www.transterpreter.org/wiki/Publications>.

SYSTEMS, F. *Failures-Divergence Refinement: FDR2 User Manual.* [S.l.], june 2005.

TEAM, S. *SPARK Examiner - The SPARK Ravenscar Profile.* 1.5. ed. [S.l.], December 2006.

TELELOGIC. *Rhapsody.* 2009. Disponível em: <http://www.telelogic.com/products/rhapsody/index.cfm>.

WELCH, P. H.; MARTIN, J. M. R. Formal Analysis of Concurrent Java Systems. In: P.H.WELCH; A.W.P.BAKKERS (Ed.). *Communicating Process Architectures 2000.* IOS Press (Amsterdam), 2000. (Concurrent Systems Engineering, v. 58), p. 275–301. ISBN 1 58603 077 9. ISSN 1383-7575. Disponível em: <http://www.cs.kent.ac.uk/pubs/2000/1145>.

XMOBOTS. *Home Page da empresa brasileira XMobots, desenvolvedora de Veículos Não-Tripulados.* 2009. Disponível em: <www.xmobots.com.br>.

YEUNG, W.; LEUNG, K.; WANG, J.; DONG, W. Improvements towards formalizing UML state diagrams in CSP. *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, p. 7 pp.–, December 2005. ISSN 1530-1362.

APÊNDICE A – Notação de Object-Z e CSP

A.1 Notação da linguagem Object-Z

x : T	Definição de variável x do tipo T
x_1 : T_1; ...; x_n : $\mathrm{T_n}$	Lista de declarações
x_1, x_n : $\mathrm{T_n}$	Declaração de variáveis do mesmo tipo T
$[X_1, ..., X_n]$	Definição de tipos
$X ::= x_1 \mid \ldots \mid x_n$	Definição do tipo X como uma seqüência de elementos
$true, false$	Constantes lógicas
$\mathbb{B}$	Conjunto das constantes lógicas
$\neg P$	Negação lógica
$P \wedge Q$	Conjunção lógica
$P \vee Q$	Disjunção lógica
$P \Rightarrow Q$	Implicação lógica
$P \Leftrightarrow Q$	Equivalência lógica
$\forall D \bullet P$	Quantificação universal
$\exists D \bullet P$	Quantificação existencial
$t_1 = t_2$	Equalização dos termos
$t_1 \neq t_2$	Diferenciação dos termos
$\varnothing$	Conjunto vazio
$t \in S$	Pertence de um conjunto
$t \notin S$	Não pertence ao conjunto
$S_1 \subseteq S_2$	Subconjunto
$\{t_1, ..., t_n\}$	Conjunto de elementos

$\{D \mid P\}$	Expressão de conjunto
$\{D \mid P \bullet t\}$	Abstração de conjunto
$\mathbb{P}S$	Conjunto total *(Power Set)*
$\sharp S$	Tamanho do conjunto
$(t_1, ..., t_n)$	Tuplas ordenadas
$S_1 \times ... \times S_n$	Produto cartesiano
$first(t_1, ..., t_n)$	Seleção de termos
$X \cup Y$	União de conjuntos
$X \cap Y$	Intersecção de conjuntos
$X \setminus Y$	Diferenciação de conjuntos
$\mathbb{R}$	Conjunto dos números reais
$\mathbb{Z}$	Conjunto dos números inteiros
$\mathbb{N}$	Conjunto dos números naturais
$\mathbb{N}_1$	Conjunto dos números estritamente positivos
div	Divisão inteira
$m..n$	Conjunto dos inteiros entre m e n inclusive
$X \leftrightarrow Y$	Conjunto das relações entre X e Y
$x\underline{R}y$	x está relacionado a R através de y
$x \rightarrowtail y$	x mapeia y
$dom\ R$	Domínio da relação
$ran\ R$	Limite da relação
$R_1 \fatsemi R_2$	Composição relacional
$R^\sim$	Inversa relacional
$R(\mid S \mid)$	Imagem do conjunto S através da relação R
$S \lhd R$	Restrição de domínio
$R \rhd T$	Restrição de limites
$f(t)$	Imagem de t sobre a função f
$X \nrightarrow Y$	Função parcial
$X \rightarrow Y$	Função total
$X \rightarrowtail\!\!\!\!\!\!+\ Y$	Função um para um parcial (injetora parcial)
$X \rightarrowtail Y$	Função um para um total (injetora total)
$f \oplus g$	g sobrescreve f
$\langle\rangle$	Seqüência vazia
$seqX$	Conjunto da seqüência finita com elementos do tipo T
seq_1X	Conjunto de seqüência, não vazia, finita com elementos do tipo T
$\langle t_1, ..., t_2\rangle$	Lista de elementos da seqüência

$A \frown B$	Concatenação de seqüências
$head\ A$	Primeiro elemento de uma seqüência não vazia
$tail\ A$	Todos os elementos de uma seqüência não vazia, sem o primeiro elemento
$last\ A$	Elemento final de uma seqüência não vazia
$\upharpoonright (f_1, ..., f_n)$	Lista de visibilidade
$\Delta(v_1, ..., v_2)$	Lista de variáveis a sofrerem mudanças de estado
Δ	Indicador de variáveis a sofrerem mudanças de estado
$\textsc{Init}$	Esquema de inicialização
$v :\downarrow A$	Declaração de polimorfismo
$v : A_{\copyright}$	Composição de objetos
$B \bigcup C$	União de classes
$v?$	Variável de entrada
$v!$	Variável de saída
v'	Valor da variável v após uma determinada operação
$obEx.op$	Aplicação de uma operação
$[v_1/v_2]$	Troca de nomes de variáveis
$[op_1/op_2]$	Troca de nomes de operações
$op \mathrel{\widehat=} opEx$	Declaração de nome de operação
$opEx_1 \wedge opEx_2$	Conjunção de operação
$opEx_1 \parallel opEx_2$	Composição paralela de operação
$opEx_1 \parallel_! opEx_2$	Sincronização de composição paralela de operação através de variáveis de saída
$opEx_1 \Box opEx_2$	Escolha de operações
$opEx_1 \fatsemi opEx_2$	Composição seqüencial de operações
$opEx_1 \bullet opEx_2$	Complemento na declaração de operações
$\wedge D \bullet opEx$	Conjunção de operações distribuídas
$\Box D \bullet opEx$	Escolha de operações distribuídas
$\fatsemi D \bullet opEx$	Composição seqüencial de operações distribuídas
$\mathbb{O}$	Universo dos objetos

A.2 A notação da linguagem CSP

$a \in x$	Membro de um conjunto
$x \subseteq u$	Subconjunto
$\{\}$	Conjunto vazio
$\{a_1, ..., a_2\}$	Conjunto de elementos
$x \cup y, \quad \bigcup X$	União
$x \cap y, \quad \bigcap X \quad (X \neq \{\})$	Intersecção
$x \setminus y$	Diferença
$\mathbb{P}(x)$	Conjunto completo *(Power Set)*
$x \times y$	Produto cartesiano
$x \longrightarrow y$	Mapeamento total de x para y
$\mathbb{N}$	Números naturais
$\mathbb{Z}$	Números inteiros
$\mathbb{R}$	Números reais
$\mathbb{R}^+$	Números reais não negativos
$\oplus, \ominus$	Adição e subtração no modulo em uma determinada base
$x \wedge y$	Conjunção
$x \vee y$	Disjunção
$\neg x$	Negação
$x \Rightarrow y$	Implicação
$x \Leftrightarrow y$	Equivalência
$\forall\, x.\chi$	Quantificação universal
$\exists\, x.\chi$	Quantificação existencial
Σ	Alfabeto de todas as comunicações
$\checkmark$	Sinal de término
τ	Ação invisível
$\Sigma^{\checkmark}$	Alfabeto com o sinal de término
$\Sigma^{\checkmark,\tau}$	Alfabeto com o sinal de término e com a ação invisível
$a.b.c$	Composição de eventos
$c?x$	Canal de entrada
$c!e$	Canal de saída
$\{\mid a, b \mid\}$	Eventos associados com os canais

A^*	Conjunto de todas as seqüências finitas sobre A
$A^{*\checkmark}$	Conjunto de todas as seqüências finitas sobre A união com o elemento sinal de término
A^ω	Conjunto de todas as seqüências infinitas sobre A
$\langle\rangle$	Seqüência vazia
$\langle a_1, ..., a_n\rangle$	Seqüência contendo elementos
$s \frown t$	Concatenação de duas sequências
$s \setminus X$	Esconder *(Hiding)*: Todos os elementos de X serão eliminados de s
$s \upharpoonright X$	Restrição
$\#s$	Tamanho da sequência
$s \downarrow a$	Número de eventos
$s \downarrow c$	Seqüência dos valores comunicados no canal c em s
$s \leq t$	Prefixo
$s \underset{X}{\parallel} t$	Composição paralela generalizada
$s \vert\vert\vert t$	Intercalação *(Interleaving)*
$\overline{S}$	Fechamento de S
$\mu\, \rho.P$	Recursão
$a \rightarrow P$	Prefixação
$?x : A \rightarrow P$	Prefixação com escolha
$(a \rightarrow P \mid b \rightarrow Q)$	Alternativa guardada
$P \Box Q$	Escolha externa
$P \sqcap Q, \quad \sqcap S$	Escolha não determinística
$P \not< b \not> Q$	Escolha condicional
$P \Vert Q$	Composição paralela sincronizada
$P_X \Vert_Y Q$	Composição paralela alfabetizada
$P \underset{X}{\parallel} Q$	Composição paralela generalizada
$P \vert\vert\vert Q$	Intercalação
$P \setminus Q$	Esconder *(Hiding)*
$f[P]$	Troca de nomes (*renaming* - funcional)
$P[R]$	Troca de nomes (*renaming* - relacional)
$P[a/b]$	Troca de nomes (*renaming* - relacional, por substituição)
$a.P$	Nomeação de processo
$P \;;\; Q$	Composição seqüencial
$P \gg Q$	Encadeamento
$P //_X Q$	Encadeamento sincronizado
$P // m : Q$	Encadeamento sincronizado de nome de processos

$P \rhd Q$	Operador de *time-out*
$P \bigtriangleup_a Q$	Interrupção
$P[x/y]$	Substituição
P/s	Operador de após (após execução)
$P \downarrow n$	Restrição para a profundidade de n
$\mathcal{L}_H(P)$	Abstração fraca
$\mathcal{E}_H(P)$	Abstração forte
$\mathcal{M}_H^S(P)$	Abstração combinada
$P \xrightarrow{a} Q$	Transição de ação simples
$P \overset{s}{\Longrightarrow} Q$	Transição de ação múltipla
$P \overset{t}{\longmapsto} Q$	Múltiplas transições de ações
$\tau^*(P)$	Expansão de P
$P \ \mathit{ref} \ B$	P refuta *(refuses)* B
$P \ \mathit{div}$	P diverge
$\mathcal{T}$	Modelo de traço
$\mathcal{N}$	Modelo de falhas/divergências
$\mathcal{F}$	Modelo de falhas estáveis
$\mathcal{I}$	Modelos de traços/divergências infinitos
$\mathcal{U}$	Modelo de falhas/divergências/traços infinitos
$\mathcal{D}$	Conjunto de processos determinísticos
$\mathcal{T}^d$	Possíveis membros determinísticos de T
$\bot_{\mathcal{N}}$	Elemento final do modelo
$\top_{\mathcal{F}}$	Elemento inicial do modelo
$\sqsubseteq_{\mathcal{T}}$	Refinamento por traços
$\sqsubseteq_{\mathcal{FD}}$	Refinamento por falhas/divergências
$\sqsubseteq_{\mathcal{F}}$	Refinamento por falhas
$\sqsubseteq_{\mathcal{I}}$	Refinamento sobre I
$\sqsubseteq_{\mathcal{U}}$	Refinamento sobre U
$\sqsubseteq$	Refinamento sobre qualquer modelo
$P \leq Q$	Ordem forte
$\sqcup X$	Menor limite superior
$\sqcap X$	Maior limite inferior
μf	Menor ponto fixo

ANEXO A – Script CSP_M da Arquitetura de Controle

Código Fonte A.1: Script CSP_M da Arquitetura de Controle

```
-- canais utilizados na arquitetura
channel s2_actuator
channel autonomous_s1
channel stop_s2
channel sensor_baseWriter
channel avoid_move
channel remote_s1
channel baseReader_remote
channel s1_move
channel baseReader_autonomous
channel move_s2
channel actuator_thruster

-- metodos internos dos processos
channel saveReceptionTime
channel getNextExecutionCps          -- Compass
channel getNextExecutionAlt          -- Altimeter
channel getNextExecutionSnr          -- Sonar
channel getNextExecutionSns          -- Sensor
channel getNextExecutionStp          -- Stop
channel getNextExecutionAvd          -- AvoidCollision
channel getMessage, decodeMessage
channel getDirection, saveDirection
channel getDepth, saveDepth
channel getSonar, saveSonar
channel sendMessageBwt               -- BaseWriter
channel sendMessageTwt               -- ThrusterWriter
channel getSensors, structureToMessage
channel generateAvoid, getNextMovement
channel checkLastReception
channel generateSetpoints
channel generateMessage

-- Composicao principal do Sistema
ROV_Control_Module = BaseReader [| {| baseReader_remote, baseReader_autonomous |} |] (Remote
    [| {| remote_s1 |} |] (S1 [| {| autonomous_s1, s1_move |} |] (Autonomous ||| (Move [| {|
    move_s2, avoid_move |} |] (S2 [| {| stop_s2, s2_actuator |} |] (Actuator [| {|
    actuator_thruster |} |] (ThrusterWriter ||| (Stop ||| (AvoidCollision ||| (Sensor [| {|
    sensor_baseWriter |} |] (BaseWriter ||| (Altimeter ||| (Compass ||| Sonar))))))))))))

-- Especificacao dos Processos
Remote = baseReader_remote -> decodeMessage -> remote_s1 -> Remote
Autonomous = SKIP
S1 =     remote_s1 -> s1_move -> S1
      [] autonomous_s1 -> s1_move -> S1
Compass = getDirection -> saveDirection -> getNextExecutionCps -> Compass
Altimeter = getDepth -> saveDepth -> getNextExecutionAlt -> Altimeter
Sonar = getSonar -> saveSonar -> getNextExecutionSnr -> Sonar
Sensor = getSensors -> structureToMessage -> sensor_baseWriter -> getNextExecutionSns ->
    Sensor
AvoidCollision = generateAvoid -> avoid_move -> getNextExecutionAvd -> AvoidCollision
Move = s1_move -> avoid_move -> getNextMovement -> move_s2 -> Move
Stop = checkLastReception -> ((stop_s2 -> getNextExecutionStp -> Stop) [] (
    getNextExecutionStp -> Stop))
Actuator = s2_actuator -> generateSetpoints -> actuator_thruster -> Actuator
ThrusterWriter = actuator_thruster -> generateMessage -> sendMessageTwt -> ThrusterWriter
BaseReader = getMessage -> saveReceptionTime -> baseReader_remote -> BaseReader
BaseWriter = sensor_baseWriter -> sendMessageBwt -> BaseWriter
S2 =     stop_s2 -> s2_actuator -> S2
      [] move_s2 -> s2_actuator -> S2

-- Checagens automaticas no modelo
assert ROV_Control_Module :[deadlock free [FD]]
assert ROV_Control_Module :[livelock free]
assert ROV_Control_Module :[deterministic [FD]]
```

ANEXO B – Códigos Fonte da Arquitetura de Controle

Seguem os códigos fonte dos arquivos utilizados na implementação da Arquitetura de Controle desenvolvida para o ROV utilizado neste trabalho.

Código Fonte B.1: Arquivo DataTypes.ads

```
with Ada.Real_Time;
--# inherit Ada.Real_Time;

package DataTypes is
   ----------------------------------------------------------------
   -- DATA TYPE DECLARATIONS USED IN THE IMPLEMENTATION --
   ----------------------------------------------------------------
   type STATUS is (normal, alert);

   msgSize: constant Integer := 100; -- Maximum message size allowed
   subtype Message_Index is Integer range 1 .. msgSize;
   subtype MESSAGE is String (Message_Index);

   subtype COMPASS_TYPE is Float range 0.0 .. 360.0;

   subtype ALTIMETER_TYPE is Float range 0.0 .. 300.0;

   subtype SPEED_TYPE is Float range -1.0 .. 1.0;

   subtype TORQUE_TYPE is Float range -500.0 .. 500.0;

   subtype Sonar_Index is Integer range 1 .. 4;
   subtype Sonar_Range is Float range 0.0 .. 1.0;
   type SONAR_TYPE is array(Sonar_Index) of Sonar_Range;

   type SensorData is
      record
         direction: COMPASS_TYPE;
         depth: ALTIMETER_TYPE;
         sonar: SONAR_TYPE;
      end record;

   type ControlData is
      record
         vx: SPEED_TYPE;
         vy: SPEED_TYPE;
         vz: SPEED_TYPE;
         vTheta: SPEED_TYPE;
      end record;

   type ActuationData is
      record
         vThruster1: TORQUE_TYPE;
         vThruster2: TORQUE_TYPE;
         vThruster3: TORQUE_TYPE;
         vThruster4: TORQUE_TYPE;
         vThruster5: TORQUE_TYPE;
         vThruster6: TORQUE_TYPE;
         vThruster7: TORQUE_TYPE;
         vThruster8: TORQUE_TYPE;
      end record;

   -- To constrain the type String, used to decode the message
   -- and as an IP address
   subtype Str_Number_Index is Integer range 1 .. 20;
   subtype Str_Number is String(Str_Number_Index);

   subtype IP_Index is Integer range 1 .. 15;
   subtype IP_Address is String(IP_Index);

   ------------------------------
   -- Default Initial Values --
   ------------------------------
   defaultMessage: constant MESSAGE :=
      "********************************************************************************
      ******************";
   defaultStatus: constant STATUS := normal;
   defaultSonar: constant SONAR_TYPE := SONAR_TYPE'(0.0, 0.0, 0.0, 0.0);
   defaultControlData: constant ControlData := ControlData'(vx => 0.0,
                                                            vy => 0.0,
                                                            vz => 0.0,
                                                            vTheta => 0.0);
   defaultSensorData: constant SensorData := SensorData'(direction => 0.0,
                                                         depth => 0.0,
                                                         sonar => defaultSonar);
   defaultActuationData: constant ActuationData := ActuationData'
```

```
      (vThruster1 => 0.0,
       vThruster2 => 0.0,
       vThruster3 => 0.0,
       vThruster4 => 0.0,
       vThruster5 => 0.0,
       vThruster6 => 0.0,
       vThruster7 => 0.0,
       vThruster8 => 0.0);
   defaultReceptionTime: constant Ada.Real_Time.Time := Ada.Real_Time.Clock;

end DataTypes;
```

Código Fonte B.2: Arquivo Constants.ads

```
with Ada.Real_Time; use type Ada.Real_Time.Time;
with System, DataTypes;
--# inherit System,
--#         DataTypes,
--#         Ada.Real_Time;

-- Encapsulate the informations for configurations used in the processes
package Constants is

   ------------------------
   -- Timing information --
   ------------------------
   -- Start Time of the program
   START_TIME: constant Ada.Real_Time.Time := Ada.Real_Time.Clock;

   -- Start Time of each periodic Task of the program
   SENSOR_START: constant Ada.Real_Time.Time := START_TIME + Ada.Real_Time.Milliseconds(10);

   COMPASS_START: constant Ada.Real_Time.Time := START_TIME + Ada.Real_Time.Milliseconds(30);

   ALTIMETER_START: constant Ada.Real_Time.Time := START_TIME + Ada.Real_Time.Milliseconds
        (15);

   SONAR_START: constant Ada.Real_Time.Time := START_TIME + Ada.Real_Time.Milliseconds(20);

   STOP_START: constant Ada.Real_Time.Time := START_TIME + Ada.Real_Time.Milliseconds(25);

   -- Period of each periodic Task of the program
   SENSOR_PERIOD: constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(100);

   COMPASS_PERIOD: constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(100);

   ALTIMETER_PERIOD: constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(100);

   SONAR_PERIOD: constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(100);

   STOP_PERIOD: constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(100);

   ----------------
   -- Priorities --
   ----------------
   TASK_PRIORITY : constant System.Priority := 10;
   CHANNEL_PRIORITY : constant System.Priority := 10;

   ------------------------
   -- Socket Information --
   ------------------------
   BASEREADER_PORT: constant Integer := 14001;

   THRUSTERWRITER_IP: constant DataTypes.IP_Address := "143.107.099.227";
   THRUSTERWRITER_PORT: constant Integer := 14002;

   BASEWRITER_IP: constant DataTypes.IP_Address := "143.107.099.227";
   BASEWRITER_PORT: constant Integer := 14000;

end Constants;
```

Código Fonte B.3: Arquivo Main_ROV_Embedded.adb

```
-- Enables the Ravenscar Profile
pragma Profile(Ravenscar);

with BaseReader,
     Remote,
     BlackBoard,
     Compass,
     Altimeter,
     Sonar,
     Sensor,
     BaseWriter,
     S1,
     S2,
     Move,
     AvoidCollision,
     Actuator,
     Stop,
     ThrusterWriter;
--# inherit BaseReader,
--#         Remote,
--#         MsgChannel,
--#         ControlDataChannel,
--#         BlackBoard,
--#         Compass,
--#         Altimeter,
--#         Sonar,
--#         Sensor,
--#         BaseWriter,
--#         S1,
--#         S2,
--#         Move,
--#         AvoidCollision,
--#         Actuator,
--#         Stop,
--#         ThrusterWriter,
--#         ActuationDataChannel,
--#         StatusChannel,
```

```
--#           Ada.Real_Time;
--# main_program;
--# global in       Ada.Real_Time.ClockTime;
--#          in out ControlDataChannel.s1_par_input_d;
--#          in out ControlDataChannel.s1_move_d;
--#          in out ControlDataChannel.s1_move_s;
--#          in out MsgChannel.baseReader_remote_s;
--#          in out MsgChannel.baseReader_remote_d;
--#          in out ControlDataChannel.remote_s1_s;
--#          in out ControlDataChannel.move_s2_s;
--#          in out ControlDataChannel.stop_s2_s;
--#          in out ControlDataChannel.s2_actuator_s;
--#          in out StatusChannel.avoid_move_s;
--#          in out ActuationDataChannel.actuator_thruster_s;
--#          in out ControlDataChannel.s2_actuator_d;
--#          in out StatusChannel.avoid_move_d;
--#          in out ActuationDataChannel.actuator_thruster_d;
--#          in out ControlDataChannel.s2_par_input_d;
--#          in out MsgChannel.sensor_baseWriter_s;
--#          in out MsgChannel.sensor_baseWriter_d;
--#          in out BlackBoard.blckBoard;
--# derives ControlDataChannel.s1_par_input_d,
--#         ControlDataChannel.s1_move_d,
--#         ControlDataChannel.s1_move_s,
--#         MsgChannel.baseReader_remote_s,
--#         MsgChannel.baseReader_remote_d,
--#         ControlDataChannel.remote_s1_s,
--#         ControlDataChannel.move_s2_s,
--#         ControlDataChannel.stop_s2_s,
--#         ControlDataChannel.s2_actuator_s,
--#         StatusChannel.avoid_move_s,
--#         ActuationDataChannel.actuator_thruster_s,
--#         ControlDataChannel.s2_actuator_d,
--#         StatusChannel.avoid_move_d,
--#         ActuationDataChannel.actuator_thruster_d,
--#         ControlDataChannel.s2_par_input_d        from ControlDataChannel.s1_par_input_d,
--#                                                       ControlDataChannel.s1_move_d,
--#                                                       ControlDataChannel.s1_move_s,
--#                                                       MsgChannel.baseReader_remote_s,
--#                                                       MsgChannel.baseReader_remote_d,
--#                                                       ControlDataChannel.remote_s1_s,
--#                                                       ControlDataChannel.move_s2_s,
--#                                                       ControlDataChannel.stop_s2_s,
--#                                                       ControlDataChannel.s2_actuator_s,
--#                                                       StatusChannel.avoid_move_s,
--#                                                       ActuationDataChannel.
    actuator_thruster_s,
--#                                                       ControlDataChannel.s2_actuator_d,
--#                                                       StatusChannel.avoid_move_d,
--#                                                       ActuationDataChannel.
    actuator_thruster_d,
--#                                                       ControlDataChannel.s2_par_input_d &
--#         MsgChannel.sensor_baseWriter_s,
--#         MsgChannel.sensor_baseWriter_d           from MsgChannel.sensor_baseWriter_s,
--#                                                       MsgChannel.sensor_baseWriter_d &
--#         BlackBoard.blckBoard                     from ControlDataChannel.s1_par_input_d,
--#                                                       ControlDataChannel.s1_move_d,
--#                                                       ControlDataChannel.s1_move_s,
--#                                                       MsgChannel.baseReader_remote_s,
--#                                                       MsgChannel.baseReader_remote_d,
--#                                                       ControlDataChannel.remote_s1_s,
--#                                                       ControlDataChannel.move_s2_s,
--#                                                       ControlDataChannel.stop_s2_s,
--#                                                       ControlDataChannel.s2_actuator_s,
--#                                                       StatusChannel.avoid_move_s,
--#                                                       ActuationDataChannel.
    actuator_thruster_s,
--#                                                       ControlDataChannel.s2_actuator_d,
--#                                                       StatusChannel.avoid_move_d,
--#                                                       ActuationDataChannel.
    actuator_thruster_d,
--#                                                       ControlDataChannel.s2_par_input_d,
--#                                                       Ada.Real_Time.ClockTime &
--#         null                                     from BlackBoard.blckBoard;
procedure Main_ROV_Embedded
--# derives ;
is
   pragma Priority(10);  -- a priority is mandatory and must appear here
begin
   null; -- all the real work is done somewhere else
end Main_ROV_Embedded;
```

Código Fonte B.4: Arquivo ActuationDataChannel.ads

```
with Constants, DataTypes;
--# inherit Constants,
--#         DataTypes;

-- A channel has two protected objects: Data and Sync. The Data protected
-- object is the data communicated through the channel. Sync is a synchronizer.
package ActuationDataChannel
--# own protected actuator_thruster_d : Data (priority => Constants.CHANNEL_PRIORITY,
--#                                          suspendable);
--#     protected actuator_thruster_s : Sync (priority => Constants.CHANNEL_PRIORITY,
--#                                          suspendable);
is

   protected type Data is

      pragma Priority(Constants.CHANNEL_PRIORITY);

      entry get(someData : out DataTypes.ActuationData);
      --# global in out Data;
      --# derives Data,
      --#         someData from Data;

      procedure put(someData: in DataTypes.ActuationData);
      --# global in out Data;
      --# derives Data from *,
      --#                   someData;

   private
      chData : DataTypes.ActuationData := DataTypes.defaultActuationData;
```

```
      readyToRead : Boolean := False; -- Initially, there is no data to read
   end Data;

   protected type Sync is

      pragma Priority(Constants.CHANNEL_PRIORITY);

      entry stay;
      --# global in out Sync;
      --# derives Sync from *;

      procedure proceed;
      --# global in out Sync;
      --# derives Sync from *;

   private
      hasRead : Boolean := False;
   end Sync;

   ----------------------------
   -- Channel Instantiations --
   ----------------------------
   actuator_thruster_d: Data;
   actuator_thruster_s: Sync;

end ActuationDataChannel;
```

Código Fonte B.5: Arquivo ActuationDataChannel.adb

```
package body ActuationDataChannel is

   protected body Data is
      entry get(someData : out DataTypes.ActuationData) when readyToRead
      --# global in       chData;
      --#           out readyToRead;
      --# derives readyToRead from  &
      --#         someData    from chData;
      is
      begin
         -- Read the encapsulated data
         someData := chData;
         -- Block successive readings
         readyToRead := False;
      end get;

      procedure put(someData : in DataTypes.ActuationData)
      --# global out readyToRead;
      --#        out chData;
      --# derives readyToRead from  &
      --#         chData      from someData;
      is
      begin
         -- Update the encapsulated data
         chData := someData;
         -- Permit for the next writing operation
         readyToRead := True;
      end put;
   end Data;

   protected body Sync is
      entry stay when hasRead
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Stay for the data to be read
         hasRead := False;
      end stay;

      procedure proceed
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Declare the completion of a reading operation
         hasRead := True;
      end proceed;
   end Sync;

end ActuationDataChannel;
```

Código Fonte B.6: Arquivo Actuator.ads

```
with Constants, DataTypes;
--# inherit Constants,
--#         DataTypes,
--#         ControlDataChannel,
--#         ActuationDataChannel;

package Actuator
--# own task actuatorTask : Actuator_task;
is

   ------------------------------
   -- Task that do the real job --
   ------------------------------
   task type Actuator_task
   --# global in out ControlDataChannel.s2_actuator_d;
   --#               out ActuationDataChannel.actuator_thruster_d;
   --#               out ControlDataChannel.s2_actuator_s;
   --#               out ActuationDataChannel.actuator_thruster_s;
   --# derives ActuationDataChannel.actuator_thruster_d,
   --#         ControlDataChannel.s2_actuator_s,
   --#         ControlDataChannel.s2_actuator_d,
   --#         ActuationDataChannel.actuator_thruster_s from  &
   --#         null                                    from ControlDataChannel.s2_actuator_d
       ;
   --# declare Suspends => (ControlDataChannel.s2_actuator_d, ActuationDataChannel.
       actuator_thruster_s);
```

```
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end Actuator_task;

   ------------------------------
   -- Methods of the package --
   ------------------------------
   procedure generateSetpoints(ctrlData: in DataTypes.ControlData;
                               setpoints: out DataTypes.ActuationData);
   --# derives setpoints from ctrlData;

   --------------------------
   -- Task instantiation --
   --------------------------
   actuatorTask: Actuator_task;

end Actuator;
```

Código Fonte B.7: Arquivo Actuator.adb

```
with ControlDataChannel, ActuationDataChannel;
with Ada.Text_IO, Ada.Exceptions;

package body Actuator is

   task body Actuator_task is

      -- control data to be transformed
      ctrlData: DataTypes.ControlData := DataTypes.defaultControlData;

      -- setpoints to the thrusters
      setpoints: DataTypes.ActuationData := DataTypes.defaultActuationData;

   begin

      loop

         -- receive data from the input channel
         ControlDataChannel.s2_actuator_d.get(ctrlData);
         ControlDataChannel.s2_actuator_s.proceed;

         -- generate the setpoints to the thrusters
         generateSetpoints(ctrlData, setpoints);

         -- send data through the output channel
         ActuationDataChannel.actuator_thruster_d.put(setpoints);
         ActuationDataChannel.actuator_thruster_s.stay;

      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in Actuator");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end Actuator_task;


   procedure generateSetpoints(ctrlData: in DataTypes.ControlData;
                               setpoints: out DataTypes.ActuationData) is
      pragma Inline(generateSetpoints);
   begin
      setpoints.vThruster1 := 10.0;
      setpoints.vThruster2 := 20.0;
      setpoints.vThruster3 := 30.0;
      setpoints.vThruster4 := 40.0;
      setpoints.vThruster5 := 50.0;
      setpoints.vThruster6 := 60.0;
      setpoints.vThruster7 := 70.0;
      setpoints.vThruster8 := 80.0;
   end generateSetpoints;

end Actuator;
```

Código Fonte B.8: Arquivo Altimeter.ads

```
with DataTypes, Constants;
--# inherit DataTypes,
--#         Constants,
--#         BlackBoard;

package Altimeter
--# own task altimeterTask: Altimeter_task;
is

   ----------------------------------
   -- Task that do the real job --
   ----------------------------------
   task type Altimeter_task
   --# global out BlackBoard.blckBoard;
   --# derives BlackBoard.blckBoard from ;
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end Altimeter_task;

   ------------------------------
   -- Methods of the package --
   ------------------------------
   procedure getDepth(depth: out DataTypes.ALTIMETER_TYPE);
   --# derives depth from ;

   procedure saveDepth(depth: in DataTypes.ALTIMETER_TYPE);
   --# global out BlackBoard.blckBoard;
   --# derives BlackBoard.blckBoard from depth;

   --------------------------
   -- Task instantiation --
   --------------------------
   altimeterTask: Altimeter_task;
```

```
end Altimeter;
```

Código Fonte B.9: Arquivo Altimeter.adb

```
with Ada.Real_Time; use type Ada.Real_Time.Time;
with BlackBoard;
with Ada.Exceptions, Ada.Text_IO;

package body Altimeter is

   task body Altimeter_task is

      -- stores the altimeter readings
      depth: DataTypes.ALTIMETER_TYPE;

      -- time attibutes (only for Periodic Processes)
      nextExecution: Ada.Real_Time.Time := Constants.ALTIMETER_START;
      PERIOD: constant Ada.Real_Time.Time_Span := Constants.ALTIMETER_PERIOD;

      -- time method (only for Periodic Processes)
      procedure getNextExecution is
      begin
         nextExecution := nextExecution + PERIOD;
      end getNextExecution;

   begin
      loop
         delay until nextExecution;

         -- get the depth from the altimeter
         getDepth(depth);

         -- save the depth into the BlackBoard
         saveDepth(depth);

         getNextExecution;
      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in Altimeter");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end Altimeter_task;


   procedure getDepth(depth: out DataTypes.ALTIMETER_TYPE) is
      pragma Inline(getDepth);
   begin
      depth := 15.0;
   end getDepth;


   procedure saveDepth(depth: in DataTypes.ALTIMETER_TYPE) is
      pragma Inline(saveDepth);
   begin
      BlackBoard.blckBoard.setDepth(depth);
   end saveDepth;

end Altimeter;
```

Código Fonte B.10: Arquivo AvoidCollision.ads

```
with DataTypes, Constants;
--# inherit DataTypes,
--#         Constants,
--#         StatusChannel,
--#         BlackBoard;

package AvoidCollision
--# own task avoidCollisionTask : AvoidCollision_task;
is

   ---------------------------------
   -- Task that do the real job --
   ---------------------------------
   task type AvoidCollision_task
   --# global in      BlackBoard.blckBoard;
   --#             out StatusChannel.avoid_move_d;
   --#             out StatusChannel.avoid_move_s;
   --# derives StatusChannel.avoid_move_d,
   --#         StatusChannel.avoid_move_s from &
   --#         null                        from BlackBoard.blckBoard;
   --# declare Suspends => StatusChannel.avoid_move_s;
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end AvoidCollision_task;

   ------------------------------
   -- Methods of the package --
   ------------------------------
   procedure getSonarReadings(sonars: out DataTypes.SONAR_TYPE);
   --# global in BlackBoard.blckBoard;
   --# derives sonars from BlackBoard.blckBoard;

   procedure generateAvoid(sonars: in DataTypes.SONAR_TYPE;
                           LIMIT: in DataTypes.Sonar_Range;
                           avoidData: out DataTypes.STATUS);
   --# derives avoidData from sonars,
   --#                        LIMIT;

   --------------------------
   -- Task instantiation --
   --------------------------
   avoidCollisionTask: AvoidCollision_task;

end AvoidCollision;
```

Código Fonte B.11: Arquivo AvoidCollision.adb

```
with Ada.Real_Time; use type Ada.Real_Time.Time;
with StatusChannel, BlackBoard;
with Ada.Exceptions, Ada.text_IO;

package body AvoidCollision is

   task body AvoidCollision_task is

      -- to indicate the status of the robot
      avoidData: DataTypes.STATUS := DataTypes.defaultStatus;

      -- readings of the sonar
      sonars: DataTypes.SONAR_TYPE;

      -- limit that indicates the presence of obstacles
      LIMIT: constant DataTypes.Sonar_Range := 0.8;

      -- time attibutes (only for Periodic Processes)
      nextExecution: Ada.Real_Time.Time := Constants.COMPASS_START;
      PERIOD: constant Ada.Real_Time.Time_Span := Constants.COMPASS_PERIOD;

      -- time method (only for Periodic Processes)
      procedure getNextExecution is
      begin
         nextExecution := nextExecution + PERIOD;
      end getNextExecution;

   begin
      loop
         delay until nextExecution;

         -- get the sonar readings in the BlackBoard
         getSonarReadings(sonars);

         -- interpret the sonar readings
         generateAvoid(sonars, LIMIT, avoidData);

         -- send data through the output channel
         StatusChannel.avoid_move_d.put(avoidData);
         StatusChannel.avoid_move_s.stay;

         getNextExecution;
      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in AvoidCollision");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end AvoidCollision_task;


   procedure getSonarReadings(sonars: out DataTypes.SONAR_TYPE) is
      pragma Inline(getSonarReadings);
   begin
      BlackBoard.blckBoard.getSonar(sonars);
   end getSonarReadings;


   procedure generateAvoid(sonars: in DataTypes.SONAR_TYPE;
                           LIMIT: in DataTypes.Sonar_Range;
                           avoidData: out DataTypes.STATUS) is
      pragma Inline(generateAvoid);
   begin
      for i in sonars'range loop
         if sonars(i) >= LIMIT then
            avoidData := DataTypes.alert;
         else
            avoidData := DataTypes.normal;
         end if;
      end loop;
   end generateAvoid;

end AvoidCollision;
```

Código Fonte B.12: Arquivo BaseReader.ads

```
with DataTypes, Constants, UDP_Sockets;
--# inherit DataTypes,
--#         Constants,
--#         MsgChannel,
--#         UDP_Sockets,
--#         BlackBoard,
--#         Ada.Real_Time;

package BaseReader
--# own task baseReaderTask : BaseReader_task;
is

   --------------------------------
   -- Task that do the real job --
   --------------------------------
   task type BaseReader_task
   --# global in      Ada.Real_Time.ClockTime;
   --#             out MsgChannel.baseReader_remote_d;
   --#             out MsgChannel.baseReader_remote_s;
   --#             out BlackBoard.blckBoard;
   --# derives MsgChannel.baseReader_remote_d,
   --#         MsgChannel.baseReader_remote_s from  &
   --#         BlackBoard.blckBoard              from Ada.Real_Time.ClockTime;
   --# declare Suspends => MsgChannel.baseReader_remote_s;
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end BaseReader_task;

   -----------------------------
   -- Methods of the package --
   -----------------------------
   procedure getMessage(socket: in out UDP_Sockets.UDP_Socket;
                        receivedMsg: out DataTypes.MESSAGE);
   --# derives receivedMsg from socket &
```

```
   --#          socket       from ;
   --# declare delay;

   procedure saveReceptionTime;
   --# global in      Ada.Real_Time.ClockTime;
   --#            out BlackBoard.blckBoard;
   --# derives BlackBoard.blckBoard from Ada.Real_Time.ClockTime;

   ------------------------
   -- Task instantiation --
   ------------------------
   baseReaderTask: BaseReader_task;

end BaseReader;
```

Código Fonte B.13: Arquivo BaseReader.adb

```
with MsgChannel, Ada.Real_Time, BlackBoard;
with Ada.Text_IO, Ada.Exceptions;

package body BaseReader is

   task body BaseReader_task is

      -- message received from the Base Station
      receivedMsg: DataTypes.MESSAGE := DataTypes.defaultMessage;
--        receivedMsg: DataTypes.MESSAGE := "$x1.00y0.50z0.30t-1.0$
     ********************************************************************************";

      -- socket to receive messages, not in the formal specification
      mySocket: UDP_Sockets.UDP_Socket;

   begin
      -- Create and Initialize the socket
      UDP_Sockets.Create_Receiver_Socket(mySocket, Constants.BASEREADER_PORT);

      loop

         -- get the message sent from the base station
         getMessage(mySocket, receivedMsg);

         -- store the reception time
         saveReceptionTime;

         -- send it through the output channel
         MsgChannel.baseReader_remote_d.put(receivedMsg);
         MsgChannel.baseReader_remote_s.stay;

      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in BaseReader");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end BaseReader_task;

   -- wait for a message from the network
   procedure getMessage(socket: in out UDP_Sockets.UDP_Socket;
                        receivedMsg: out DataTypes.MESSAGE) is
      pragma Inline(getMessage);
   begin
      UDP_Sockets.Receive_Message(socket, receivedMsg);
   end getMessage;

   -- save the time of the message reception
   procedure saveReceptionTime is
      pragma Inline(saveReceptionTime);
   begin
      BlackBoard.blckBoard.setReceptionTime(Ada.Real_Time.Clock);
   end saveReceptionTime;

end BaseReader;
```

Código Fonte B.14: Arquivo BaseWriter.ads

```
with DataTypes, Constants, UDP_Sockets;
--# inherit DataTypes,
--#         Constants,
--#         MsgChannel,
--#         UDP_Sockets;

package BaseWriter
--# own task baseWriterTask : BaseWriter_task;
is

   ------------------------------
   -- Task that do the real job --
   ------------------------------
   task type BaseWriter_task
   --# global in out MsgChannel.sensor_baseWriter_d;
   --#            out MsgChannel.sensor_baseWriter_s;
   --# derives MsgChannel.sensor_baseWriter_d,
   --#         MsgChannel.sensor_baseWriter_s from &
   --#         null                          from MsgChannel.sensor_baseWriter_d;
   --# declare Suspends => MsgChannel.sensor_baseWriter_d;
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end BaseWriter_task;

   ---------------------------
   -- Methods of the package --
   ---------------------------
   procedure sendMessage(socket: in out UDP_Sockets.UDP_Socket;
                         msg: in DataTypes.MESSAGE);
   --# derives socket from *,
   --#                     msg;
```

```
   ------------------------
   -- Task instantiation --
   ------------------------
   baseWriterTask: BaseWriter_task;

end BaseWriter;
```

Código Fonte B.15: Arquivo BaseWriter.adb

```
with MsgChannel, Ada.Text_IO, Ada.Exceptions;

package body BaseWriter is

   task body BaseWriter_task is

      -- message received from the Base Station
      msgToSend: DataTypes.MESSAGE := DataTypes.defaultMessage;

      -- socket to receive messages, not in the formal specification
      mySocket: UDP_Sockets.UDP_Socket;

   begin
      -- Create and Initialize the socket
      UDP_Sockets.Create_Sender_Socket(mySocket,
                                       Constants.BASEWRITER_IP,
                                       Constants.BASEWRITER_PORT);

      loop

         -- receive data from the input channel
         MsgChannel.sensor_baseWriter_d.get(msgToSend);
         MsgChannel.sensor_baseWriter_s.proceed;

         -- send the message to the Base Station
         sendMessage(mySocket, msgToSend);

--             -- PARA TESTE
--             for I in 1 .. msgToSend'Last loop
--                Ada.Text_IO.Put(msgToSend(I));
--             end loop;
--             Ada.Text_IO.New_Line;

      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in BaseWriter");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end BaseWriter_task;

   -- send a message over the network
   procedure sendMessage(socket: in out UDP_Sockets.UDP_Socket;
                         msg: in DataTypes.MESSAGE) is
      pragma Inline(sendMessage);
   begin
      UDP_Sockets.Send_Message(socket, msg);
   end sendMessage;

end BaseWriter;
```

Código Fonte B.16: Arquivo BlackBoard.ads

```
with DataTypes, Constants, Ada.Real_Time;
--# inherit DataTypes,
--#         Constants,
--#         Ada.Real_Time;

-- Package used as a common region for sharing sensor and time
-- information between the processes of the system.
package BlackBoard
--# own protected blckBoard : The_BlackBoard (priority => Constants.CHANNEL_PRIORITY);
is

   protected type The_BlackBoard is

      pragma Priority(Constants.CHANNEL_PRIORITY);

      procedure setDirection(reading: in DataTypes.COMPASS_TYPE);
      --# global out The_BlackBoard;
      --# derives The_BlackBoard from reading;

      procedure setDepth(reading: in DataTypes.ALTIMETER_TYPE);
      --# global out The_BlackBoard;
      --# derives The_BlackBoard from reading;

      procedure setSonar(reading: in DataTypes.SONAR_TYPE);
      --# global out The_BlackBoard;
      --# derives The_BlackBoard from reading;

      procedure getSonar(reading: out DataTypes.SONAR_TYPE);
      --# global in The_BlackBoard;
      --# derives reading from The_BlackBoard;

      procedure getSensorReadings(allReadings: out DataTypes.SensorData);
      --# global in The_BlackBoard;
      --# derives allReadings from The_BlackBoard;

      procedure getReceptionTime(theTime: out Ada.Real_Time.Time);
      --# global in The_BlackBoard;
      --# derives theTime from The_BlackBoard;

      procedure setReceptionTime(theTime: in Ada.Real_Time.Time);
      --# global out The_BlackBoard;
      --# derives The_BlackBoard from theTime;

   private
      sensorReadings: DataTypes.SensorData := DataTypes.defaultSensorData;
      receptionTime: Ada.Real_Time.Time := DataTypes.defaultReceptionTime;
```

```
   end The_BlackBoard;

   ----------------------------------
   -- Instance of the BlackBoard --
   ----------------------------------
   blckBoard: The_BlackBoard;

end BlackBoard;
```

Código Fonte B.17: Arquivo BlackBoard.adb

```
package body BlackBoard is

   protected body The_BlackBoard is

      procedure setDirection(reading: in DataTypes.COMPASS_TYPE)
      --# global out sensorReadings.direction;
      --# derives sensorReadings.direction from reading;
      is
         pragma Inline(setDirection);
      begin
         sensorReadings.direction := reading;
      end setDirection;

      procedure setDepth(reading: in DataTypes.ALTIMETER_TYPE)
      --# global out sensorReadings.direction;
      --# derives sensorReadings.direction from reading;
      is
         pragma Inline(setDepth);
      begin
         sensorReadings.depth := reading;
      end setDepth;

      procedure setSonar(reading: in DataTypes.SONAR_TYPE)
      --# global out sensorReadings.direction;
      --# derives sensorReadings.direction from reading;
      is
         pragma Inline(setSonar);
      begin
         sensorReadings.sonar := reading;
      end setSonar;

      procedure getSonar(reading: out DataTypes.SONAR_TYPE)
      --# global in sensorReadings.sonar;
      --# derives reading from sensorReadings.sonar;
      is
         pragma Inline(getSonar);
      begin
         reading := sensorReadings.sonar;
      end getSonar;

      procedure getSensorReadings(allReadings: out DataTypes.SensorData)
      --# global in sensorReadings;
      --# derives allReadings from sensorReadings;
      is
         pragma Inline(getSensorReadings);
      begin
         allReadings := sensorReadings;
      end getSensorReadings;

      procedure getReceptionTime(theTime: out Ada.Real_Time.Time)
      --# global in receptionTime;
      --# derives theTime from receptionTime;
      is
      begin
         theTime := receptionTime;
      end getReceptionTime;

      procedure setReceptionTime(theTime: in Ada.Real_Time.Time)
      --# global out receptionTime;
      --# derives receptionTime from theTime;
      is
      begin
         receptionTime := theTime;
      end setReceptionTime;

   end The_BlackBoard;

end BlackBoard;
```

Código Fonte B.18: Arquivo Compass.ads

```
with DataTypes, Constants;
--# inherit DataTypes,
--#         Constants,
--#         BlackBoard;

package Compass
--# own task compassTask : Compass_task;
is

   ----------------------------------
   -- Task that do the real job --
   ----------------------------------
   task type Compass_task
   --# global out BlackBoard.blckBoard;
   --# derives BlackBoard.blckBoard from ;
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end Compass_task;

   ------------------------------
   -- Methods of the package --
   ------------------------------
   procedure getDirection(direction: out DataTypes.COMPASS_TYPE);
   --# derives direction from ;

   procedure saveDirection(direction: in DataTypes.COMPASS_TYPE);
   --# global out BlackBoard.blckBoard;
   --# derives BlackBoard.blckBoard from direction;
```

```
   -------------------------
   -- Task instantiation --
   -------------------------
   compassTask: Compass_task;

end Compass;
```

Código Fonte B.19: Arquivo Compass.adb

```
with Ada.Real_Time; use type Ada.Real_Time.Time;
with BlackBoard;
with Ada.Text_IO, Ada.Exceptions;

package body Compass is

   task body Compass_task is

      -- store the compass readings
      direction: DataTypes.COMPASS_TYPE;

      -- time attibutes (only for Periodic Processes)
      nextExecution: Ada.Real_Time.Time := Constants.COMPASS_START;
      PERIOD: constant Ada.Real_Time.Time_Span := Constants.COMPASS_PERIOD;

      -- time method (only for Periodic Processes)
      procedure getNextExecution is
      begin
         nextExecution := nextExecution + PERIOD;
      end getNextExecution;

   begin
      loop
         delay until nextExecution;

         -- get the direction from the compass
         getDirection(direction);

         -- save the direction into the BlackBoard
         saveDirection(direction);

         getNextExecution;
      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in Compass");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end Compass_task;


   procedure getDirection(direction: out DataTypes.COMPASS_TYPE) is
      pragma Inline(getDirection);
   begin
      direction := 10.0;
   end getDirection;


   procedure saveDirection(direction: in DataTypes.COMPASS_TYPE) is
      pragma Inline(saveDirection);
   begin
      BlackBoard.blckBoard.setDirection(direction);
   end saveDirection;

end Compass;
```

Código Fonte B.20: Arquivo ControlDataChannel.ads

```
with Constants, DataTypes;
--# inherit Constants,
--#         DataTypes;

-- A channel has two protected objects: Data and Sync. The Data protected
-- object is the data communicated through the channel. Sync is a synchronizer.
package ControlDataChannel
--# own protected s1_par_input_d : Data (priority => Constants.CHANNEL_PRIORITY,
--#                                      suspendable);
--#     protected remote_s1_s    : Sync (priority => Constants.CHANNEL_PRIORITY,
--#                                      suspendable);
--#     protected s1_move_d      : Data (priority => Constants.CHANNEL_PRIORITY,
--#                                      suspendable);
--#     protected s1_move_s      : Sync (priority => Constants.CHANNEL_PRIORITY,
--#                                      suspendable);
--#     protected move_s2_s      : Sync (priority => Constants.CHANNEL_PRIORITY,
--#                                      suspendable);
--#     protected s2_par_input_d : Data (priority => Constants.CHANNEL_PRIORITY,
--#                                      suspendable);
--#     protected stop_s2_s      : Sync (priority => Constants.CHANNEL_PRIORITY,
--#                                      suspendable);
--#     protected s2_actuator_d  : Data (priority => Constants.CHANNEL_PRIORITY,
--#                                      suspendable);
--#     protected s2_actuator_s  : Sync (priority => Constants.CHANNEL_PRIORITY,
--#                                      suspendable);
is

   protected type Data is

      pragma Priority(Constants.CHANNEL_PRIORITY);

      entry get(someData : out DataTypes.ControlData);
      --# global in out Data;
      --# derives Data,
      --#         someData from Data;

      procedure put(someData: in DataTypes.ControlData);
      --# global in out Data;
      --# derives Data from *,
```

```
      --#                       someData;

   private
      chData : DataTypes.ControlData := DataTypes.defaultControlData;
      readyToRead : Boolean := False; -- Initially, there is no data to read
   end Data;


   protected type Sync is

      pragma Priority(Constants.CHANNEL_PRIORITY);

      entry stay;
      --# global in out Sync;
      --# derives Sync from *;

      procedure proceed;
      --# global in out Sync;
      --# derives Sync from *;

   private
      hasRead : Boolean := False;
   end Sync;

   ----------------------------
   -- Channel Instantiations --
   ----------------------------
   s1_par_input_d: Data; -- data to remote and autonomous channels (parallel)
   remote_s1_s: Sync;

   s1_move_d: Data;
   s1_move_s: Sync;

   s2_par_input_d: Data; -- data to move and stop channels (parallel)
   move_s2_s: Sync;
   stop_s2_s: Sync;

   s2_actuator_d: Data;
   s2_actuator_s: Sync;

end ControlDataChannel;
```

Código Fonte B.21: Arquivo ControlDataChannel.adb

```
package body ControlDataChannel is

   protected body Data is
      entry get(someData : out DataTypes.ControlData) when readyToRead
      --# global in      chData;
      --#            out readyToRead;
      --# derives readyToRead from  &
      --#         someData     from chData;
      is
      begin
         -- Read the encapsulated data
         someData := chData;
         -- Block successive readings
         readyToRead := False;
      end get;

      procedure put(someData : in DataTypes.ControlData)
      --# global out readyToRead;
      --#        out chData;
      --# derives readyToRead from  &
      --#         chData       from someData;
      is
      begin
         -- Update the encapsulated data
         chData := someData;
         -- Permit for the next writing operation
         readyToRead := True;
      end put;
   end Data;

   protected body Sync is
      entry stay when hasRead
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Stay for the data to be read
         hasRead := False;
      end stay;

      procedure proceed
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Declare the completion of a reading operation
         hasRead := True;
      end proceed;
   end Sync;

end ControlDataChannel;
```

Código Fonte B.22: Arquivo Decoder.ads

```
with DataTypes;
--# inherit Datatypes;

-- Package used to decode the messages received from the Base Station.
package Decoder
--# own endValue,
--#     endMsg,
--#     sensorId,
--#     firstIndex,
--#     lastIndex;
--# initializes endValue,
--#             endMsg,
```

```
--#              sensorId,
--#              firstIndex,
--#              lastIndex;
is

   procedure convertToStructure(msg: in DataTypes.MESSAGE;
                                str: out DataTypes.ControlData);
   --# global in out sensorId;
   --#        in out firstIndex;
   --#           out endMsg;
   --#           out lastIndex;
   --# derives endMsg,
   --#         sensorId    from  &
   --#         firstIndex  from sensorId &
   --#         lastIndex   from firstIndex &
   --#         str         from msg;

private

   -- Variables used during the decodification process
   endValue, endMsg: Boolean;
   sensorId, firstIndex, lastIndex: DataTypes.Message_Index;

   procedure saveSensorValue(msg: in DataTypes.MESSAGE;
                             str: out DataTypes.ControlData);
   --# global in sensorId;
   --# derives str from sensorId,
   --#                  msg;

   procedure getSensorValue(msg: in DataTypes.MESSAGE;
                            value: out DataTypes.SPEED_TYPE);
   --# global in     firstIndex;
   --#        in out lastIndex;
   --#           out endValue;
   --#           out endMsg;
   --#           out sensorId;
   --# derives endValue,
   --#         endMsg     from  &
   --#         sensorId,
   --#         lastIndex from lastIndex &
   --#         value      from firstIndex,
   --#                         lastIndex,
   --#                         msg;

   function extractNumber (Item: DataTypes.MESSAGE;
                           Since: DataTypes.Message_Index;
                           To: DataTypes.Message_Index)
                           return DataTypes.SPEED_TYPE;

end Decoder;
```

Código Fonte B.23: Arquivo Decoder.adb

```
with Ada.Characters.Handling;

package body Decoder is

   procedure convertToStructure(msg: in DataTypes.MESSAGE;
                                str: out DataTypes.ControlData)
   is
      pragma Inline(convertToStructure);
   begin
      if msg(1) = '$' then
         endMsg := False;
         sensorId := 2;
         loop
            firstIndex := sensorId + 1;
            lastIndex := firstIndex + 1;
            saveSensorValue(msg, str);
            exit when endMsg;
         end loop;
      end if;
   end convertToStructure;

   procedure saveSensorValue(msg: in DataTypes.MESSAGE;
                             str: out DataTypes.ControlData)
   is
      pragma Inline(saveSensorValue);
   begin
      case msg(sensorId) is
         when 'x' => getSensorValue(msg, str.vx);
         when 'y' => getSensorValue(msg, str.vy);
         when 'z' => getSensorValue(msg, str.vz);
         when 't' => getSensorValue(msg, str.vTheta);
         when others => endMsg := True; -- stop the loop and discard the message
      end case;

   end saveSensorValue;

   procedure getSensorValue(msg: in DataTypes.MESSAGE; value: out DataTypes.SPEED_TYPE) is
      pragma Inline(getSensorValue);
   begin
      endValue := False;
      loop
         if msg(lastIndex) = '$' then -- found the end of the message
            endValue := True;
            endMsg := True;
         else
            if Ada.Characters.Handling.Is_Digit(msg(lastIndex)) or
              msg(lastIndex) = '.' or
              msg(lastIndex) = '-'
            then
               lastIndex := lastIndex + 1;
            else
               endValue := True;
            end if;
         end if;
         exit when endValue;
      end loop;

      sensorId := lastIndex;

      -- converte a string indo de firstIndex até lastIndex para Float.
```

```
      value := Decoder.extractNumber(msg, firstIndex, lastIndex -1);

   end getSensorValue;

   -- to extract the substring with the correct indexes
   function extractNumber (Item : DataTypes.MESSAGE;
                           Since : DataTypes.Message_Index;
                           To : DataTypes.Message_Index)
                           return DataTypes.SPEED_TYPE is
      pragma Inline(extractNumber);
      result: DataTypes.SPEED_TYPE;
   begin
      declare
         subtype Str_Number is String(1..To - Since + 1);
      begin

         result := Float'Value (Str_Number (Item (Since..To)));

         -- check the consistency (in case of errors, set zero for safety)
         if result < DataTypes.SPEED_TYPE'First
           or
             result > DataTypes.SPEED_TYPE'Last
         then
            result := 0.0;
         end if;

         return result;
      end;
   end extractNumber;

end Decoder;
```

Código Fonte B.24: Arquivo Encoder.ads

```
with DataTypes;
--# inherit DataTypes;

-- Package that encodes the SensorData and ActuationData into a MESSAGE type
package Encoder is

   -- transforms a SensorData into a MESSAGE
   procedure sensorToMessage(struct: in DataTypes.SensorData;
                             msg: out DataTypes.MESSAGE);
   --# derives msg from struct;

   -- transforms an ActuationData into a MESSAGE
   procedure actuationToMessage(struct: in DataTypes.ActuationData;
                                msg: out DataTypes.MESSAGE);
   --# derives msg from struct;

end Encoder;
```

Código Fonte B.25: Arquivo Encoder.adb

```
with Ada.Strings.Bounded;
with Ada.Strings.Fixed;

package body Encoder is

   package Bounded_Message is new Ada.Strings.Bounded.Generic_Bounded_Length(DataTypes.
        msgSize);

   procedure sensorToMessage(struct: in DataTypes.SensorData;
                             msg: out DataTypes.MESSAGE)
   is
      pragma Inline(sensorToMessage);
      temp: Bounded_Message.Bounded_String;
   begin
      -- load data into the temporary bounded string
      temp := Bounded_Message.Append(temp, '$');
      temp := Bounded_Message.Append(temp, 'd');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.
           direction), Ada.Strings.Both));
      temp := Bounded_Message.Append(temp, 'h');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.depth),
           Ada.Strings.Both));
      temp := Bounded_Message.Append(temp, '$');

      -- transfer data into the message
      for i in 1 .. Bounded_Message.Length(temp) loop
         msg(i) := Bounded_Message.Element(temp, i);
      end loop;
   end sensorToMessage;

   procedure actuationToMessage(struct: in DataTypes.ActuationData;
                                msg: out DataTypes.MESSAGE)
   is
      pragma Inline(actuationToMessage);
      temp: Bounded_Message.Bounded_String;
   begin
      -- load data into the temporary bounded string
      temp := Bounded_Message.Append(temp, '$');
      temp := Bounded_Message.Append(temp, 'i');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.
           vThruster1), Ada.Strings.Both));
      temp := Bounded_Message.Append(temp, 'j');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.
           vThruster2), Ada.Strings.Both));
      temp := Bounded_Message.Append(temp, 'k');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.
           vThruster3), Ada.Strings.Both));
      temp := Bounded_Message.Append(temp, 'l');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.
           vThruster4), Ada.Strings.Both));
      temp := Bounded_Message.Append(temp, 'm');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.
           vThruster5), Ada.Strings.Both));
      temp := Bounded_Message.Append(temp, 'n');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.
           vThruster6), Ada.Strings.Both));
```

```
      temp := Bounded_Message.Append(temp, 'o');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.
          vThruster7), Ada.Strings.Both));
      temp := Bounded_Message.Append(temp, 'p');
      temp := Bounded_Message.Append(temp, Ada.Strings.Fixed.Trim(Float'Image(struct.
          vThruster8), Ada.Strings.Both));
      temp := Bounded_Message.Append(temp, '$');

      -- transfer data into the message
      for i in 1 .. Bounded_Message.Length(temp) loop
         msg(i) := Bounded_Message.Element(temp, i);
      end loop;
   end actuationToMessage;

end Encoder;
```

Código Fonte B.26: Arquivo Move.ads

```
with DataTypes, Constants;
use type DataTypes.STATUS;
--# inherit DataTypes,
--#         Constants,
--#         ControlDataChannel,
--#         StatusChannel;

package Move
--# own task moveTask : Move_task;
is

   ------------------------------
   -- Task that do the real job --
   ------------------------------
   task type Move_task
   --# global in out ControlDataChannel.s1_move_d;
   --#        in out StatusChannel.avoid_move_d;
   --#           out ControlDataChannel.s1_move_s;
   --#           out StatusChannel.avoid_move_s;
   --#           out ControlDataChannel.s2_par_input_d;
   --#           out ControlDataChannel.move_s2_s;
   --# derives ControlDataChannel.s1_move_d,
   --#         ControlDataChannel.s1_move_s,
   --#         StatusChannel.avoid_move_d,
   --#         StatusChannel.avoid_move_s,
   --#         ControlDataChannel.s2_par_input_d,
   --#         ControlDataChannel.move_s2_s      from &
   --#         null                              from ControlDataChannel.s1_move_d,
   --#                                                StatusChannel.avoid_move_d;
   --# declare Suspends => (ControlDataChannel.s1_move_d, StatusChannel.avoid_move_d,
        ControlDataChannel.move_s2_s);
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end Move_task;

   ---------------------------
   -- Methods of the package --
   ---------------------------
   procedure getNextMovement(dangerLevel: in DataTypes.STATUS;
                             speedData: in DataTypes.ControlData;
                             avoidData: in out DataTypes.ControlData;
                             movement: out DataTypes.ControlData);
   --# derives movement from avoidData,
   --#                       dangerLevel,
   --#                       speedData &
   --#         avoidData from dangerLevel,
   --#                       speedData;

   ---------------------
   -- Task instantiation --
   ---------------------
   moveTask: Move_task;

end Move;
```

Código Fonte B.27: Arquivo Move.adb

```
with ControlDataChannel, StatusChannel;
with Ada.Text_IO, Ada.Exceptions;

package body Move is

   task body Move_task is

      -- indicates the presence of obstacles
      dangerLevel: DataTypes.STATUS := DataTypes.defaultStatus;

      -- command from the Deliberative Layer
      speedData: DataTypes.ControlData := DataTypes.defaultControlData;

      -- last command used in the vehicle
      avoidData: DataTypes.ControlData := DataTypes.defaultControlData;

      -- effective command, combined from the others
      movement: DataTypes.ControlData := DataTypes.defaultControlData;

   begin

      loop

         -- receive data from the deliberative layer
         ControlDataChannel.s1_move_d.get(speedData);
         ControlDataChannel.s1_move_s.proceed;

         -- receive the data from the AvoidCollision
         StatusChannel.avoid_move_d.get(dangerLevel);
         StatusChannel.avoid_move_s.proceed;

         -- generate the next movement
         getNextMovement(dangerLevel, speedData, avoidData, movement);

         -- send data through the output channel
```

```
         ControlDataChannel.s2_par_input_d.put(movement);
         ControlDataChannel.move_s2_s.stay;

      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in Move");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end Move_task;


   procedure getNextMovement(dangerLevel: in DataTypes.STATUS;
                             speedData: in DataTypes.ControlData;
                             avoidData: in out DataTypes.ControlData;
                             movement: out DataTypes.ControlData) is
      pragma Inline(getNextMovement);
   begin
      if dangerLevel = DataTypes.normal then
         movement := speedData;
         avoidData := speedData;
      else
         movement := avoidData;
      end if;
   end getNextMovement;

end Move;
```

Código Fonte B.28: Arquivo MsgChannel.ads

```
with Constants, DataTypes;
--# inherit Constants,
--#         DataTypes;

-- A channel has two protected objects: Data and Sync. The Data protected
-- object is the data communicated through the channel. Sync is a synchronizer.
package MsgChannel
--# own protected baseReader_remote_d : Data (priority => Constants.CHANNEL_PRIORITY,
--#                                            suspendable);
--#     protected baseReader_remote_s : Sync (priority => Constants.CHANNEL_PRIORITY,
--#                                            suspendable);
--#     protected sensor_baseWriter_d : Data (priority => Constants.CHANNEL_PRIORITY,
--#                                            suspendable);
--#     protected sensor_baseWriter_s : Sync (priority => Constants.CHANNEL_PRIORITY,
--#                                            suspendable);
is

   protected type Data is

      pragma Priority(Constants.CHANNEL_PRIORITY);

      entry get(someData : out DataTypes.MESSAGE);
      --# global in out Data;
      --# derives Data,
      --#         someData from Data;

      procedure put(someData: in DataTypes.MESSAGE);
      --# global in out Data;
      --# derives Data from *,
      --#                   someData;

   private
      chData : DataTypes.MESSAGE := DataTypes.defaultMessage;
      readyToRead : Boolean := False; -- Initially, there is no data to read
   end Data;


   protected type Sync is

      pragma Priority(Constants.CHANNEL_PRIORITY);

      entry stay;
      --# global in out Sync;
      --# derives Sync from *;

      procedure proceed;
      --# global in out Sync;
      --# derives Sync from *;

   private
      hasRead : Boolean := False;
   end Sync;

   -----------------------------
   -- Channel Instantiations --
   -----------------------------
   baseReader_remote_d: Data;
   baseReader_remote_s: Sync;

   sensor_baseWriter_d: Data;
   sensor_baseWriter_s: Sync;

end MsgChannel;
```

Código Fonte B.29: Arquivo MsgChannel.adb

```
package body MsgChannel is

   protected body Data is
      entry get(someData : out DataTypes.MESSAGE) when readyToRead
      --# global in      chData;
      --#            out readyToRead;
      --# derives readyToRead from &
      --#         someData     from chData;
      is
      begin
         -- Read the encapsulated data
         someData := chData;
```

```
         -- Block successive readings
         readyToRead := False;
      end get;

      procedure put(someData : in DataTypes.MESSAGE)
      --# global out readyToRead;
      --#        out chData;
      --# derives readyToRead from &
      --#         chData      from someData;
      is
      begin
         -- Update the encapsulated data
         chData := someData;
         -- Permit for the next writing operation
         readyToRead := True;
      end put;
   end Data;

   protected body Sync is
      entry stay when hasRead
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Stay for the data to be read
         hasRead := False;
      end stay;

      procedure proceed
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Declare the completion of a reading operation
         hasRead := True;
      end proceed;
   end Sync;

end MsgChannel;
```

Código Fonte B.30: Arquivo Remote.ads

```
with Constants, DataTypes;
--# inherit Constants,
--#         DataTypes,
--#         MsgChannel,
--#         Decoder,
--#         ControlDataChannel;

package Remote
--# own task remoteTask : Remote_task;
is

   --------------------------------
   -- Task that do the real job --
   --------------------------------
   task type Remote_task
   --# global in out MsgChannel.baseReader_remote_d;
   --#           out ControlDataChannel.remote_s1_s;
   --#           out ControlDataChannel.s1_par_input_d;
   --#           out MsgChannel.baseReader_remote_s;
   --# derives ControlDataChannel.remote_s1_s,
   --#         ControlDataChannel.s1_par_input_d,
   --#         MsgChannel.baseReader_remote_s,
   --#         MsgChannel.baseReader_remote_d     from &
   --#         null                               from MsgChannel.baseReader_remote_d;
   --# declare Suspends => (MsgChannel.baseReader_remote_d, ControlDataChannel.remote_s1_s);
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end Remote_task;

   ----------------------------
   -- Methods of the package --
   ----------------------------
   procedure decodeMessage(msg: in DataTypes.MESSAGE;
                           ctrl: out DataTypes.ControlData);
   --# derives ctrl from msg;

   ------------------------
   -- Task instantiation --
   ------------------------
   remoteTask: Remote_task;

end Remote;
```

Código Fonte B.31: Arquivo Remote.adb

```
with MsgChannel, Decoder, ControlDataChannel;
with Ada.Text_IO, Ada.Exceptions;

package body Remote is

   task body Remote_task is

   -- message received from the base station
   baseMsg: DataTypes.MESSAGE := DataTypes.defaultMessage;

   -- output data to control the vehicle
   ctrlData: DataTypes.ControlData := DataTypes.defaultControlData;

   begin

      loop

         -- receive data from the input channel
         MsgChannel.baseReader_remote_d.get(baseMsg);
         MsgChannel.baseReader_remote_s.proceed;

         -- convert the received message into a structure
         decodeMessage(baseMsg, ctrlData);
```

```
         -- send data through the output channel
         ControlDataChannel.s1_par_input_d.put(ctrlData);
         ControlDataChannel.remote_s1_s.stay;

      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in Remote");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end Remote_task;


   procedure decodeMessage(msg: in DataTypes.MESSAGE;
                           ctrl: out DataTypes.ControlData) is
      pragma Inline(decodeMessage);
   begin
      Decoder.convertToStructure(msg, ctrl);
   end decodeMessage;

end Remote;
```

Código Fonte B.32: Arquivo S1.ads

```
with Constants;
--# inherit Constants,
--#         DataTypes,
--#         ControlDataChannel;

package S1
--# own task s1Task : S1_task;
is

   ---------------------------------
   -- Task that do the real job --
   ---------------------------------
   task type S1_task
   --# global in out ControlDataChannel.s1_par_input_d;
   --#            out ControlDataChannel.remote_s1_s;
   --#            out ControlDataChannel.s1_move_d;
   --#            out ControlDataChannel.s1_move_s;
   --# derives ControlDataChannel.remote_s1_s,
   --#         ControlDataChannel.s1_move_d,
   --#         ControlDataChannel.s1_move_s,
   --#         ControlDataChannel.s1_par_input_d from &
   --#         null                              from ControlDataChannel.s1_par_input_d;
   --# declare Suspends => (ControlDataChannel.s1_par_input_d, ControlDataChannel.s1_move_s);
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end S1_task;

   ------------------------------
   -- Methods of the package --
   ------------------------------

   --------------------------
   -- Task instantiation --
   --------------------------
   s1Task: S1_task;

end S1;
```

Código Fonte B.33: Arquivo S1.adb

```
with DataTypes, ControlDataChannel;
with Ada.Text_IO, Ada.Exceptions;

package body S1 is

   task body S1_task is

   -- output data to control the vehicle
   ctrlData: DataTypes.ControlData := DataTypes.defaultControlData;

   begin

      loop

         -- receive data from the input channel
         ControlDataChannel.s1_par_input_d.get(ctrlData);
         ControlDataChannel.remote_s1_s.proceed;

         -- send data through the output channel
         ControlDataChannel.s1_move_d.put(ctrlData);
         ControlDataChannel.s1_move_s.stay;

      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in S1");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end S1_task;

end S1;
```

Código Fonte B.34: Arquivo S2.ads

```
with Constants;
--# inherit Constants,
--#         DataTypes,
```

```
--#           ControlDataChannel;

package S2
--# own task s2Task : S2_task;
is

   --------------------------------
   -- Task that do the real job --
   --------------------------------
   task type S2_task
   --# global in out ControlDataChannel.s2_par_input_d;
   --#              out ControlDataChannel.move_s2_s;
   --#              out ControlDataChannel.stop_s2_s;
   --#              out ControlDataChannel.s2_actuator_d;
   --#              out ControlDataChannel.s2_actuator_s;
   --# derives ControlDataChannel.s2_par_input_d,
   --#         ControlDataChannel.move_s2_s,
   --#         ControlDataChannel.stop_s2_s,
   --#         ControlDataChannel.s2_actuator_d,
   --#         ControlDataChannel.s2_actuator_s  from  &
   --#         null                              from ControlDataChannel.s2_par_input_d;
   --# declare Suspends => (ControlDataChannel.s2_par_input_d, ControlDataChannel.
       s2_actuator_s);
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end S2_task;

   -----------------------------
   -- Methods of the package --
   -----------------------------

   -------------------------
   -- Task instantiation --
   -------------------------
   s2Task: S2_task;

end S2;
```

Código Fonte B.35: Arquivo S2.adb

```
with DataTypes, ControlDataChannel;
with Ada.Text_IO, Ada.Exceptions;

package body S2 is

   task body S2_task is

   -- to store the transmitted data
   ctrlData: DataTypes.ControlData := DataTypes.defaultControlData;

   begin

      loop

         -- receive data from the input channel
         ControlDataChannel.s2_par_input_d.get(ctrlData);
         ControlDataChannel.move_s2_s.proceed;
         ControlDataChannel.stop_s2_s.proceed;

         -- send data through the output channel
         ControlDataChannel.s2_actuator_d.put(ctrlData);
         ControlDataChannel.s2_actuator_s.stay;

      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in S2");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end S2_task;

end S2;
```

Código Fonte B.36: Arquivo Sensor.ads

```
with DataTypes, Constants;
--# inherit DataTypes,
--#         Constants,
--#         BlackBoard,
--#         MsgChannel;

package Sensor
--# own task sensorTask : Sensor_task;
is

   --------------------------------
   -- Task that do the real job --
   --------------------------------
   task type Sensor_task
   --# global in       BlackBoard.blckBoard;
   --#              out MsgChannel.sensor_baseWriter_d;
   --#              out MsgChannel.sensor_baseWriter_s;
   --# derives MsgChannel.sensor_baseWriter_d,
   --#         MsgChannel.sensor_baseWriter_s from  &
   --#         null                           from BlackBoard.blckBoard;
   --# declare Suspends => MsgChannel.sensor_baseWriter_s;
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end Sensor_task;

   -----------------------------
   -- Methods of the package --
   -----------------------------
   procedure getSensors(sensors: out DataTypes.SensorData);
   --# global in BlackBoard.blckBoard;
   --# derives sensors from BlackBoard.blckBoard;

   procedure structureToMessage(struct: in DataTypes.SensorData;
```

```
                                     msg: out DataTypes.MESSAGE);
   --# derives msg from struct;

   -------------------------
   -- Task instantiation --
   -------------------------
   sensorTask: Sensor_task;

end Sensor;
```

Código Fonte B.37: Arquivo Sensor.adb

```
with Ada.Real_Time; use type Ada.Real_Time.Time;
with MsgChannel, BlackBoard, Encoder;
with Ada.Text_IO, Ada.Exceptions;

package body Sensor is

   task body Sensor_task is

      -- to store the sensor readings
      sensors: DataTypes.SensorData := DataTypes.defaultSensorData;

      -- message to send to the Base Station
      msgToSend: DataTypes.MESSAGE := DataTypes.defaultMessage;

      -- time attibutes (only for Periodic Processes)
      nextExecution: Ada.Real_Time.Time := Constants.SENSOR_START;
      PERIOD: constant Ada.Real_Time.Time_Span := Constants.SENSOR_PERIOD;

      -- time method (only for Periodic Processes)
      procedure getNextExecution is
      begin
         nextExecution := nextExecution + PERIOD;
      end getNextExecution;

   begin
      loop
         delay until nextExecution;

         -- get the direction from the compass
         getSensors(sensors);

         -- save the direction into the BlackBoard
         structureToMessage(sensors, msgToSend);

         -- send data over the output channel
         MsgChannel.sensor_baseWriter_d.put(msgToSend);
         MsgChannel.sensor_baseWriter_s.stay;

         getNextExecution;
      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in Sensor");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end Sensor_task;


   procedure getSensors(sensors: out DataTypes.SensorData) is
      pragma Inline(getSensors);
   begin
      BlackBoard.blckBoard.getSensorReadings(sensors);
   end getSensors;


   procedure structureToMessage(struct: in DataTypes.SensorData;
                                     msg: out DataTypes.MESSAGE) is
      pragma Inline(structureToMessage);
   begin
      Encoder.sensorToMessage(struct, msg);
   end structureToMessage;

end Sensor;
```

Código Fonte B.38: Arquivo Shadows.ads

```
package Ada.Streams is
end Ada.Streams;

package GNAT is
end GNAT;

package GNAT.Sockets is
end GNAT.Sockets;
```

Código Fonte B.39: Arquivo Sonar.ads

```
with DataTypes, Constants;
--# inherit DataTypes,
--#         Constants,
--#         BlackBoard;

package Sonar
--# own task sonarTask: Sonar_task;
is

   ---------------------------------
   -- Task that do the real job --
   ---------------------------------
   task type Sonar_task
   --# global out BlackBoard.blckBoard;
   --# derives BlackBoard.blckBoard from ;
```

```
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end Sonar_task;

   ------------------------------
   -- Methods of the package --
   ------------------------------
   procedure getSonar(sonars: out DataTypes.SONAR_TYPE);
   --# derives sonars from ;

   procedure saveSonar(sonars: in DataTypes.SONAR_TYPE);
   --# global out BlackBoard.blckBoard;
   --# derives BlackBoard.blckBoard from sonars;

   --------------------------
   -- Task instantiation --
   --------------------------
   sonarTask: Sonar_task;

end Sonar;
```

Código Fonte B.40: Arquivo Sonar.adb

```
with Ada.Real_Time; use type Ada.Real_Time.Time;
with BlackBoard;
with Ada.Text_IO, Ada.Exceptions;

package body Sonar is

   task body Sonar_task is

      -- stores the sonar readings
      sonar: DataTypes.SONAR_TYPE;

      -- time attibutes (only for Periodic Processes)
      nextExecution: Ada.Real_Time.Time := Constants.SONAR_START;
      PERIOD: constant Ada.Real_Time.Time_Span := Constants.SONAR_PERIOD;

      -- time method (only for Periodic Processes)
      procedure getNextExecution is
      begin
         nextExecution := nextExecution + PERIOD;
      end getNextExecution;

   begin
      loop
         delay until nextExecution;

         -- get the sonar readings
         getSonar(sonar);

         -- save the sonar readings into the BlackBoard
         saveSonar(sonar);

         getNextExecution;
      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in Sonar");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end Sonar_task;


   procedure getSonar(sonars: out DataTypes.SONAR_TYPE) is
      pragma Inline(getSonar);
   begin
      sonars(1) := 0.5;
      sonars(2) := 0.5;
      sonars(3) := 0.5;
      sonars(4) := 0.5;
   end getSonar;


   procedure saveSonar(sonars: in DataTypes.SONAR_TYPE) is
      pragma Inline(saveSonar);
   begin
      BlackBoard.blckBoard.setSonar(sonars);
   end saveSonar;

end Sonar;
```

Código Fonte B.41: Arquivo StatusChannel.ads

```
with Constants, DataTypes;
--# inherit Constants,
--#         DataTypes;

-- A channel has two protected objects: Data and Sync. The Data protected
-- object is the data communicated through the channel. Sync is a synchronizer.
package StatusChannel
--# own protected avoid_move_d : Data (priority => Constants.CHANNEL_PRIORITY,
--#                                    suspendable);
--#     protected avoid_move_s : Sync (priority => Constants.CHANNEL_PRIORITY,
--#                                    suspendable);
is

   protected type Data is

      pragma Priority(Constants.CHANNEL_PRIORITY);

      entry get(someData : out DataTypes.STATUS);
      --# global in out Data;
      --# derives Data,
      --#         someData from Data;

      procedure put(someData: in DataTypes.STATUS);
      --# global in out Data;
```

```
      --# derives Data from *,
      --#                   someData;

   private
      chData : DataTypes.STATUS := DataTypes.defaultStatus;
      readyToRead : Boolean := False; -- Initially, there is no data to read
   end Data;


   protected type Sync is

      pragma Priority(Constants.CHANNEL_PRIORITY);

      entry stay;
      --# global in out Sync;
      --# derives Sync from *;

      procedure proceed;
      --# global in out Sync;
      --# derives Sync from *;

   private
      hasRead : Boolean := False;
   end Sync;

   ----------------------------
   -- Channel Instantiations --
   ----------------------------
   avoid_move_d: Data;
   avoid_move_s: Sync;

end StatusChannel;
```

Código Fonte B.42: Arquivo StatusChannel.adb

```
package body StatusChannel is

   protected body Data is
      entry get(someData : out DataTypes.STATUS) when readyToRead
      --# global in      chData;
      --#           out readyToRead;
      --# derives readyToRead from &
      --#         someData    from chData;
      is
      begin
         -- Read the encapsulated data
         someData := chData;
         -- Block successive readings
         readyToRead := False;
      end get;

      procedure put(someData : in DataTypes.STATUS)
      --# global out readyToRead;
      --#        out chData;
      --# derives readyToRead from &
      --#         chData      from someData;
      is
      begin
         -- Update the encapsulated data
         chData := someData;
         -- Permit for the next writing operation
         readyToRead := True;
      end put;
   end Data;

   protected body Sync is
      entry stay when hasRead
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Stay for the data to be read
         hasRead := False;
      end stay;

      procedure proceed
      --# global out hasRead;
      --# derives hasRead from ;
      is
      begin
         -- Declare the completion of a reading operation
         hasRead := True;
      end proceed;
   end Sync;

end StatusChannel;
```

Código Fonte B.43: Arquivo Stop.ads

```
with DataTypes, Constants;
--# inherit DataTypes,
--#         Constants,
--#         BlackBoard;

package Sonar
--# own task sonarTask: Sonar_task;
is

   -------------------------------
   -- Task that do the real job --
   -------------------------------
   task type Sonar_task
   --# global out BlackBoard.blckBoard;
   --# derives BlackBoard.blckBoard from ;
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end Sonar_task;

   ----------------------------
   -- Methods of the package --
```

```
-----------------------------
procedure getSonar(sonars: out DataTypes.SONAR_TYPE);
--# derives sonars from ;

procedure saveSonar(sonars: in DataTypes.SONAR_TYPE);
--# global out BlackBoard.blckBoard;
--# derives BlackBoard.blckBoard from sonars;

-------------------------
-- Task instantiation --
-------------------------
sonarTask: Sonar_task;

end Sonar;
```

Código Fonte B.44: Arquivo Stop.adb

```
with DataTypes, BlackBoard, ControlDataChannel;
with Ada.Text_IO, Ada.Exceptions;

package body Stop is

   task body Stop_task is

      -- time of the last received message
      lastReception: Ada.Real_Time.Time := Ada.Real_Time.Clock;

      -- data with null speeds
      stpData: constant DataTypes.ControlData := DataTypes.defaultControlData;

      -- timeout that caracterizes lost of communication with the Base Station
      TIMEOUT: constant Ada.Real_Time.Time_Span :=
        Ada.Real_Time.Milliseconds(1000);

      -- time attibutes (only for Periodic Processes)
      nextExecution: Ada.Real_Time.Time := Constants.COMPASS_START;
      PERIOD: constant Ada.Real_Time.Time_Span := Constants.COMPASS_PERIOD;

      -- time method (only for Periodic Processes)
      procedure getNextExecution is
      begin
         nextExecution := nextExecution + PERIOD;
      end getNextExecution;

   begin
      loop
         delay until nextExecution;

         -- check the time of the last message reception
         checkLastReception(lastReception);

         if Ada.Real_Time.">=" ((Ada.Real_Time.Clock - lastReception), TIMEOUT)
         then
            ControlDataChannel.s2_par_input_d.put(stpData);
            ControlDataChannel.stop_s2_s.stay;
         end if;

         getNextExecution;
      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in Stop");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end Stop_task;


   procedure checkLastReception(lastReception: out Ada.Real_Time.Time) is
   begin
      BlackBoard.blckBoard.getReceptionTime(lastReception);
   end checkLastReception;

end Stop;
```

Código Fonte B.45: Arquivo ThrusterWriter.ads

```
with DataTypes, Constants;
--# inherit DataTypes,
--#         Constants,
--#         Encoder,
--#         ActuationDataChannel,
--#         ControlDataChannel;

package ThrusterWriter
--# own task thrusterWriterTask : ThrusterWriter_task;
is

   ------------------------------
   -- Task that do the real job --
   ------------------------------
   task type ThrusterWriter_task
   --# global in out ActuationDataChannel.actuator_thruster_d;
   --#           out ActuationDataChannel.actuator_thruster_s;
   --# derives ActuationDataChannel.actuator_thruster_d,
   --#         ActuationDataChannel.actuator_thruster_s from &
   --#         null                                     from ActuationDataChannel.
      actuator_thruster_d;
   --# declare Suspends => ActuationDataChannel.actuator_thruster_d;
   is
      pragma Priority(Constants.TASK_PRIORITY);
   end ThrusterWriter_task;

   ---------------------------
   -- Methods of the package --
   ---------------------------
   procedure sendMessage(msg: in DataTypes.MESSAGE);
   --# derives null from msg;
```

```
   procedure generateMessage(struct: in DataTypes.ActuationData;
                             msg: out DataTypes.MESSAGE);
   --# derives msg from struct;

   ------------------------
   -- Task instantiation --
   ------------------------
   thrusterWriterTask: ThrusterWriter_task;

end ThrusterWriter;
```

Código Fonte B.46: Arquivo ThrusterWriter.adb

```
with ActuationDataChannel, Encoder;
with Ada.Text_IO, Ada.Exceptions;

package body ThrusterWriter is

   task body ThrusterWriter_task is

      -- store the setpoints to the thrusters
      setpoints: DataTypes.ActuationData := DataTypes.defaultActuationData;

      -- message to the controller
      controllerMsg: DataTypes.MESSAGE := DataTypes.defaultMessage;

   begin

      loop

         -- receive data from the input channel
         ActuationDataChannel.actuator_thruster_d.get(setpoints);
         ActuationDataChannel.actuator_thruster_s.proceed;

         -- generate the message to the controller
         generateMessage(setpoints, controllerMsg);

         -- send the message to the Thruster Station
         sendMessage(controllerMsg);

      end loop;

   exception
      when Error : others =>
         Ada.Text_IO.Put_Line("Error in ThrusterWriter");
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Name(Error));
         Ada.Text_IO.Put_Line(Ada.Exceptions.Exception_Message(Error));

   end ThrusterWriter_task;

   -- simulate the sending of a message over the CAN network
   procedure sendMessage(msg: in DataTypes.MESSAGE) is
      pragma Inline(sendMessage);
   begin
      -- print the message on the screen
      Ada.Text_IO.Put_Line(msg);
   end sendMessage;

   procedure generateMessage(struct: in DataTypes.ActuationData;
                             msg: out DataTypes.MESSAGE) is
      pragma Inline(generateMessage);
   begin
      Encoder.actuationToMessage(struct, msg);
   end generateMessage;

end ThrusterWriter;
```

Código Fonte B.47: Arquivo UDP_Sockets.ads

```
with DataTypes;
with GNAT.Sockets;
with Ada.Streams;
--# inherit DataTypes,
--#         Ada.Streams,
--#         GNAT.Sockets;

-----------------------------------------------------------------------------
-- This package contains all the code necessary to send and receive UDP. It --
--  was develop to use with two sockets: one to send and other to receive   --
-- the messages, both in different tasks.                                   --
-----------------------------------------------------------------------------
package UDP_Sockets is

   type UDP_Socket is private;

   procedure Create_Receiver_Socket(this: in out UDP_Socket;
                                    RECEPTION_PORT: in Integer);
   --# derives this from *,
   --#                   RECEPTION_PORT;

   procedure Create_Sender_Socket(this: in out UDP_Socket;
                                  DESTINATION_IP: in DataTypes.IP_Address;
                                  DESTINATION_PORT: in Integer);
   --# derives this from *,
   --#                   DESTINATION_PORT,
   --#                   DESTINATION_IP;

   procedure Receive_Message(this: in out UDP_Socket;
                             msg: out DataTypes.MESSAGE);
   --# derives this from &
   --#         msg from this;

   procedure Send_Message(this: in out UDP_Socket; msg: in DataTypes.MESSAGE);
   --# derives this from *,
   --#                   msg;

   procedure Close_Socket(this: in out UDP_Socket);
   --# derives this from *;

private
```

```
   --# hide UDP_Sockets;

   function To_String (S : Ada.Streams.Stream_Element_Array)
                       return DataTypes.MESSAGE;

   function Get_Buffer_Data(this: UDP_Socket)
                            return Ada.Streams.Stream_Element_Array;

   -- Size of the data buffer, the same of the received message sizes
   BUFFER_SIZE  : Ada.Streams.Stream_Element_Offset :=
     Ada.Streams.Stream_Element_Offset(DataTypes.msgSize);

   type UDP_Socket is
      record
         Address         : GNAT.Sockets.Sock_Addr_Type;
         Socket          : GNAT.Sockets.Socket_Type;
         DataBuffer      : Ada.Streams.Stream_Element_Array (1 .. BUFFER_SIZE);
         Last            : Ada.Streams.Stream_Element_Offset;
      end record;

end UDP_Sockets;
```

Código Fonte B.48: Arquivo UDP_Sockets.adb

```
package body UDP_Sockets is
   --# hide UDP_Sockets;

   use GNAT.Sockets, Ada.Streams;

   procedure Create_Receiver_Socket(this: in out UDP_Socket;
                                    RECEPTION_PORT: in Integer)
   is
      pragma Inline(Create_Receiver_Socket);
   begin
      -- Configure the Socket
      Create_Socket (this.Socket, Family_Inet, Socket_Datagram);

      this.Address.Port := GNAT.Sockets.Port_Type(RECEPTION_PORT);

      -- Associates the socket with the reception Address
      Bind_Socket (this.Socket, this.Address);
   end create_Receiver_Socket;


   procedure Create_Sender_Socket(this: in out UDP_Socket;
                                  DESTINATION_IP: in DataTypes.IP_Address;
                                  DESTINATION_PORT: in Integer)
   is
      pragma Inline(Create_Sender_Socket);
   begin
      -- Create the Socket
      Create_Socket (this.Socket, Family_Inet, Socket_Datagram);

      -- Configure the destination Address
      this.Address.Addr := Inet_Addr (DESTINATION_IP);
      this.Address.Port := GNAT.Sockets.Port_Type(DESTINATION_PORT);

      -- Associates a default address to which messages should be sent
      Connect_Socket (this.Socket, this.Address);
   end create_Sender_Socket;


   procedure Receive_Message(this: in out UDP_Socket;
                             msg: out DataTypes.MESSAGE) is
      pragma Inline(Receive_Message);
   begin
      -- Receive data over the socket
      Receive_Socket(this.Socket, this.DataBuffer, this.Last);

      -- extract the message from the buffer data as a String
      msg := To_String(Get_Buffer_Data(this));
   end Receive_Message;


   procedure Send_Message(this: in out UDP_Socket;
                          msg: in DataTypes.MESSAGE) is
      pragma Inline(Send_Message);
   begin
      -- insert the msg into the DataBuffer
      for i in msg'Range loop
         this.DataBuffer(Ada.Streams.Stream_Element_Offset(i)) :=
           Stream_Element'Val (Character'Pos (msg (i)));
         end loop;

      -- Send data over the socket
      Send_Socket (this.Socket, this.DataBuffer, this.Last);
   end Send_Message;


   procedure Close_Socket(this: in out UDP_Socket) is
      pragma Inline(Close_Socket);
   begin
      -- Finalize the socket
      Close_Socket (this.Socket);
   end Close_Socket;


   function Get_Buffer_Data(this: UDP_Socket) return Stream_Element_Array is
      pragma Inline(Get_Buffer_Data);
   begin
      return this.DataBuffer;
   end Get_Buffer_Data;


   function To_String (S : Stream_Element_Array) return DataTypes.MESSAGE is
      pragma Inline(To_String);
      Result : DataTypes.MESSAGE;
   begin
      for I in Result'Range loop
         Result(I) :=
           Character'Val (Stream_Element'Pos
                          (S (Stream_Element_Offset(I) + S'First - 1)));
      end loop;
```

```
      return Result;
   end To_String;

begin
   GNAT.Sockets.Initialize;       -- required by the library

end UDP_Sockets;
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo