UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

NVL: UMA LINGUAGEM DE COORDENAÇÃO PARA REDES DE VÉICULOS AUTÓNOMOS

Manuel António Ribeiro

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2014

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

NVL: UMA LINGUAGEM DE COORDENAÇÃO PARA REDES DE VÉICULOS AUTÓNOMOS

Manuel António Ribeiro

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada pelo Prof. Doutor Eduardo Resende Brandão Marques
e co-orientada pelo Prof. Doutor Francisco Cipriano da Cunha Martins

2014

Agradecimentos

Gostaria de agradecer aos meus pais, por todo o suporte e força dada durante todo o meu percurso académico.

Ao meu orientador Eduardo Resende Brandão Marques e co-orientador Francisco Cipriano da Cunha Martins pela disponibilidade e ajuda prestada durante todo o desenvolvimento deste projeto.

A todo o grupo do LSTS, que sempre se mostrou disponível para ajudar com problemas técnicos e disponibilizou material sem o qual não seria possível realizar este projeto, em especial ao Eng. João Tasso e ao José Pinto.

Aos colegas da Faculdade de Ciências da Universidade de Lisboa pela amizade, estímulo e suas contribuições ao longo de todo o percurso académico.

E finalmente a todos aqueles que de uma forma ou de outra deram a sua contribuição para a realização deste projeto.

Resumo

O uso de veículos autónomos não tripulados tem-se vindo a massificar recentemente com as as mais diversas aplicações. Em particular, muitos cenários podem beneficiar do uso cooperativo de vários veículos em rede, tornando-se então imprescindível a concepção de sistemas de software apropriados para o efeito.

Frequentemente, a condução de operações em cenários multi-veículo ainda resulta de especificações separadas por veículo, que se compõem depois em tempo de execução de forma relativamente *ad-hoc*, e requerendo um elevado grau de intervenção humana. Em contraste, propomos a escrita de "programas de rede" que possam definir o comportamento global de multiplos veículos autónomos, com uma semântica bem fundada e um alto grau de automação.

Tendo esta proposta geral em mente, foi desenhada e implementada uma linguagem de coordenação para redes de veículos autónomos, a *Networked Vehicles' Language* (NVL). Um programa NVL define a coordenação global de veículos selecionados numa rede e a associação de tarefas a veículos ou grupos de veículos, com expressividade para padrões de temporização, sequência, concorrência, e escolha no fluxo de um programa. Em suporte ao corpo principal de um programa, as tarefas são definidas por controladores multi-veículo cujo código pode ser sintetizado automaticamente.

O desenvolvimento da NVL decorreu em colaboração com o Laboratório de Sistemas e Tecnologia Subaquática (LSTS) da Faculdade de Engenharia da Universidade do Porto, que há mais de uma década desenvolve de raiz veículos autónomos e um *toolchain* de software associado. A NVL foi desenhada e implementada por forma a ser compatível com o *toolchain* do LSTS, e foi avaliada em simulação e ainda experiências de campo com veículos reais do LSTS.

Palavras-chave: veículos autónomos, linguagem de programação, coordenação distribuída

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

Nos últimos anos tem-se massificado o desenvolvimento e uso de veículos autónomos não-tripulados para as mais diversas aplicações.

A característica primária de um veículo é antes de mais, estar equipado com sensores e actuadores base, que, junto com software de controlo, permitem a sua locomoção num meio físico (ex., aéreo, terrestre, aquático). Um veículo pode de resto ter variados níveis de automação/autonomia, desde a necessidade de controlo direto por um operador humano, até à capacidade de operar várias horas sem essa intervenção. O veículo pode também ter apenas a capacidade de executar um conjunto de tarefas prescritas, ou uma capacidade de deliberação/reatividade na execução de tarefas.

Várias aplicações beneficiam do uso cooperativo de vários veículos num ambiente de rede [24, 9, 23]. No entanto, existem diversos desafios à especificação do comportamento conjunto de múltiplos veículos. É complexa a especificação de um comportamento global para o conjunto de veículos, tendo em conta uma topologia dinâmica de rede (ex., comunicação intermitente com perda total ou parcial de dados), necessidade de sincronização temporal e espacial, possível heterogeneidade do tipo de veículos inválidos (ex., a colaboração de veículos aéreos e aquáticos [20]), e interação com operadores humanos.

Este quadro geral tem sido considerado pelo Laboratório de Sistemas e Tecnologias Subaquáticas (LSTS), da Faculdade de Engenharia do Porto, que na última década desenvolveu de raiz veículos autónomos de vários tipos e software associado para a operação de veículos em rede, em parcerias com a Força Aérea, Marinha Portuguesa, entre outras entidades. Esta tecnologia tem sido aplicada em vários cenários de aplicação real, tendo-se tornado rotineiro o uso de múltiplos veículos nos últimos anos [7, 20, 10, 13].

Um dos problemas colocados nestes cenários é a necessidade de uma preparação laboriosa dos cenários de operação, por forma a que o comportamento "composto" de vários veículos atinja os objetivos em causa. Tipicamente, os veículos são programados separa-

damente para esse efeito, residindo o "plano global" na mente de operadores humanos. Em colaboração com o LSTS, surgiu a questão de como especificar o comportamento coordenado/cooperativo entre veículos, definindo "programas de rede", com especificação e semântica bem fundadas, por forma a especificar o comportamento global de vários veículos num cenário de interesse.

1.2 Contribuição

Esta tese apresenta uma linguagem de coordenação para redes de veículos autónomos, a *Networked Vehicles' Language* (NVL). Salientamos de seguida os aspectos essenciais de contribuição.

A linguagem tem como aspectos base a seleção dinâmica de veículos e alocação destes a controladores multi-veículo. As primitivas da linguagem permitem compor a execução de controladores de acordo com requisitos de temporização, precedência, concorrência e escolha no fluxo do programa. Estes aspectos vão de encontro a características que consideramos fundamentais na operação de cenários multi-veículo.

O software desenvolvido compreende ambientes de edição e validação prévia de programas NVL, bem como da sua execução distribuída na forma de um interpretador para a linguagem, e módulos supervisores que correm localmente no veículo. O software foi desenhado e implementado por forma a integrar-se com o *toolchain* de software do LSTS [19].

Finalmente, salientamos a validação da linguagem em cenários multi-veículo em ambiente de simulação e experiências de campo com veículos reais.

1.3 Estrutura da tese

O restante da tese está estruturada da seguinte forma:

- O Capítulo 2 ("Trabalho relacionado") contextualiza o trabalho no âmbito dos desenvolvimentos de veículos autónomos do LSTS e no estado-da-arte em geral.
- O Capítulo 3 ("A Linguagem NVL") apresenta a linguagem NVL. Usando um cenário exemplo, são ilustradas as características nucleares de programas NVL e as correspondentes primitivas de programação.
- O Capítulo 4 ("Desenho e Arquitetura") descreve o desenho e arquitectura dos ambiente de desenvolvimento e execução, em suporte a programas NVL.
- O Capítulo 5 ("Implementação") apresenta os traços essenciais da implementação dos vários componentes de software desenvolvidos.

- O Capítulo 6 (“Resultados experimentais”) apresenta resultados da execução de programas NVL, em ambiente de simulação e em experiências de campo realizadas com veículos reais.
- O Capítulo 7 (“Conclusão”) faz uma discussão final das contribuições e identifica itens de trabalho futuro.

Capítulo 2

Trabalho relacionado

2.1 Trabalho no LSTS

Ao longo da última década, o LSTS tem desenvolvido vários veículos autónomos e ferramentas de software de raiz para a sua operação. Os desenvolvimentos compreendem diversos tipos de veículos autónomos aéreos e aquáticos, bóias de recolha de dados oceanográficos, ou *gateways* de comunicação para mediação a meios heterogéneos de comunicação (ex., *modems* acústicos, wifi, Iridium). Como suporte a esta infraestrutura, um *toolchain* de software, agora *open-source* (`http://github.com/LSTS`), toma também papel de destaque. A Figura 2.1 dá uma ideia do tipo de interações consideradas nas aplicações alvos para os veículos do LSTS. Nesta secção fazemos uma descrição destes desenvolvimentos e aplicações alvo.

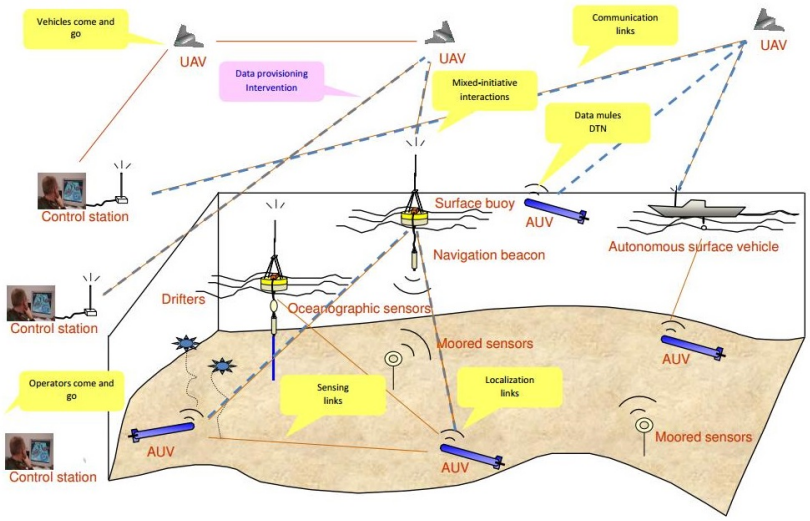


Figura 2.1: Rede de veículos LSTS

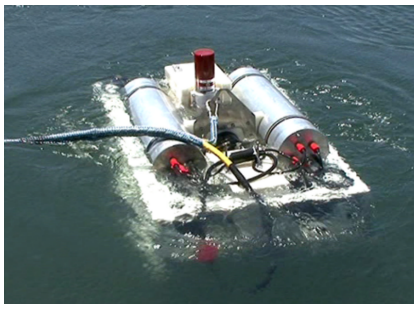
2.1.1 Veículos autónomos

(a) LAUV Noptilus

(b) Xtreme-2 e Manta

(c) UAV X8

(d) ASV Swordfish



(e) ROV KOS

Figura 2.2: Véiculos desenvolvidos no LSTS

O LSTS desenvolveu diferentes veículos com diferentes objetivos:

- AUVs ("Autonomous Underwater Vehicles") como o Noptilus Light AUV (Figura 2.2a) ou o LAUV Xtreme (Figura 2.2b) são veículos autónomos submarinos tipicamente usados em missões de recolha de dados oceanográficos, como batimetria ou inspeção do fundo oceânico. Este tipo de veículos tem a capacidade de operar sem intervenção humana ou conectividade durante largos períodos de tempo.

- UAVs ("Unmanned Aerial Vehicles") são veículos aéreos autónomos como o X8 (Figura 2.2c) com maior alcance a nível de comunicação. Podem ser empregues em diversas aplicações, em particular como "mulas" de dados para outro veículo e como *gateway* móvel.

- ASVs ("Autonomous surface Vehicle") como o Swordfish (Figura 2.2d) são veículos autónomos que se mantêm à superfície da água. Têm sido empregues como *gateways* de comunicação móvel pelo LSTS.
- ROVs como o ROV KOS (Figura 2.2e) são veículos controlados remotamente por um operador, fisicamente ligados a um cabo de comunicação.

2.1.2 Arquitetura de Controlo

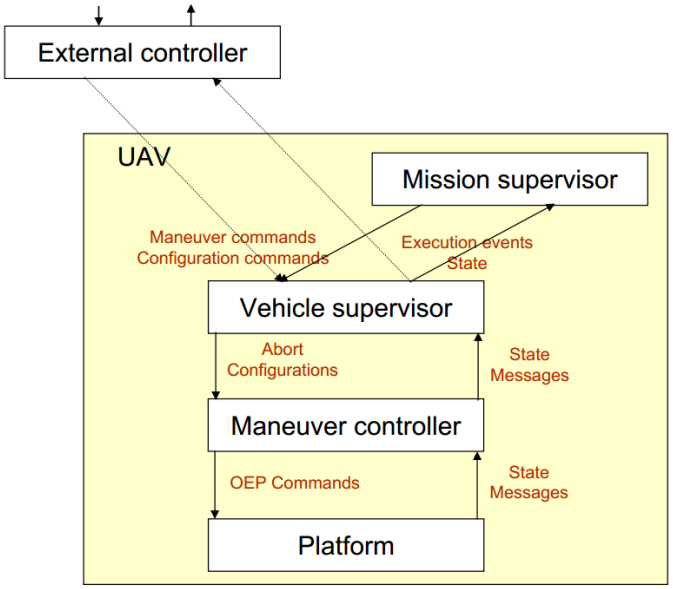


Figura 2.3: Arquitetura de Controlo

A arquitetura de controlo dos veículos é baseada num modelo por camadas, onde cada camada encapsula detalhes de baixo-nível, ao mesmo tempo que fornece interfaces para receção do estado da camada imediatamente a baixo e permite também o envio de comandos [18]. Existem quatro camadas, ilustradas na Figura 2.3 (retirada de [25]), formadas por controladores baixo-nível, controladores de manobra, supervisor do veículo e supervisor de missão:

- Os controladores de baixo-nível abstraem numa interface modular todas as interações dos sensores e atuadores do veículo, para o efeito de locomoção e cálculo do posicionamento do veículo.
- Os controladores de manobra disponibilizam primitivas para controlo do veículo de uma forma abstrata e percetível por um operador, por exemplo mover um veículo para um determinado ponto no espaço (manobra conhecida como "Goto") ou orbitar o veículo à volta de determinada área ("Loiter").

- O supervisor do veículo tem por função monitorizar o estado do veículo, habilitando e desabilitando interações em função deste na forma de manobras ou outros comandos do supervisor de missão e/ou controladores externos. O seu funcionamento, ilustrado na Figura 2.4 (retirada de [25]), pode ser entendido como uma máquina de estados em que o veículo está numa das de várias configurações que habilitam ou não a execução de manobras.

- Um supervisor de missão executa determinado plano de ação, cujas primitivas elementares são manobras ou outros comandos para o veículo, operando a bordo do veículo ou remotamente na rede. Para aferir sobre o estado do veículo e reagir em conformidade para a execução de um plano, o supervisor de missão usa a abstração de estado fornecida por um supervisor de veículo.

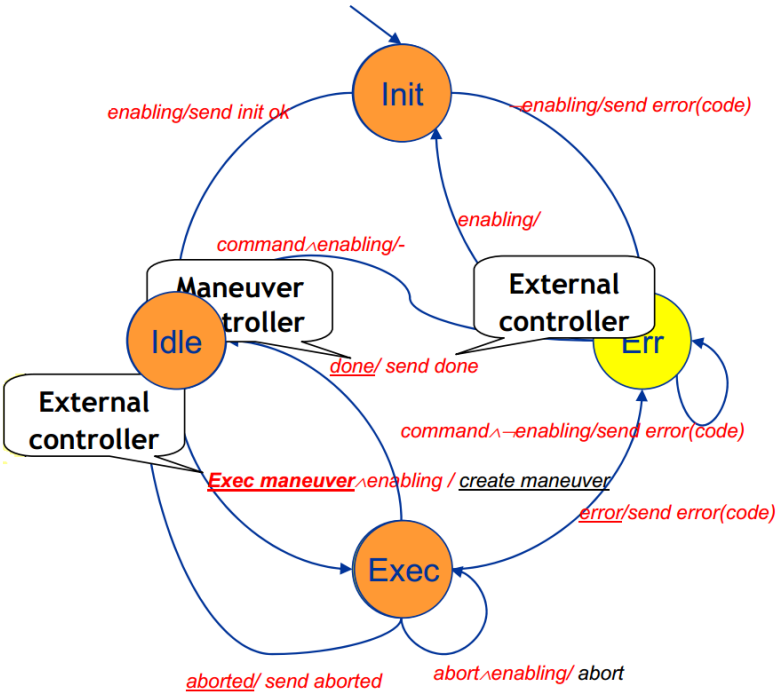


Figura 2.4: Supervisor do Veículo

2.1.3 Ferramentas de software

2.1.3.1 IMC

O IMC (Inter-Module Communication) é um protocolo de comunicação para redes de veículos e sensores baseado em mensagens [18, 14] O protocolo tem como objetivo a interoperabilidade dos vários componentes de software através da troca de mensagens, em um âmbito de rede ou interno a um veículo, abstraindo a heterogeneidade de componentes

de hardware ou softwaere que estejam envolvidos. Para tal, o IMC define um conjunto de mensagens e um formato de serialização neutro sendo o protocolo especificado num formato XML, a partir do qual é gerado código que possibilita o uso do IMC em C++ ou Java.

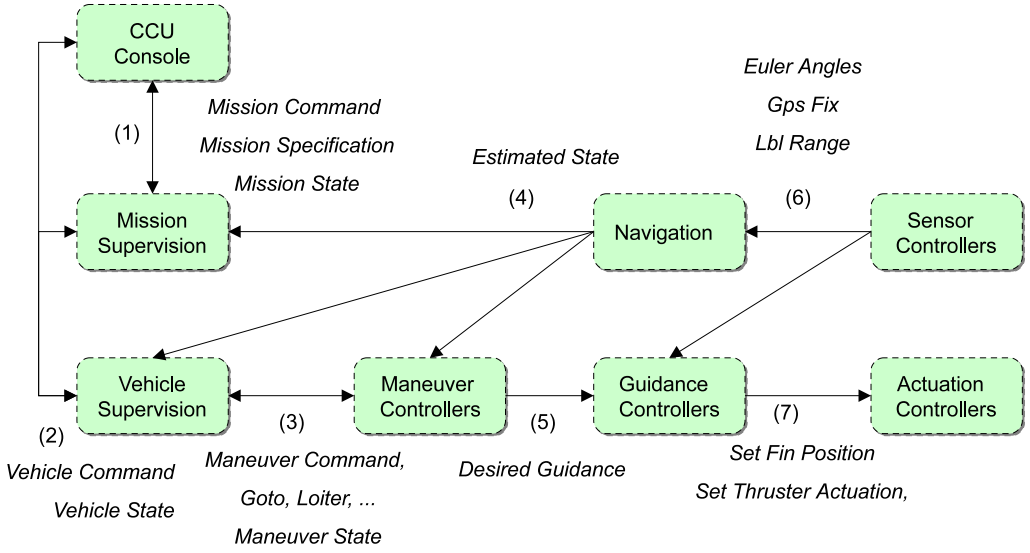


Figura 2.5: Exemplo do fluxo de mensagens IMC

O IMC é modular na medida em que as mensagens são agrupadas de acordo com o seu propósito, em particular de acordo com a camada de controlo a que se referem, como ilustrado na Figura 2.5 (retirada de [14]):

- Mensagens de controlo de missão — especificam a missão e o seu ciclo de vida, servem de interface entre o um módulo externo (como o Neptus, discutido abaixo) e o módulo de supervisão de missão;
- Mensagens de controlo de veículo — usadas para interagir com o veículo a partir de uma fonte externa ou o módulo de supervisão de missão para mandar executar comandos/responder a pedidos;
- Mensagens de manobras — usadas para definir manobras e estados associados a essa manobra;
- Mensagens de "guidance" — mensagens de controlo de baixo nível, por exemplo para controlo da direção ou velocidade de um veículo;
- Mensagens de sensores — usadas para transmitir os dados dos sensores
- Mensagens dos atuadores — especificam a interação sobre os atuadores que fazem mover o veículo;

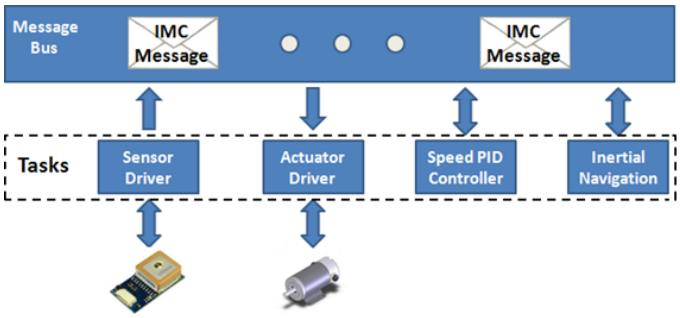


Figura 2.6: Conceito de Troca de Mensagens

2.1.3.2 DUNE

O DUNE (DUNE Uniform Navigational Environment) é o sistema de software de bordo, usado em veículos autónomos, bem como gateways e bóias oceanográficas [18]. O DUNE é escrito em C++ e pode ser usado em diferentes sistemas operativos (ex. Linux, Windows, SunOS) para a programação de tarefas em tempo real. Uma tarefa DUNE é escrita modularmente e corre de forma isolada das outras, trocando mensagens IMC com outras tarefas através de um *bus* de mensagens, como ilustrado na Figura 2.6. No contexto de um veículo autónomo, existem tarefas distintas por cada sensor, atuador, tipo de controlo, bem como para a execução de manobras, supervisão do veículo, e supervisão de plano.

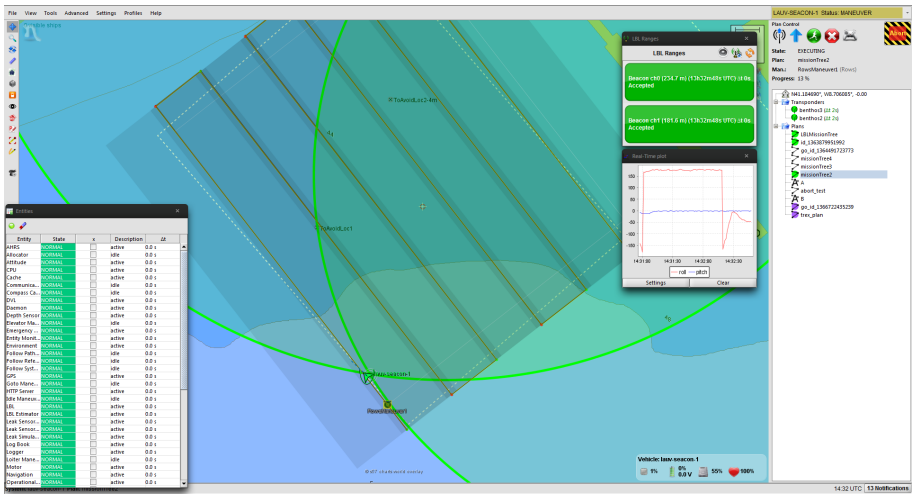


Figura 2.7: Consola Neptus

2.1.3.3 Neptus

O Neptus é uma infraestrutura de comando e controlo, permitindo a operadores humanos conduzir diversas fases de operação de um veículo autónomo como planeamento,

execução de planos, e revisão de resultados. Para tal dispõe de vários tipos de consolas gráficas, sendo exemplificada uma na Figura 2.7. Detalhamos algumas das funcionalidades do Neptus em suporte à NVL em capítulos posteriores.

2.1.4 Cenários de operação multi-veículo

Mencionamos aqui alguns cenários multi-veículo que foram ou testados em veículos concretos, simulados, ou objeto de modelação por parte do LSTS.

Cada uma destas aplicações requer a presença contínua numa área vasta e a capacidade de dar uma resposta ágil em função do meio ambiente. Esta resposta pode ir desde uma simples visita feita por um veículo munido de vários sensores, transmissão de dados, reconfiguração de dispositivo ou até reposicionamento físico na rede.

2.1.4.1 Varrimento de área

Neste cenário, são utilizados vários veículos autónomos (AUVs) para fazer um varrimento do oceano numa determinada área de interesse. Esta área poder ser atribuída manualmente a cada veículo, necessitando de supervisão e gestão por parte de um operador humano [15], ou feita de um modo automatizado, no qual existe um processo de seleção dos veículos disponíveis na rede e lhes é atribuída uma área disponível (não atribuída a nenhum veículo previamente) e a introdução de um outro veículo UAV para a recolha de dados dos AUVs [26].

2.1.4.2 Data Muling

Um cenário de *data muling* tem como foco a recolha, o transporte e a disseminação de dados recorrendo a veículos autónomos [5]. Tipicamente, alguns veículos agem como produtores de informação e outros como ”mulas”, i.e., transferem dados de veículos produtores e posteriormente tratam da sua disseminação [12]. Da colecta à entrega de dados recolhidos, têm de ser considerados vários factores limitativos que decorrem da infraestrutura de rede disponível, como a possível perda de dados ou conectividade intermitente com topologia dinâmica. Um exemplo básico de mula de dados é uma missão na qual um dos veículos está a desempenhar determinada missão de recolha de dados e num período predefinido é selecionado um veículo disponível na rede que que se deslocará até a um ponto comum aos dois veículos (uma localização estrategicamente predefinida) com o objetivo de adquirir todos os dados que o primeiro veículo recolheu.

2.1.4.3 Monitorização de peixes-lua

O LSTS fez uso dos seus veículos e recorrendo ao seu conjunto de ferramentas realizou uma experiência na tentativa de acompanhar os peixes-lua [20] e adquirir alguns dados relevantes sobre os seus comportamentos. Para tal foram necessários três veículos, um AUV, um UAV e um ASV e um peixe lua etiquetado. Fazendo uso da etiqueta de marcação instalada no peixe-lua, obtém-se, via satélite, a localização do peixe que permite ao Neptus gerar um objetivo de alto nível a ser enviado para o o veículo, que se encarrega de gerar um padrão de patrulhamento a ser realizado em torno do peixe etiquetado. O UAV obtém imagem aérea da zona em questão. O ASV aguarda que periodicamente o AUV se desloque para a superfície e obtém os seus dados que serão por fim enviados para o UAV, que funciona como mula de dados.

2.2 Estado da Arte

A seguir fazemos referência a trabalho relacionado com relevância à NVL.

2.2.1 Modelo *fork/join*

Algumas linguagens de programação têm primitivas built-in que habilitam um modelo de concorrência *fork*/join para tarefas, como a X10 [21], ou o Habanero-Java [4]. Tipicamente são providas construções para iniciar tarefas concorrentemente (*fork*) e para as sincronizar (*join*), Ilustramos o modelo *fork/join*) com um fragmento da linguagem X10 na Figura 2.8, para o cálculo da sequência de Fibonacci. Basicamente, são criando duas tarefas f1 e f2 que executam em paralelo, mas cuja conclusão é sincronizada, antes de os seus resultados finais poderem ser usados. Na NVL definimos primitivas com espírito semelhante, mas com requisitos temporais extra associados.

```
def run() {
  if (r < 2) return;
  val f1 = new Fib(r-1),
  val f2 = new Fib(r-2);
  finish {
    async f1.run();
    async f2.run();
  }
  r = f1.r + f2.r;
}
```

Figura 2.8: Fragmento X10 ilustrando modelo *fork*/join

2.2.2 CSL - Especificação de comportamentos em MSN

A CSL [11] é uma linguagem de alto nível para controlo de feedback numa rede de sensores móveis (MSN) de modo a atingir objetivos da rede. É composta por duas sub-linguagens, uma linguagem declarativa (Mission Control Language CSL-MCL) e outra imperativa (Run-time Patching Language CSL-RPL), e foi desenhada para permitir a atualização dos controladores durante a sua execução dando a possibilidade de controlo hierárquico com controladores mais simples nos níveis mais baixos. O seu motor de execução aloca dinamicamente recursos às tarefas e ajusta-se em tempo real à alocação de recursos, permitindo que os controladores sejam simples, intuitivos e possuam escalabilidade.

A motivação para a criação desta linguagem surge de dois problemas que estão interligados e que simplificam o controlo de redes de sensores móveis, que são:

- A necessidade de programação de controladores para redes distribuídas de sensores móveis ad hoc — surge devido às limitações humanas, onde um utilizador singular não consegue controlar simultaneamente os detalhes de baixo-nível de vários recursos;

- A necessidade de os controladores poderem ser alterados durante a sua execução — podem simplificar-se os controladores, uma vez que não necessitam de prever tudo o que pode acontecer podendo ser re-configurados conforme o objetivo que se pretende no decorrer do tempo.

2.2.3 BigActors

É abordado, numa mistura do modelo de bi-grafos e actores, o problema de modelação e controlo de robôs heterogéneos móveis, onde estes necessitam de comunicar entre si, observar, controlar e migrar sobre estrutura do mundo modelado. Para tal, é usado um modelo de BigActor [17], um híbrido que combina o modelo de atores [1] e o modelo de bi-gráfos [16] para modelar uma missão de monitoramento ambiental usando um conjunto de veículos autónomos.

2.2.4 *Vignettes*

Por meio de simulação computacional e animação é possível realizar estudos experimentais detalhados de cenários de emergência, que desempenham um papel crucial no desenvolvimento de soluções inovadoras para análise da situação e de apoio à decisão. Ao coordenar e automatizar tarefas é possível construir técnicas de vigilância, por exemplo na marinha, usando veículos autónomos, mas é essencial analisar em detalhe os diversos cenários, avaliar os seus algoritmos e garantir a qualidade das soluções, por via de testes sistemáticos com um nível de detalhe acima do que é possível com testes manuais. A

abordagem proposta para criar estes casos de teste, assume desenvolvimento interativo, e introduz o conceito de *vignettes* ao fazer uma análise e validação de *vignettes* complexas para testar exaustivamente cenários reais.

Vignette — é uma história incorporada num cenário, na medida em que a história vai evoluindo como um conjunto de eventos discretos envolvendo diferentes agentes e o ambiente físico no qual operam. Descreve com elevado detalhe a distribuição dos agentes e eventos no espaço e tempo, baseado em configurações iniciais, condições ambientais dinâmicas e aspectos operacionais incertos [22].

Gerador de *Vignettes* — Ferramenta que permite especificar *vignettes*.

Especificação de *Vignette* — descrição estática baseada em texto, estruturada e precisa de uma *vignettes*.

Uma especificação de *vignette* criada pelo gerador de *vignettes* representa o conceito de um modelo de máquina de estados que pode ser interpretada por um ambiente de simulação.

Exemplo:

Existe necessidade de especificar cem barcos de pesca numa determinada área, cada um equipado com sistema de comunicação. Tipicamente seriam definidos em separado, da seguinte forma:

```
WhiteTrafficArea                                    1
FishingBoat-1                                       2
Position(X, Y) = (x-1, y-1)                         3
CommunicationDevice-1                               4
Type = link-11                                      5
Range = 5000 m                                      6
.                                                   7
.                                                   8
.                                                   9
FishingBoat-100                                     10
Position(X, Y) = (x-100, y-100)                     11
CommunicationDevice-100                             12
Type = link-11                                      13
Range = 3000 m                                      14
```

Figura 2.9: Definição individual dos barcos de pesca [22]

No cenário '*White Traffic Area*, ilustrado pela Figura 2.10 (retirada de [22]), existe um número de agentes (*Fishing Boats*) que são referidos como '*white traffic*' e que se movem aleatoriamente dentro de uma área especificada.

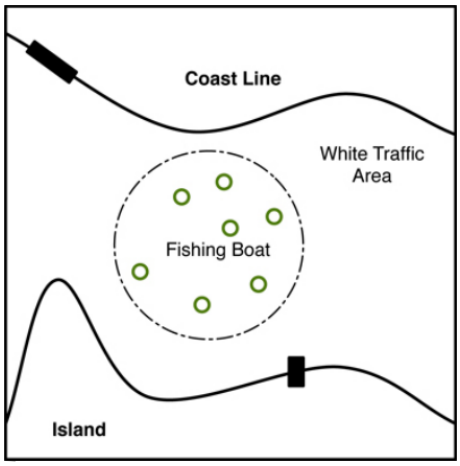


Figura 2.10: White Traffic Area Cenário

Na abordagem proposta, o cenário *White Traffic Area* seria representado da seguinte forma:

Vantagens:

Na primeira especificação (Figura 2.9), foi preciso detalhar manualmente todos os aspectos para cada barco de pesca, enquanto que na segunda especificação (Figura 2.11, retirada de [22]), o Gerador de *vignettes* produz esses detalhes automaticamente, mostrando alguns benefícios claros:

1. Definir e compreender a segunda especificação é muito mais fácil do que definir a primeira especificação onde é necessário definir individualmente cada barco;

2. Produzir *vignettes* (ou elementos de *vignettes*) é mais rápido, especialmente quando se precisa gerar *vignettes* que têm como base um padrão comum;

3. Qualquer erro na segunda especificação é mais facilmente detetado e corrigido do que na primeira especificação.

```
WhiteTrafficArea                                                    1
#number = 100                                                       2
FishingBoat                                                         3
Position(X,Y) = #valor aleatorio dentro da area A1                  4
CommunicationDevice                                                 5
Type = link-11                                                      6
Range = #valor aleatorio entre (2000 m) e (5000 m)                  7
#: Elementos e valores processados automaticamente                  8
    pelo Gerador de Vignettes                                       9
```

Figura 2.11: *White Traffic Area* produzido pelo gerador de *vignettes*

Capítulo 3

A Linguagem NVL

Este capítulo apresenta a linguagem de programação desenvolvida, para o controlo de múltiplos veículos autónomos, designada por NVL (*Networked Vehicles Language*), que especifica um comportamento global para um conjunto de veículos autónomos e projeta esse comportamento em programas locais a cada veículo.

Por forma a apresentar a linguagem NVL, começamos por descrever um exemplo de aplicação (Secção 3.1). O exemplo é depois usado como ilustração dos conceitos nucleares da linguagem, compreendendo: as noções de controlador (Secção 3.2) e procedimento (Secção 3.3; as instruções da linguagem para seleção de veículos (Secção 3.4), execução de controladores (Secção 3.5, e fluxo de controlo (Secção 3.6); e, finalmente, a implementação concreta de controladores (Secção 3.7).

3.1 Exemplo

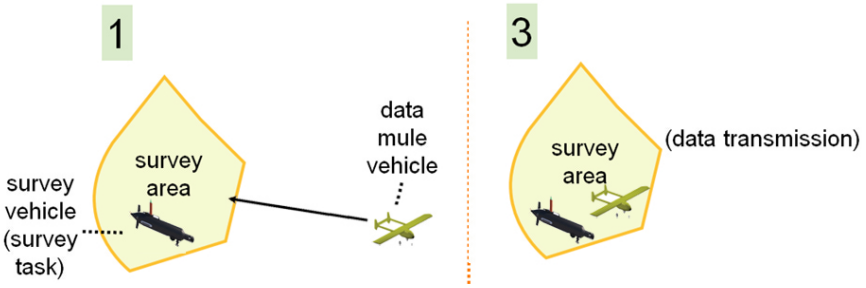


Figura 3.1: Cenário Exemplo

Devido à crescente procura de recolha de dados oceanográficos e missões de busca e salvamento, entre outras, tratamos aqui um cenário exemplo simples ilustrado pela Figura 3.1, que consiste na recolha e transmissão de dados entre dois veículos distintos. Consideramos que um veículo UUV recolhe dados numa determinada área e transfere esses dados para um outro veículo UAV, que funciona como mula de dados. A cada iteração de recolha de dados por parte do UUV, deve ser feita uma ligação entre o veículo UUV e um

UAV disponível, para que a troca de dados possa ocorrer.

De forma a satisfazer tal requisito, esquematizamos a operação desejada para um programa na Figura 3.2. O programa NVL terá que selecionar ambos os veículos da rede, de seguida instruir o UUV para executar a coleta de dados (invocar um controlador designado por collectData) e o UAV para se mover para a área de operação (moveToOpArea). Quando os veículos UUV e UAV terminarem as operações de collectData e moveToOpArea respetivamente, o programa executa um *rendezvous* (RV) entre os dois veículos para que a transferência de dados. Após o *rendezvous*, o programa deve recomeçar caso seja necessário recolher mais dados, ou terminar.

Na Figura 3.5, listamos um fragmento do programa NVL para o cenário exemplo, que usaremos nas secções seguintes para discussão dos vários aspectos da linguagem. Neste ponto, observe-se que um programa NVL consiste em um conjunto de procedimentos e controladores. No caso em apreço, são especificados apenas um procedimento main e os três controladores considerados para o exemplo: collectData (linha 2), moveToOpArea (linha 7) e RV (linha 12) e apenas um procedimento main (linha 19).

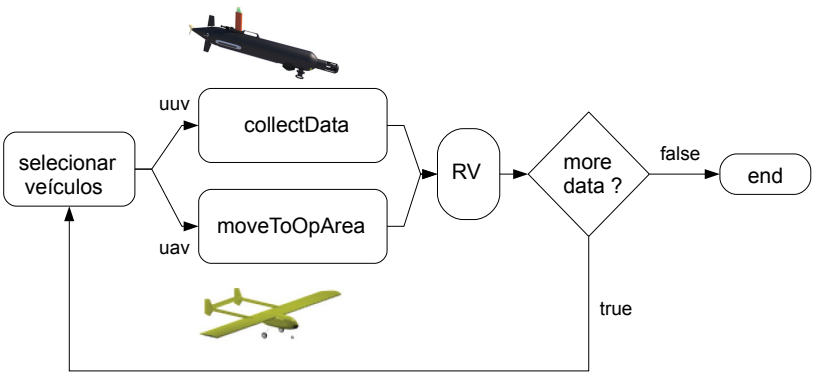


Figura 3.2: Fluxograma do Exemplo

3.2 Controladores

Um controlador NVL define um conjunto de veículos requeridos para a execução de uma determinada tarefa, que ações deve tomar cada veículo durante a execução da mesma, e um conjunto de possíveis resultados. O controlador especifica um comportamento global para o conjunto de veículos e este comportamento é projetado em programas locais a cada veículo (Figura 3.3).

Por exemplo, o controlador RV da Figura 3.5 (linhas 12–16) requer dois veículos, survey e mule, e declara três possíveis resultados (moreData, rvDone, e rvError). Os outros controladores no exemplo (collectData, moveToOpArea) requerem apenas um veículo

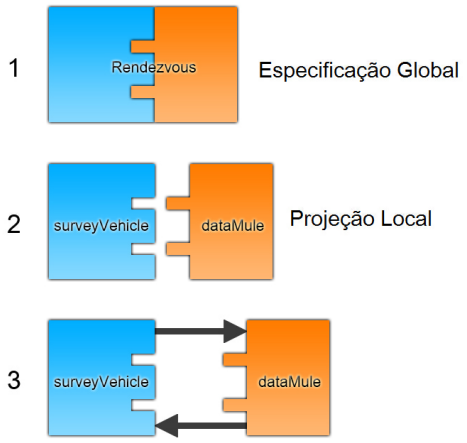


Figura 3.3: Conceito de Especificação e Projeção

e declaram apenas um resultado possível (done). Além desta especificação abstrata, a Figura 3.5 omite a implementação concreta dos controladores em questão, discutida adiante neste capítulo (Secção 3.7).

A associação de veículos a controladores segue uma relação tradicional entre tarefas, a execução de controladores, e recursos associados a essas tarefas, veículos. Para execução de um controlador, é necessário que lhe sejam previamente associados os veículos necessários (por exemplo dois no caso de RV acima). Durante o tempo que o controlador executar, estes veículos não poderão ser associados a qualquer outro controlador. No final da execução, é esperado que o controlador produza um resultado, de entre a lista de resultados que declara, e que os veículos sejam libertos para futura operação.

3.3 Procedimentos

Um programa NVL define um conjunto de procedimentos, entre os quais obrigatoriamente um procedimento chamado main, o procedimento executado de início pelo programa. O programa exemplo da Figura 3.5 define main como o único procedimento (linhas 19–52).

Um procedimento é uma sequência de instruções que especificam um fluxo de selecção de veículos numa rede e a associação de veículos selecionados à execução de controladores. Um programa NVL tem apenas uma linha de execução, e um procedimento activo associado a essa linha de execução. O significado das diferentes instruções NVL é sumarizado na Figura 3.4. Em seguida usamos o programa exemplo da Figura 3.5 para apresentar o uso das várias instruções.

Instrução	Descrição

choose <var> {
case <value 1> { <block 1> }
...
case <value n> { <block n> }
default { <fallback block> }
}
	Avalia o valor de <var> e executa o bloco de instruções <block i> se o <var> for igual a <value i>, ou executa <fallback block> se não houver correspondência com nenhum dos valores <value 1> até <value n>.

continue <proc>	Executa o procedimento <proc>.
delay <duration>	Pausa a execução durante <duration>.
exit	Termina o programa.
join <duration> <output 1> ... <output n> **or** { <instructions> }	Define um ponto de espera pela conclusão dos controladores executados. O programa espera no máximo uma duração <duration>, para obter o resultado dos controladores. Um bloco de instruções **or** pode ser especificado opcionalmente para lidar com erros. Em caso de omissão, está implícito um bloco **or** { **exit** }.
message <msg>	Imprime a mensagem <msg> na consola do programa.
select <duration> <vehicle 1> (<filter 1>), ... <vehicle n> (<filter n>) **then** { <block 1> } **or** { <block 2> }	Procura por n veículos disponíveis que satisfazem determinados filtros de seleção, durante um período de tempo <duration>, no máximo. Em caso de sucesso, as instruções no bloco **then** são executadas sequencialmente (<block 1>) com o veículo <vehicle 1> até ao <vehicle n> dentro do âmbito. Caso contrário, é executado o bloco de instruções (<block 2>) do bloco **or**. Em caso de omissão do bloco **or**, está implícito um bloco **or** { **exit** }.
spawn <output var> <ctrl> (<param 1> : <vehicle 1>, <param n> : <vehicle n>) **or** { <instructions> }	Executa o controlador <ctrl> usando o veículo <vehicle i> definindo o papel do mesmo no controlador, usando o parâmetro <param i>. O resultado do controlador é identificado por <output var>. Um bloco de instruções **or** pode ser especificado opcionalmente para lidar com erros. Em caso de omissão, está implícito um bloco **or** { **exit** }.

Figura 3.4: Instruções NVL

```
// controller specifications
controller collectData {
  vehicles v
  outputs done
  ...
}
controller moveToOpArea {
  vehicles v
  outputs done
  ...
}
controller RV {
  vehicles survey, mule
  outputs moreData, rvDone, rvError
  ...
}
...
// program procedures
procedure main {
 // select vehicles
 select 5m
  uuv ( type: "UUV",
        id: "lauv-seacon-1" ),
  uav ( type: "UAV" )
 then {
  // spawn survey controller for uuv
  spawn cRes collectData (v: uuv)
  // spawn moveToOpArea controller for uav
  spawn mRes moveToOpArea (v: uuv)
  join 1h cRes mRes
  // spawn RV controller for uuv and uav
  spawn rvRes RV(survey: uuv, mule: uav)
  join 15m rvRes
  // evaluate controller output
  choose rvRes {
   case moreData {
    // execute again
    continue main
   }
   default {
    // do nothing
   }
  }
 }
 or {
  // selection failed, retry 1 hour later
  delay 1h
  continue main
 }
 // end the program
 message "terminated"
}
```

Figura 3.5: NVL rendezvous program

3.4 Selecção de Veículos

Para o efeito de seleção de veículos disponíveis na rede, a NVL define a instrução **select**. A instrução seleciona e associa a variáveis do programa veículos disponíveis na rede que estejam de acordo com determinados critérios de selecção. Em associação à instrução, definem-se também 2 sequências de instruções, um bloco **then** e um block **or**. As instruções

do bloco **then** são executadas em caso de sucesso na seleção de veículos, ou seja, se todos os veículos necessários estão disponíveis antes do tempo limite. Em caso de falha na seleção, são executadas alternativamente as instruções do bloco **or**. Para esta lógica funcionar, assume-se naturalmente a existência de um mecanismo que anuncia a presença e disponibilidade de veículos na rede ao programa NVL.

O exemplo da Figura 3.2 contém uma instrução **select** logo no início do procedimento main. O critério de seleção é especificado por (Figura 3.5, linhas 21–24):

```
select 5m
uuv ( type: "UUV",
id: "lauv-seacon-1" ),
uav ( type: "UAV" )
```

Temos então que o tempo limite de seleção é de 5 minutos (5m, linha 21) e que se pretende selecionar dois veículos e associá-los às variáveis uuv e uav do programa. Os critérios de seleção estipulam que uuv deve corresponder a um veículo UUV e identificado por lauv−seacon−1 (type: "UUV", id: "lauv−seacon−1", linhas 22 e 23) e qualquer veículo de tipo UAV disponível (type: "UAV", linha 24).

Para a instrução **select** do programa exemplo temos também os blocos **then** (linhas 25–44) e **or** (linhas 45–49), cujo conteúdo é discutido a seguir. Neste ponto, note-se apenas que as variáveis uuv e uav ficam definidas (podem ser usadas) unicamente no âmbito do bloco **then**, visto que as variáveis de seleção apenas têm valores definidos no caso da seleção ser bem sucedida (o caso em que o **then** é executado). O bloco **or** não pode referenciar as variáveis.

3.5 Execução de Controladores

Após uma seleção bem sucedida de veículos, estes podem ser associados à execução de controladores. Para efeito, a NVL define duas instruções: **spawn** e **join**. Estas instruções permitem uma forma simples de paralelismo *"fork-join"*: é possível lançar de forma concorrente mais do que um controlador, com **spawn**, e sincronizar a sua posterior conclusão no fluxo do programa, com **join**. A instrução **spawn** lança um controlador, especificando os veículos a usar na execução do mesmo. Por sua vez, a instrução **join** aguarda pelos resultados da execução de um ou mais controladores até um tempo limite.

A execução de controladores usando **spawn** e **join** é ilustrada pelo programa exemplo (Figura 3.5, linhas 26–33). O fragmento em causa é o seguinte:

```
// spawn survey controller for uuv
spawn cRes collectData (v: uuv)
// spawn moveToOpArea controller for uav
spawn mRes moveToOpArea (v: uuv)
join 1h cRes mRes
// spawn RV controller for uuv and uav
spawn rvRes RV(survey: uuv, mule: uav)
join 15m rvRes
```

Usando duas instruções **spawn**, os controladores collectData para uuv e moveToOpArea

para uav são primeiro iniciados e executados concorrentemente. Seguidamente, uma instrução **join** espera que ambos os controladores terminem a execução no tempo limite de 1 hora (1h no código). Após este ponto de sincronização na operação de uuv e uav, (ambos) os veículos são associados a uma (única) execução do controlador RV com a terceira instrução **spawn**. A sequência termina com uma instrução **join** para aguardar o resultado do controlador RV.

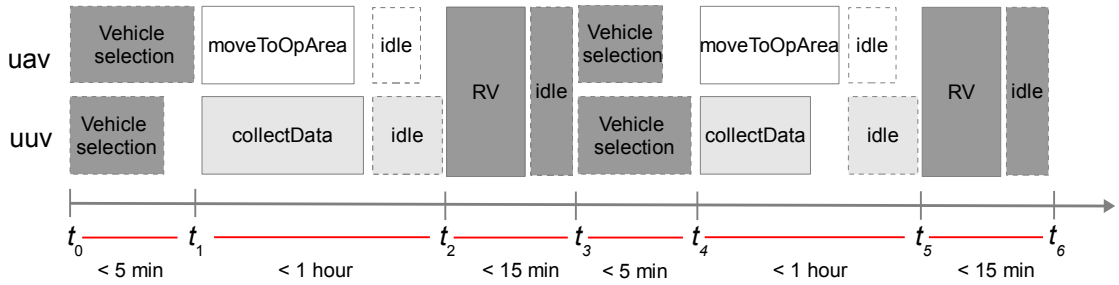


Figura 3.6: Execução do controlador para o programa exemplo

Uma possível execução para a sequência spawn-join é ilustrada na Figura 3.6. O tempo t_0 marca o início da seleção de veículos, feita a partir da instrução **select**. O tempo t_1 marca o início da execução do controlador moveTopOpArea para uav e do controlador collectData para o uuv. O tempo t_2 corresponde à instrução **join** que aguarda pelo fim da execução de ambos os controladores nos dois veículos, ainda nesse instante é iniciada a execução do controlador RV em ambos os veículos. O tempo t_3 marca o fim da execução do controlador RV e a chegada do seu resultado. Como indica a Figura 3.6, os tempos estão de acordo com os parâmetros no programa NVL: 5 minutos de tempo limite para selecionar os veículos pretendidos, 1 hora de tempo máximo para obter o resultado da execução dos controladores moveTopOpArea e collectData (primeiro **join**) e 15 minutos de tempo máximo para obter o resultado da execução do controlador RV (segundo **join**). Caso o controlador RV termine com resultado moreData o procedimento main é executado desde início, e uma execução semelhante de controladores poderá ter lugar, do tempo t_3 ao tempo t_6. As restrições temporais serão as mesmas na segunda execução. Dados os critérios do **select** no programa, o UUV usado será o mesmo nas duas iterações, mas o UAV utilizado poderá ser diferente.

A Figura 3.6 ilustra outro aspecto relevante na operação dos veículos. Os veículos podem ficar inativos durante algum tempo antes dos pontos de junção (**join**) mesmo que o controlador ao qual estejam associados tenha terminado a sua execução. Existem duas razões para este comportamento. A primeira é que o ponto de junção (instrução **join**) em causa pode estar a aguardar para que o outro controlador termine. A segunda razão pode ser derivada de fatores não negligenciáveis como atrasos na rede (i.e., devido a conexão intermitente) que podem acontecer antes do programa NVL tomar conhecimento que os controladores já terminaram efetivamente. O comportamento inativo do veículo é

assumido como estando implementado por um mecanismo predefinido a nível do veículo (por exemplo, emergir o UUV, o que acaba por facilitar a conectividade entre este e outro veículo).

Para finalizar a discussão da execução de controladores **spawn** e **join**, referimo-nos ao mecanismo de tratamento de erros. Os erros podem ter várias causas, tais como, não cumprimento do limite de tempo para execução ou perda de coneção com um veículo. Para lidar com erros, de forma similar a **select**, as instruções **spawn** e **join** podem ter um bloco **or** opcional associado, que é executado no caso de algum erro. Por omissão, o caso do programa exemplo, é assumido um bloco **or** { **exit** } que causa a paragem do programa.

3.6 Instruções de fluxo de controlo

Para governar o fluxo de controlo de um programa, a NVL define adicionalmente outras instruções: **choose**, **continue**, **delay** e **exit**.

A instrução **choose** avalia o resultado da execução de um controlador, para permitir ao programa uma escolha no seu fluxo em função desse resultado. Note-se que deverá ser precedida necessariamente por uma instrução **join** para o resultado que avalia. As escolhas de **choose** são definidas por múltiplas secções **case** e um bloco opcional **default**, num estilo semelhante à construção *switch-case* de muitas linguagens de programação. Cada secção **case** tem associado um valor constante do domínio de outputs do controlador em causa, e um bloco de instruções. Se o resultado avaliado for o definido para uma secção **case**, o programa procede com o bloco de instruções associado. Se nenhum dos valores for igual aos valores das secções **case**, as instruções do caso **default** são executadas. No exemplo dado, no fragmento:

```
choose rvRes {
   case moreData {
     // execute again
     continue main
   }
   default {
    // do nothing
   }
}
```

o resultado rvRes do controlador RV é avaliado para determinar se o programa deve ser executado novamente (caso o resultado tenha lançado moreData) ou simplesmente terminar (caso por defeito). De notar que o bloco **default** não contém instruções, permitindo que o programa proceda para a instrução na linha 51, na qual o programa termina com uma instrução **message**, que escreve uma mensagem para a consola do operador.

O fluxo do programa pode ser modificado com as instruções **continue**. A instrução **continue** habilita uma forma simples de *"continuation-style passing"*. A execução de **continue** <proc> faz com que o procedimento activo pare imediatamente e que o pro-

grama continue com a execução do procedimento <proc> (desde o seu início). No programa exemplo, a instrução é usada em dois locais para fazer com que o procedimento main seja executado repetidamente (re-executado desde o início): caso seja necessário recolher mais dados pelo UUV (Figura 3.5, linha 38), e para lidar com eventuais erros no processo de seleção de veículos (linha 48). Para além deste uso "iterativo", a instrução pode também ser usado para fluxo de controlo arbitrário entre procedimentos.

A instrução **delay** pausa a execução do programa durante um tempo especificado, antes de executar a instrução imediatamente a seguir no procedimento atual. No programa exemplo, existe uma pausa de 1 hora no caso de falha de seleção de veículos, especificada usando **delay** 1 h (Figura 3.5, linha 47).

Por fim, a instrução **exit** tem como efeito a paragem imediata do programa. No programa exemplo da ela não é usada, mas pode ser vista como implícita no fim do procedimento main, após a última instrução (Figura 3.5, linha 51).

3.7 Implementação de Controladores

A implementação de um controlador é baseada em código que necessita ser instalado em cada veículo que participa nas tarefas a serem executadas pelo controlador. Esse código pode ser definido diretamente por um programador ou gerado a partir de uma especificação codificada diretamente em NVL para definição do controlador. Não sendo a definição de controladores o objetivo principal da linguagem, esta facilidade permite no entanto a rápida definição de controladores simples. Como exemplo, a Figura 3.7 apresenta um fragmento NVL para a definição do controlador RV do programa exemplo.

Em NVL, um controlador é especificado como um tipo de máquina de estados finitos, na qual a cada estado corresponde a um passo de controlo, definido em secções **step**. Dois passos são definidos no exemplo da Figura 3.7, start e dataTransfer. Cada passo compreende por sua vez a especificação de duas fases temporizadas para o comportamento dos veículos, a fase de controlo, na secção **control**, e a fase de sincronização, na secção **synchronize**:

1. A fase de controlo define a execução de um plano IMC local a cada veículo em um tempo limite. Por exemplo, no passo start na figura, os planos IMC executados são surface e loiter para os veículos survey e mule, respetivamente, e o tempo limite é de 2 minutos;

2. A fase de sincronização, executada em sequência à fase de controlo, determina o próximo passo a executar conjuntamente pelos veículos, ou um resultado final para a execução do controlador. No passo start do exemplo ilustrado, se a fase de controlo terminar com sucesso em cada veículo, os veículos concordam em avançar para o próximo passo dataTransfer. Caso contrário, o controlador termina a sua execução com resultado rvError.

```
controller RV {
 vehicles survey, mule;
 outputs moreData, rvDone, rvError ;
 step start {
   control 2m {
     at survey surface;
     at mule loiter ;
   }
   synchronize 2m{
    dataTransfer <- mule:done survey:done ;
    stop rvError <- default;
   }
 }
 step dataTransfer {
  control 13m {
   at survey uploadData;
   at mule downloadData;
  }
  synchronize 2m{
    stop done < mule:done survey:done ;
    stop moreData < mule:done survey:getMoreData ;
    stop rvError < default;
  }
 }
}
```

Figura 3.7: Especificação do controlador Rendezvous

Capítulo 4

Desenho e Arquitetura

Neste capítulo descrevemos o desenho e arquitetura do sistema computacional de suporte à NVL. Começamos por introduzir a arquitetura do sistema e os seus componentes (Secção 4.1). Seguidamente, descrevemos o suporte à edição de programas (Secção 4.2), as interações fundamentais entre componentes na execução de programas (Secção 4.3), e aspectos complementares relativos à configuração e monitorização de veículos (Secção 4.4).

4.1 Arquitetura

A arquitetura da implementação da NVL é ilustrada na Figura 4.1. Esta arquitetura compreende um ambiente integrado para desenvolvimento de software (IDE) para criação de programas NVL, módulos de execução NVL na forma de um interpretador e supervisores de veículo, e três dos componentes do conjunto de ferramentas do LSTS apresentados no Capítulo 2 : o DUNE, o Neptus e o IMC.

Em sumário, o papel de cada um dos componentes da arquitetura é o seguinte:

- A escrita e validação de um programa NVL envolve o uso de um ambiente de edição, o NVL IDE;
- Um programa NVL (bem-formado) é executado por um interpretador da linguagem;
- O interpretador NVL interage com supervisores NVL locais a cada veículo, e que têm por papel responder a pedidos de execução de controladores e mediar o acesso ao veículo;
- Uma instância DUNE tem acesso direto ao veículo, respondendo a pedidos do supervisor NVL local;
- O Neptus é usado com os propósitos de edição de planos IMC locais a cada veículo, previamente à execução de um programa, e monitorização do estado dos veículos por operadores humanos, durante a execução;

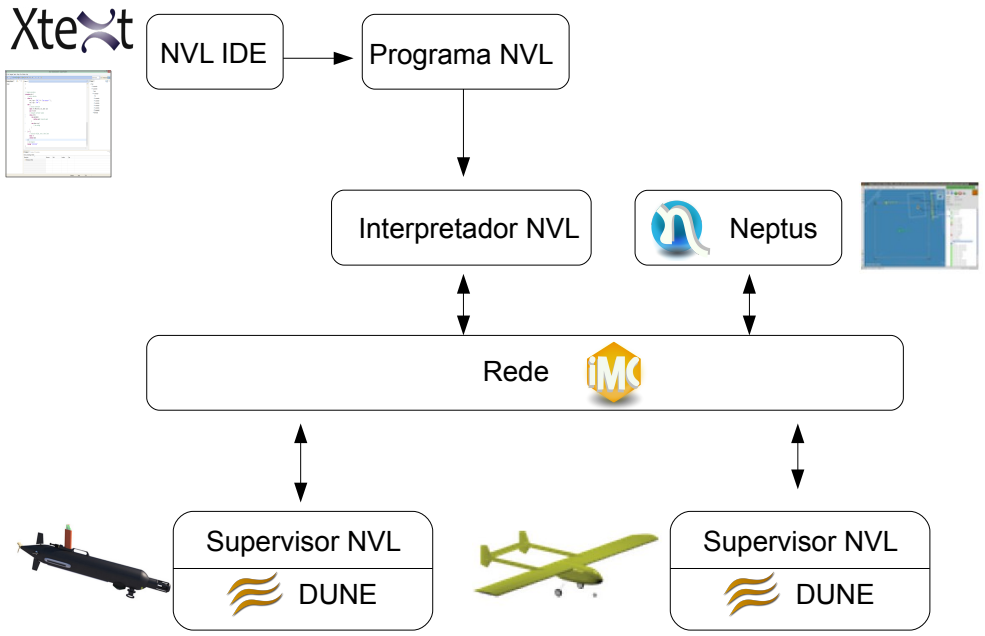


Figura 4.1: Arquitetura NVL

- No ambiente de rede em causa, os componentes juntam-se a esta ou saem dinamicamente, sendo a interação entre estes governada pelo protocolo IMC.

4.2 Ambiente de desenvolvimento

A linguagem NVL foi desenhada usando como recurso o sistema Xtext [3] [27]. O Xtext permite o desenho e implementação de linguagens de domínio especifico (*DSL* - Domain Specific Language), em integração com o popular ambiente de desenvolvimento do Eclipse IDE [6], usado para desenvolvimento de software nas principais linguagens de programação, como Java ou C/C++.

Com o Xtext foi possível lidar de forma relativamente simplificada com o desenho e implementação da linguagem, para os aspectos de definição da gramática da linguagem, análise sintática, validação semântica e geração de código. Do mesmo modo, permitiu também a criação de um plugin Eclipse, que pode ser gerado a partir de um projeto Xtext, e que contém todas as características habituais de um IDE moderno (por exemplo, editor com coloração de sintaxe, consola de erros, ou explorador de ficheiros), permitindo a edição e validação de programas de forma intuitiva. Uma consola de edição de um programa NVL é mostrada na Figura 4.2. Os itens principais de edição são assinalados na figura. A árvore de ficheiros de um projeto NVL pode ser examinada (1), incluindo os ficheiros de controladores sintetizados na pasta `src-gen`. O utilizador escreve o pro-

grama numa sub-janela de edição (2), sendo a sintaxe NVL colorida de forma amigável ao utilizador. Na barra lateral da mesma janela (3), são assinalados eventuais erros ou avisos na formatação do programa, sendo a mesma informação agrupada e listada na aba 'Problems' (4). O erro ilustrado na figura corresponde ao uso de uma variável xpto não definida.

Neste momento o IDE é apenas usado para a edição de programas, constituindo trabalho para futuros aspectos como a execução e monitorização de programas, ou a integração em consolas Neptus. Atualmente os programas são executados usando uma linha de comandos, e o Neptus é usado separadamente.

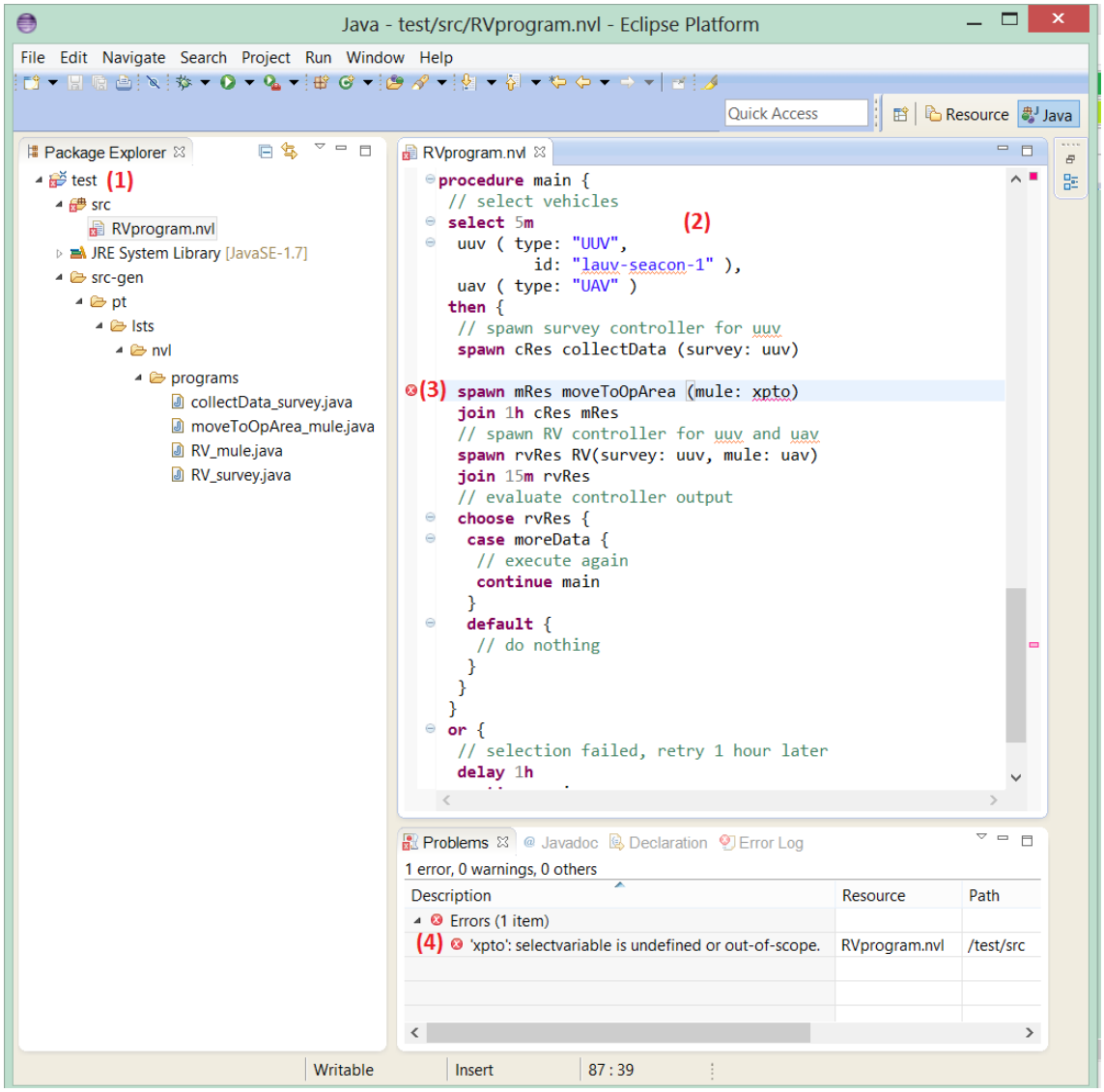


Figura 4.2: Edição de um programa NVL usando o Eclipse IDE

4.3 Execução da NVL

Para a execução de um programa, o interpretador interage com supervisores NVL locais a cada veículo. Os supervisores anunciam ao interpretador a disponibilidade do veículo para a execução de controladores, e respondem a pedidos do mesmo interpretador para a execução de controladores. Cada supervisor monitoriza e comanda o veículo fazendo por sua vez interface com a instância DUNE local.

Estas interações são conduzidas pela troca de mensagens IMC entre os vários participantes, esquematizadas na Figura 4.3. As mensagens assinaladas com prefixo `NVL` foram acrescentadas ao protocolo para suporte à NVL, enquanto as restantes já fazem parte do protocolo padrão. Discutimos a seguir as interações essenciais entre os vários componentes, e o uso inerente do IMC.

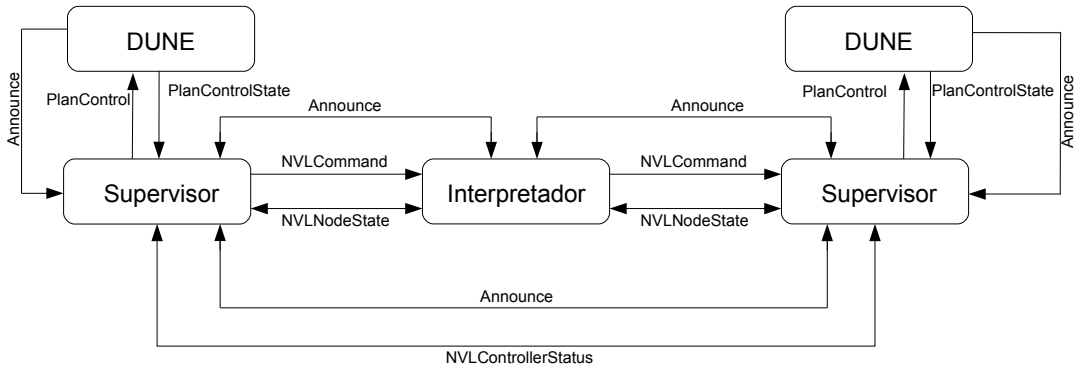


Figura 4.3: Fluxo mensagens IMC durante execução de um programa NVL

4.3.1 Anúncio e descoberta de componentes

Os vários componentes do ambiente de execução anunciam a sua presença na rede usando a mensagem `Announce`, segundo a convenção definida no IMC para um nó se anunciar na rede. Tipicamente a mensagem é transmitida por UDP *multicast*. Entre outros aspectos, uma mensagem `Announce` indica o tipo do nó origem (veículo, consola Neptus, módulo NVL, etc) e anuncia os portos UDP onde aceita a receção de dados IMC. Á parte configurações muito específicas, a comunicação ponto-a-ponto componentes nos sistemas do LSTS processa-se por UDP *unicast*.

Suportando esta lógica para anúncio de componentes, bem como a operação dual de descoberta, as ligações necessárias entre parceiros podem ser estabelecidas convenientemente. Em particular, isso terá de acontecer entre interpretador e supervisores NVL, e entre um supervisor NVL e a respectiva instância DUNE.

4.3.2 Disponibilidade e seleção de veículos

A disponibilidade do veículo é aferida por um supervisor a partir do estado IMC reportado pela instância DUNE para o veículo. No que concerne à execução da NVL, o estado de um veículo é definido pela mensagem periódica `PlanControlState` emitida pelo DUNE. A mensagem reporta se um plano IMC se encontra a ser executado, e nesse caso qual o plano em conjunto com outras informações complementares (por exemplo). Para um veículo poder ser selecionado com uma instrução **select**, deverá encontrar-se inativo, isto é, não estar a executar nenhum plano IMC (via NVL ou de outro modo), e não ter condições de erro assinaladas.

A disponibilidade de um veículo é desta forma observada por um supervisor NVL, e por sua vez reportada a um interpretador NVL usando a mensagem `NVLNodeState`. Esta mensagem abstrai detalhes do estado do veículo e reporta informação extra específica à execução de controladores NVL, quando estes são cativados (discutido a seguir).

4.3.3 Execução de controladores

Após a seleção de um conjunto veículos mediante instruções **select**, um interpretador e supervisores interagem por forma a executar controladores. Aquando da ativação de um controlador usando a instrução **spawn**, o interpretador dá ordens a cada supervisor afecto a um veículo envolvido para a execução. O comando é definido pela mensagem `NVLCommand`. Por sua vez, em resposta, o supervisor carrega o código de controlo guardado localmente e inicia-o, e a execução de um controlador interage finalmente com o DUNE, activando planos IMC com o comando `PlanControl`.

O interpretador observa a execução através da mensagem `NVLNodeState`, emitida periodicamente pelos supervisores, tendo em conta a reportagem de `PlanControlState` pelo DUNE (tal e qual no processo de seleção de veículos). O interpretador não exige uma frequência mínima temporal para a receção da mensagem, já que a execução de um programa deverá ser robusta a perdas de conectividade eventualmente longas. As restrições temporais são definidas apenas por instruções **join**, que definem o tempo limite para o término dos controladores, ou seja, a receção da mensagem `NVLNodeState` que assinalem esse término até ao tempo limite.

Complementando este quadro na execução de controladores, temos interações supervisor-supervisor quando um controlador envolve mais do que um veículo. Para controladores sintetizados da forma descrita no Capítulo 3 em particular, a troca de mensagens é crucial para que os veículos possam sincronizar no término de um passo de controlo, e ou na execução de mais um passo de computação. O estado local de controlo é expresso pela mensagem `NVLControlStatus` e transmitido por UDP *multicast*.

4.3.4 Tratamento de erros

Para todas as interações discutidas anteriormente, existem vários tipos de erro que podem ocorrer na execução de um programa, em associação a instruções **select**, **spawn** e **join**. São exemplos de erros comuns a transição de um veículo para estado de erro após a sua seleção ou ativação de um controlador para este, a perda de conectividade de um veículo, ou a não conclusão da execução um controlador no tempo limite expresso por um programa.

Todos os erros são tratados pelo interpretador pelos blocos de instrução **or**. Na ocorrência de um erro para uma instrução **select**, **spawn** e **join**, como discutido no Capítulo 3, o interpretador entra no bloco **or** associado à instrução e desassocia os veículos da execução programa. Para este feito, e independentemente da instrução, o interpretador envia um comando para abortar toda e qualquer execução que possa estar ativa para os supervisores/veículos que estejam envolvidos. Este comando é expresso (tal como para a execução de controladores) em mensagens `NVLCommand`.

4.4 Configuração e monitorização de veículos

O sistema Neptus é usado para configuração prévia de veículos, e posterior monitorização do seu comportamento execução. Focámos aqui os aspetos relevantes para suporte à NVL.

Em termos de configuração, o Neptus é usado para desenhar e transferir os planos IMC que poderão ser invocados por controladores NVL. Durante a execução, esses planos podem ser monitorizados. Esta interação decorre diretamente entre Neptus e instâncias DUNE, sem mediação dos componentes específicos à NVL.

A Figura 4.4 ilustra uma consola Neptus, em que são visíveis: um painel (1) para comandos de planos (por ex. envio, início, ou término de plano) e sua monitorização, (2) uma lista de planos que podem ser transferidos para veículos, e (3) um mapa da zona operacional onde são desenhadas as posições do veículo bem como o (s) planos atual(is) em execução. A figura ilustra a execução de um programa NVL para dois veículos, identificados como lauv−secon−1 e lauv−seacon−2.

Note-se que é a infra-estrutura específica à NVL (interpretador, supervisores) que iniciam automaticamente os planos IMC, sem intervenção do Neptus. Este é apenas estritamente necessário à conceção dos planos IMC, sendo que, caso desejável em face de um problema, o Neptus poderá interferir na execução do programa terminando a operação de um plano. Nesta situação, o término do plano é detetado pela infraestrutura NVL (a partir do estado em resposta reportado pelo veículo), e assumido como um erro de execução, aplicando-se a metodologia anteriormente discutida (Secção 4.3.4).

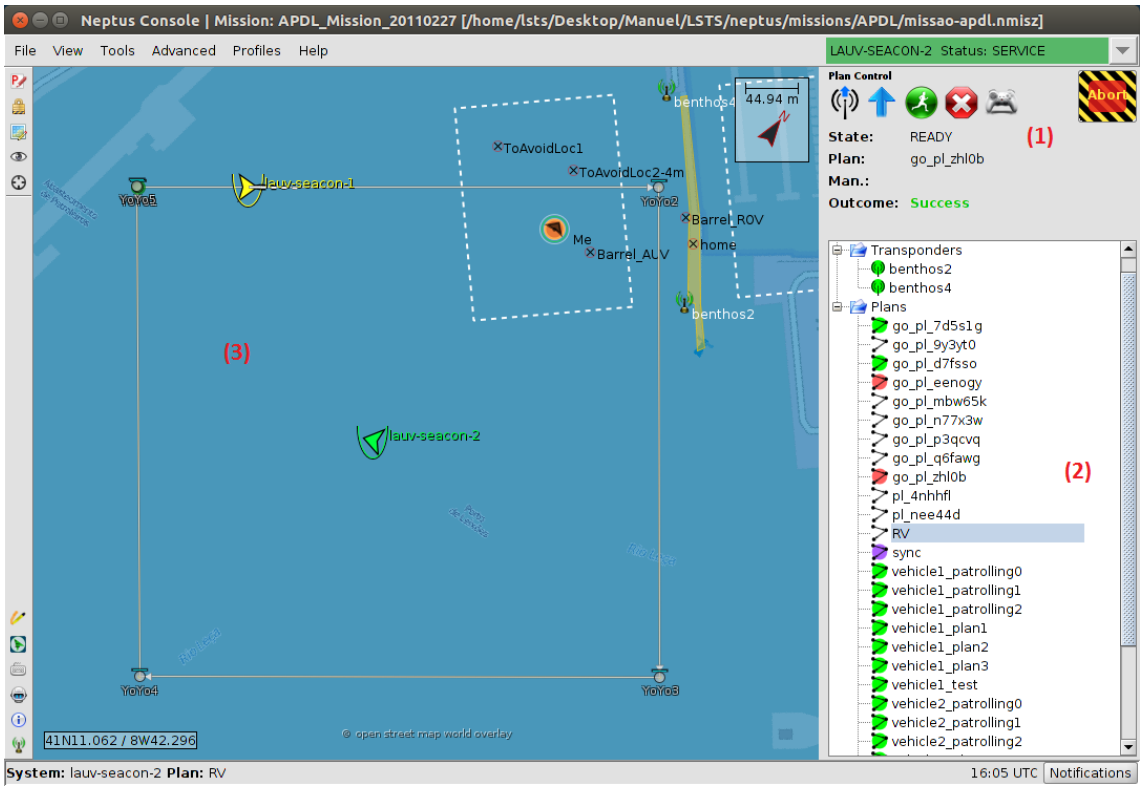


Figura 4.4: Configuração e monitorização usando o Neptus

Capítulo 5

Implementação

Este capítulo aborda os aspectos técnicos fundamentais de implementação da linguagem NVL, em correspondência ao desenho e arquitetura expostos no capítulo anterior.

Primeiro (Secção 5.1) descreve-se o uso do *framework* Xtext na definição da linguagem, no que se refere à sua especificação formal e validação, e ainda a geração de código para controladores.

Seguidamente (Secção 5.2) , explica-se a implementação do ambiente de suporte à execução de programas, nomeadamente a estrutura, configuração e codificação do interpretador e supervisores NVL, bem como as extensões ao protocolo IMC em suporte à NVL.

5.1 Ambiente de desenvolvimento

5.1.1 Gramática da linguagem

A sintaxe da NVL é especificada usando uma gramática no formato Xtext. Usando a terminologia padrão (ex., [2]), a gramática de uma linguagem é definida usando regras de produção. Cada regra de produção identifica um símbolo não-terminal à esquerda e as suas possíveis derivações em sequências de outros símbolos não-terminais ou terminais (os símbolos "finais" de derivação como palavras-chave, operadores, etc) à direita. Um símbolo não-terminal raiz indica o ponto de partida para derivações possíveis para uma linguagem.

A Figura 5.1 ilustra a correspondência entre um fragmento de um programa NVL (à esquerda) e a definição da gramática para a linguagem no formato Xtext. Na gramática, TopProgram é o símbolo raiz. A regra de produção associada a TopProgram indica que as derivações possíveis (i.e., os programas NVL) começam por uma sequências de símbolos do tipo Controller (controladores), seguidas de uma sequência de símbolos do tipo Procedure (procedimentos). O esquema aplica-se sucessivamente. Por exemplo uma derivação de Procedure começa com a palavra-chave **procedure**, seguida de um identificador e de uma sequência de derivações do símbolo Instruction , as instruções do corpo

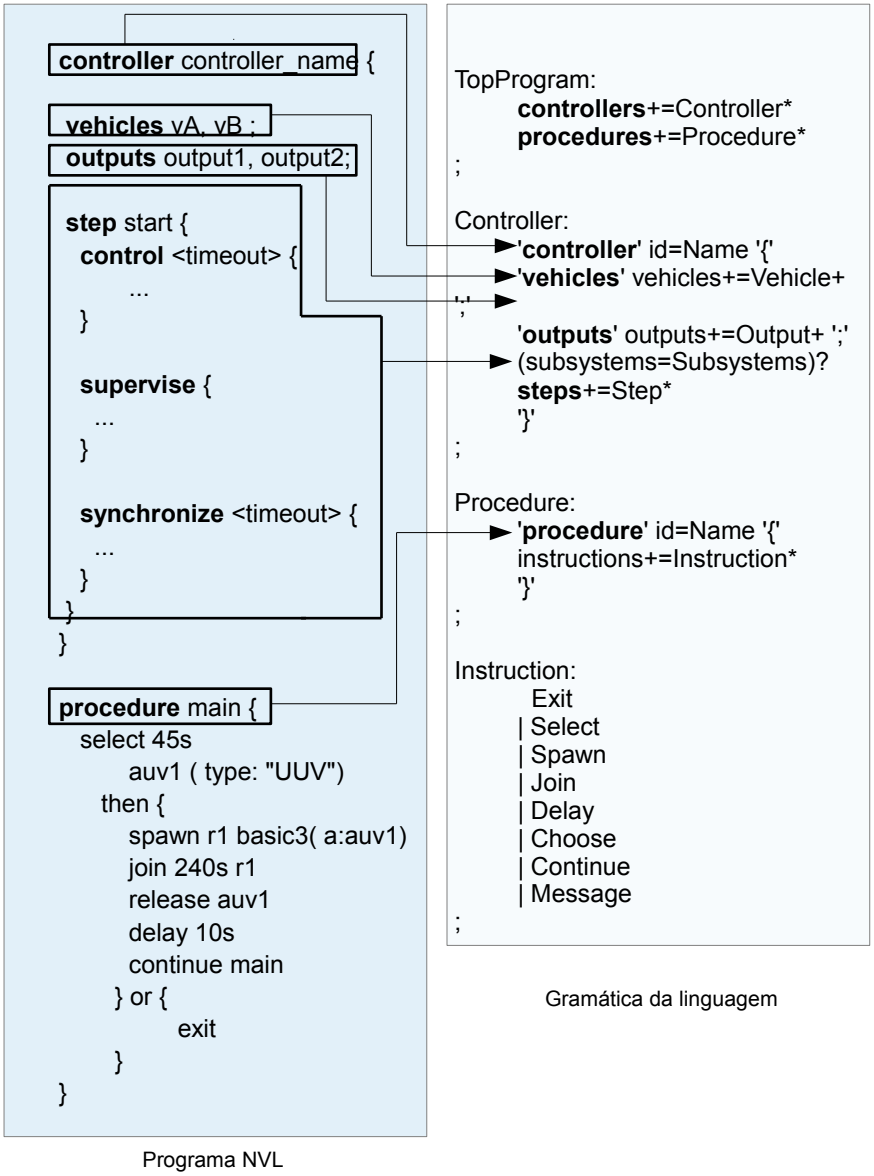


Figura 5.1: Relação entre gramática e estrutura do programa

de um procedimento.

Nas várias regras da Figura 5.1, empregam-se identificadores de atributos em associação a cada símbolo, por exemplo controllers e procedures para TopProgram. Esses atributos corresponderão a atributos de classes geradas pelo Xtext para as árvores sintáticas dos vários símbolos. Voltando ao exemplo de TopProgram, é gerada uma API de suporte a TopProgram que permite obter os valores dos atributos controllers e procedures, por sua vez também instanciados por meio de classes geradas pelo Xtext. Estas estruturas de dados são usadas na validação da linguagem e geração de código, que operam uma visita

em profundidade pela árvore sintática de um programa.

5.1.2 Validação

A validação da linguagem NVL foi implementada visitando os nós da árvore sintática, à semelhança de um *visitor pattern* [8], de forma a poder assinalar erros aquando da edição de programas NVL.

Na programação do controlador podem ser assinalados diversos erros, tais como quando:

- existe tarefa de controlo em duplicado;
- não existe uma tarefa de controlo para os veículos afetos a esse controlador;
- existe tarefa de supervisão em duplicado;
- existe uma regra de sincronização em duplicado;
- existem resultados (**outputs**) definidos e que nunca são usados
- não existem passos (steps) definidos;
- não existe um passo start definido;
- existem passos definidos e que nunca são usados;
- a duração da fase de controlo e/ou fase de sincronização não se encontra dentro dos limites predefinidos;

A nível de programação de procedimentos, são assinalados erros quando:

- não existem procedimentos definidos no programa;
- não existe um procedimento main definido;
- existem variáveis definidas em duplicado;
- as variáveis utilizadas estão fora do âmbito no qual foram especificadas;
- existe falta de instruções **join** após execução de um controlador ;

Como forma de ilustrar o princípio de validação da linguagem, apresentamos o seguinte fragmento de código, que faz a validação base do programa NVL percorrendo a árvore sintática do programa:

Numa primeira instância são validados todos os controladores especificados no programa NVL. Para validar cada controlador adicionam-se símbolos para cada veículo (a uma lista de símbolos com ordem, na qual os elementos mais novos são adicionados ao inicio da lista), os resultados e os passos (Step) e valida-se cada passo (função

```
def void checkProgram(TopProgram p) {                                          1
                                                                               2
  scopes.clear();                                                              3
  boundVehicles.clear();                                                       4
  joinSet.clear();                                                             5
  beginScope();                                                                6
                                                                               7
  for (Controller c : p.controllers) {                                         8
    validateController(c);                                                     9
  }                                                                            10
                                                                               11
  if (p.procedures == null || p.procedures.size == 0) {                        12
    error("No procedures are defined.",p,                                      13
       NvlPackage.Literals.TOP_PROGRAM_PROCEDURES);                            14
  } else {                                                                     15
    for (Procedure proc : p.procedures) {                                      16
      defineSymbol(proc.id, proc);                                             17
    }                                                                          18
    for (Procedure proc : p.procedures) {                                      19
      validateProcedure(proc);                                                 20
    }                                                                          21
    if (p.procedures.filter[id.id.equals("main")].size == 0) {                 22
      error("No procedure called 'main' is defined.",p,                        23
         NvlPackage.Literals.TOP_PROGRAM_PROCEDURES);                          24
    }                                                                          25
  }                                                                            26
  endScope();                                                                  27
                                                                               28
}                                                                              29
                                                                               30
                                                                               31
def void validateController(Controller c) {                                    32
  defineSymbol(c.id, c);                                                       33
  beginScope();                                                                34
                                                                               35
  // Define all symbols first.                                                 36
  for (Vehicle v : c.getVehicles()) {                                          37
    defineSymbol(v.id, v);                                                     38
  }                                                                            39
  for (Output o : c.getOutputs()) {                                            40
    defineSymbol(o.value, o);                                                  41
  }                                                                            42
  for (Step s : c.getSteps()) {                                                43
    defineSymbol(s.step, s);                                                   44
  }                                                                            45
  // Validate each step now.                                                   46
  for (Step s : c.getSteps()) {                                                47
    validateStep(c, s);                                                        48
  }                                                                            49
  endScope();                                                                  50
}                                                                              51
```

Figura 5.2: Validação global de um programa NVL

validateStep, linha 48). A função validateStep verifica se estão definidas ações para cada fase de controlo e sincronização, assim como também verifica a existência de ações em duplicado nas fases de controlo, supervisão e sincronização. De seguida são validados os procedimentos, caso existam procedimentos especificados, definem-se símbolos (adiciona-se a uma lista de símbolos) também para todos os procedimentos definidos no programa (linha 17) e a função validateProcedure invoca uma função de validação para cada instrução do procedimento.

Um outro exemplo de validação é dado na Figura 5.3, relativo à instrução **select**.

```
def dispatch void validate1(Select s) {                    1
  validateOrBlock(s, s.orBlock);                           2
  beginScope();                                            3
  for (SelectVariable v : s.variables) {                   4
    defineSymbol(v.variable, v);                           5
  }                                                        6
  validateBlock(s.instructions);                           7
  endScope();                                              8
}                                                          9
```

Figura 5.3: Validação da instrução **select**

A validação começa por avaliar se existe um bloco Or (linha 2) e validar as instruções contidas nesse bloco Or (com a função validateOrBlock, linha 2), de seguida definem-se símbolos (da mesma forma explicada anteriormente) para cada atributo da função SelectVariable que são: o nome a atribuir ao veículo (variable=Name) e os filtros a aplicar (filters +=SelectionFilter$*$), e de seguida é invocado o método validateBlock que trata de validar todas as instruções contidas no bloco **then** da instrução **select**.

5.1.3 Geração de código

A geração de código é feita de forma automática no momento de criação/edição de um programa NVL usando para isso o plugin gerado pelo Xtext. Esta geração tem um código-esqueleto como base, onde são alterados apenas símbolos, que vão sendo atualizados de acordo a edição do programa NVL, criando uma classe Java *on-the-fly* respetiva a cada controlador especificado no programa NVL. A Figura 5.4, ilustra a escrita de um programa NVL que ao ser editado no ambiente IDE NVL se transforma em código Java capaz de ser executado nos veículos. De forma similar à validação de instruções, está em causa uma visita pela árvore sintáctica de cada controlador.

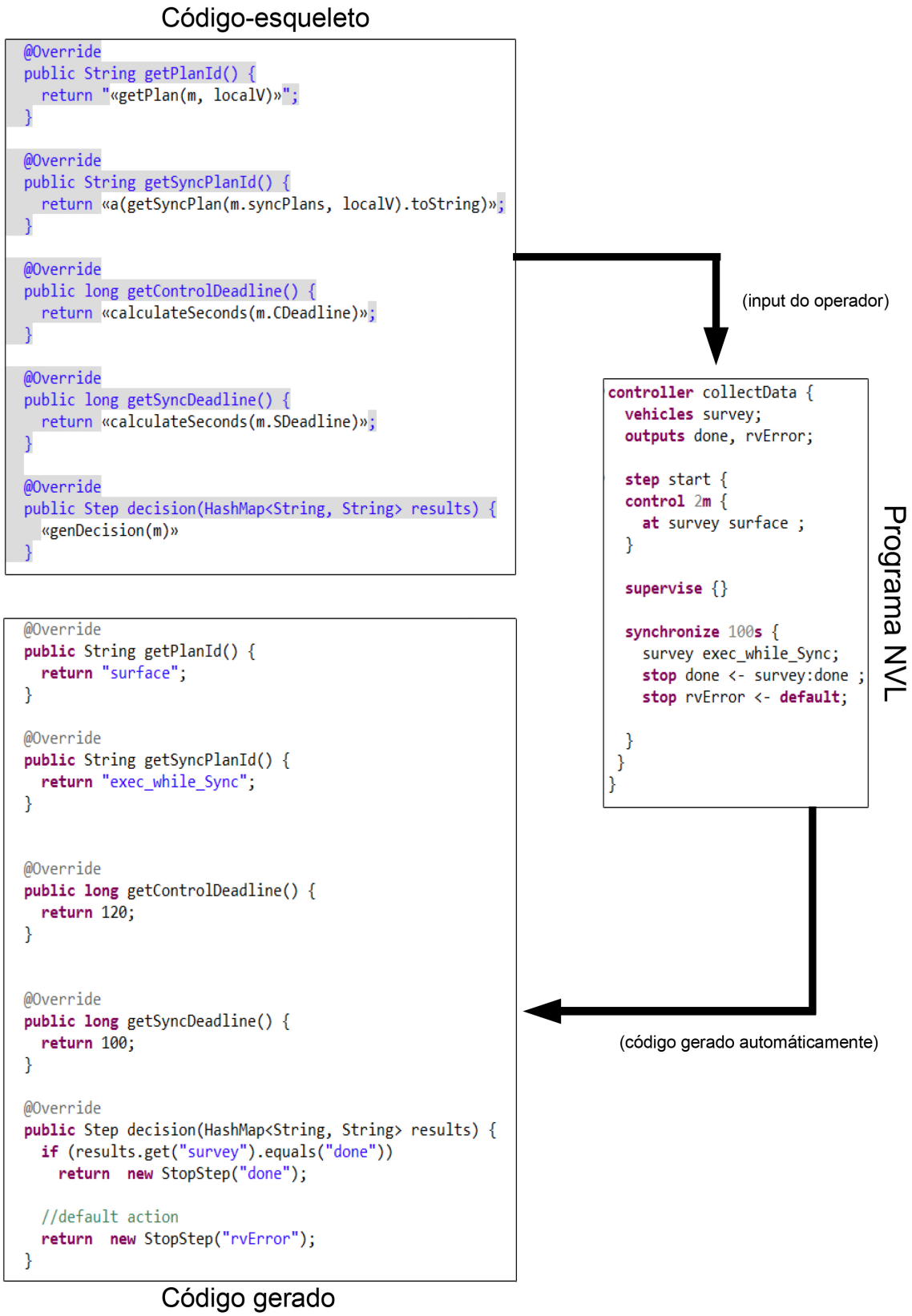


Figura 5.4: Ciclo de geração de código

O Xtext permite exportar um plugin com toda a estrutura da linguagem definida, a NVL, permitindo a criação e edição de programas NVL, com coloração de sintaxe, uma consola de erros e avisos e vista em árvore de ficheiros gerados automaticamente a partir do programa em edição (Figura 5.5).

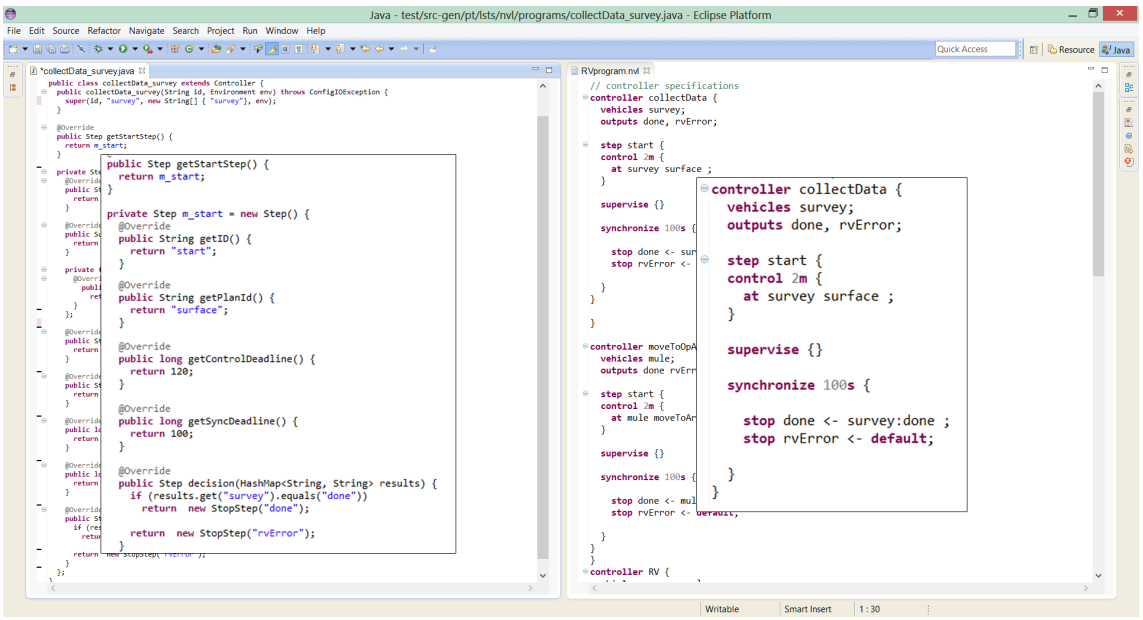


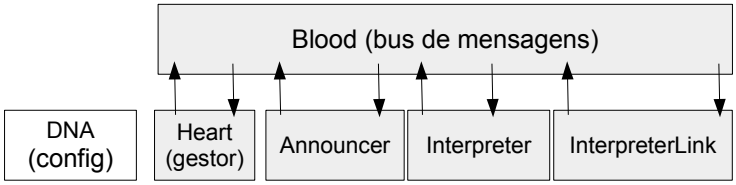
Figura 5.5: Edição de um programa NVL (à direita) e código gerado (à esquerda)

5.2 Ambiente de execução

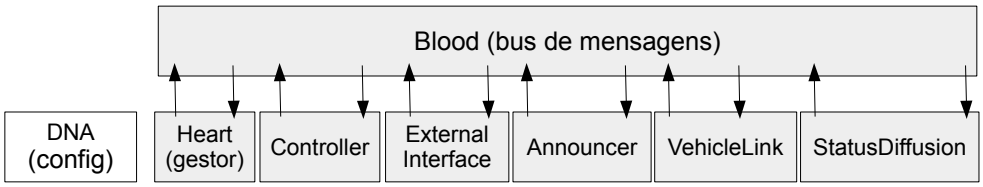
5.2.1 Organização geral

O ambiente de execução da NVL foi programado em Java, sendo a sua estrutura geral ilustrada na Figura 5.6, em termos da estrutura do interpretador (a), supervisor (b), e a hierarquia de classes subjacente (c). Os traços base de implementação são os seguintes:

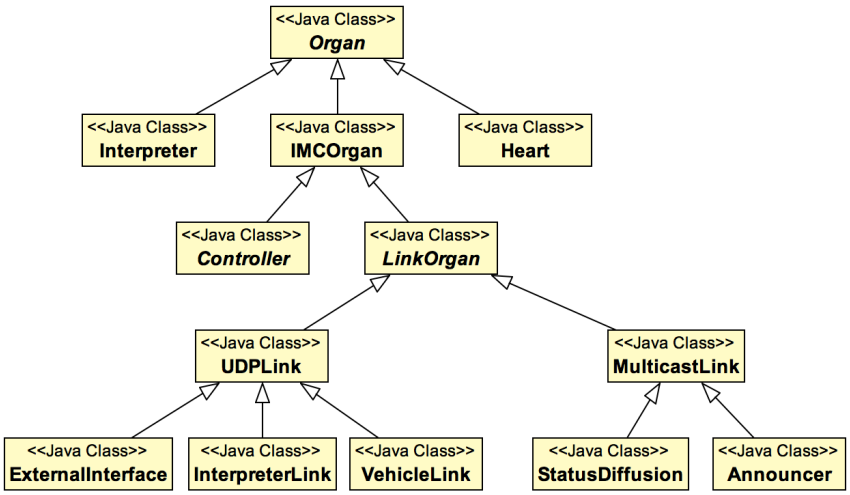
- Os componentes são organizados em tarefas, designadas por órgãos, que correm como *threads* Java independentes.
- Um órgão especial, chamado Heart (o "coração"), gere o funcionamento geral do programa (interpretador ou supervisor).
- Os órgãos do programa e respetiva parametrização são configurados num ficheiro de texto, identificado como DNA.
- Os órgãos trocam assincronamente mensagens entre si através de um *bus* de mensagens, designado por Blood ("sangue").



(a) Estrutura do interpretador



(b) Estrutura do supervisor



(c) Hierarquia de classes Java

Figura 5.6: Esquema geral de implementação

- Para comunicação externa com componentes do *toolchain* LSTS (DUNE, Neptus) ou outros NVL, parte dos órgãos mantêm ligações de rede com dados IMC, fazendo assim a ponte entre o ambiente interno e externo. Os órgãos em causa são subclasses da classe abstrata base LinkOrgan.

- Os órgãos formam uma hierarquia de classes Java, por forma a promover extensão e reutilização de funcionalidade.

Examinemos agora as estruturas do interpretador e supervisor NVL, que vão de encontro ao desenho geral do ambiente de execução descrito na Secção 4.3 do capítulo anterior. A estrutura do interpretador (Figura 5.6a) compreende três órgãos de funcionalidade (para além do "coração"): Interpreter, o órgão central de interpretação de um programa; InterpreterLink, para comunicação com supervisores NVL, ; e o Announcer, para anúncio e descoberta de componentes da rede. Por sua vez, o supervisor (Figura 5.6b) compreende também o Announcer, e ainda quatro outros órgãos de funcionalidade: Controller, para execução de controladores; VehicleLink, para comunicação com o DUNE; ExternalInterface, para comunicação com o interpretador NVL; e finalmente StatusDiffusion, para comunicação com outros supervisores NVL.

5.2.2 Esquema de configuração

As configurações (*DNA*) do interpretador e supervisor NVL são expressas por ficheiros de texto simples. Exemplificamos o formato com o ficheiro de configuração para o interpretador na Figura 5.7.

Podemos ver que o ficheiro está dividido em várias secções em correspondência aos vários órgãos do interpretador, e contém ainda uma secção especial IMC que identifica o nome do interpretador enquanto nó IMC (interpreter −nvl).

A configuração de cada órgão contém um conjunto de atributos lidos na inicialização do mesmo em tempo de execução. A configuração do órgão Heart em particular identifica com o atributo organs quais são os órgãos que deverão ser ativados no início do programa.

```
Heart {
  organs : Announcer,Interpreter,InterpreterLink ;
}

IMC {
  vehicle : interpreter-nvl ;
}

Announcer {
  type : Organ ;
  port : 30102 ;
  multicast-address : 224.0.75.69 ;
  implementation : pt.lsts.nvl.organ.imc.Announcer ;
}

Interpreter {
  type : Organ ;
  implementation : pt.lsts.nvl.interpreter.Interpreter ;
}

InterpreterLink {
  type : Organ ;
  port : 10005 ;
  implementation : pt.lsts.nvl.interpreter.InterpreterLink ;
}
```

Figura 5.7: Ficheiro de configuração para o interpretador

```
Heart {
  organs : VehicleLink,Announcer,ExternalInterface,StatusDiffusion ;
}

Announcer {
  type : Organ ;
  port : 30101 ;
  multicast-address : 224.0.75.69 ;
  implementation : pt.lsts.nvl.organ.imc.Announcer ;
}

ExternalInterface {
  type : Organ ;
  port : 10001 ;
  implementation : pt.lsts.nvl.controller.ExternalInterface ;
}

StatusDiffusion {
  type : Organ ;
  port : 10000 ;
  multicast-address : 224.0.0.1 ;
  implementation : pt.lsts.nvl.controller.StatusDiffusion ;
}

VehicleLink {
  type : Organ ;
  port : 9001 ;
  implementation : pt.lsts.nvl.controller.VehicleLink ;
}
```

Figura 5.8: Ficheiro de configuração para o supervisor

Para cada um dos outros órgãos, o atributo implementation identifica a classe Java para o órgão em causa. Os restantes parâmetros mostrados dizem respeito à funcionalidade específica de cada órgão, por exemplo a configuração de rede do 5.7 para o Announcer e do InterpreterLink.

A configuração do supervisor segue o mesmo formato e é mostrada na Figura 5.8.

5.2.3 Implementação de órgãos

A implementação de órgãos é expressa no diagrama UML da Figura 5.9. Os traços gerais de implementação são os seguintes

- A implementação de um órgão (Organ) faz uso de um objeto da classe Environment que agrupa o *bus* de mensagens (Blood), a configuração (DNA), e um *log* de mensagens (Log). Adicionalmente, a classe Timers mantém referência a métodos do órgão que devem ser invocados periodicamente (como explicado à frente no texto).

- Alguns métodos em Organ devem ser apropriadamente redefinidos para definir o ciclo-de-vida de um órgão concreto (i.e., subclasse de Organ), nomeadamente os métodos para nascimento/inicialização (onBirth()), morte (onDeath()), e onTick() para cada passo de computação (invocado ao longo da execução).

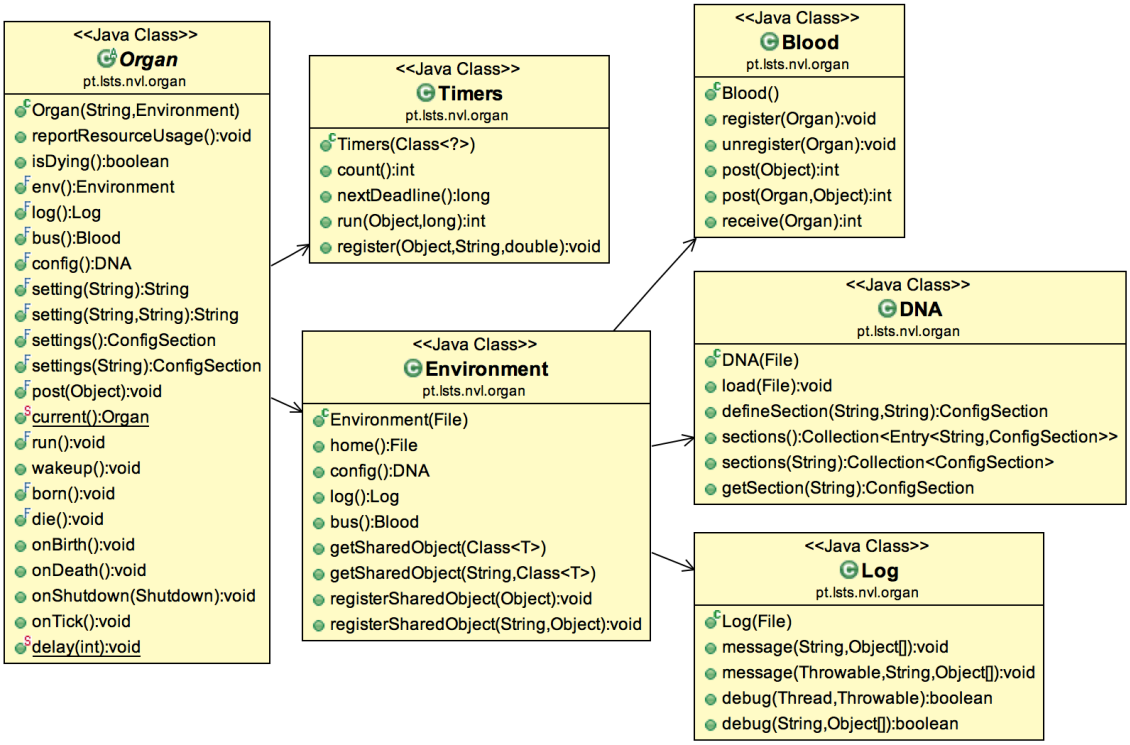


Figura 5.9: Diagrama UML para a implementação de órgãos NVL

- Os métodos utilitários bus(), config(), e log() em Organ dão acesso direto ao *bus* de mensagens, configuração, e *log* do sistema. O método post() permite enviar diretamente enviar uma mensagem para o *bus()*.

A título de exemplo do padrão de implementação listamos na Figura 5.10 fragmentos de código para a implementação do órgão Organ. Relembre-se que este é o órgão que gere todo o ambiente de execução, de qualquer forma a implementação de outros órgãos segue o mesmo padrão. Em detalhe:

- O construtor de Heart define um objecto Environment que será partilhado por todos os órgãos a partir da localização de um ficheiro de configuração.

- Os métodos onBirth() e onDeath() caracterizam as acções durante a ação e término do órgão. Como no caso o Heart é o órgão gestor, essas ações correspondem respetivamente a lançar os outros órgãos ou aguardar o seu término.

- Um método anotado com @Subscription serve para receber mensagens de determinado tipo do *bus* do sistema. No caso temos que onLaunch() é invocado aquando da receção de uma mensagem de tipo LaunchOrgan. No caso, está em causa o lançamento dinâmico de órgãos (necessários na implementação do supervisor) além dos ativados inicialmente.

- Por sua vez o método shutdownProcedure() exemplifica o lançamento de uma mensagem para o *bus* usando o método post(). O método em causa é invocado aquando da terminação abrupta do programa pelo utilizador (ex. usando o típico Control+C). Em resposta o Heart publica uma mensagem Shutdown a ser processada por todos os órgãos.

- Um método anotado com @Frequency é invocado periodicamente pelo sistema, com uma frequência expressa em Hertz (Hz). No exemplo, temos que o método checkSanity() é invocado com uma frequência de 1 Hz (1 vez por segundo).

```
package pt.lsts.nvl.organ;
...
public class Heart extends Organ ... {
  ...
  public Heart(File DNAfile) throws IOException  {
    super("Heart", new Environment(DNAfile));
    ...
  }
  ....
  @Override
  public void onBirth() {
    // Start organs
    for (Organ organ : _organs) {
      startOrgan(organ);
    }
  }
  ...
  @Override
  public void onDeath() {
    for (Organ organ : _organs) {
      try {
        organ.join();
      } catch (InterruptedException e) {

      }
    }
  }
  ...
  @Subscription
  public void onLaunch(LaunchOrgan msg) {
    ...
  }
  ...
  private void shutdownProcedure() {
    log().message("Program termination requested.");
    bus().post(new Shutdown());
    ...
  }
        ...
  @Frequency(1)
  public void checkSanity() {
    ...
  }
  ...
}
```

Figura 5.10: Fragmentos de código do órgão Heart

```
<message id="830" abbrev="NVLControllerStatus" name="NVL Vehicle Plan Status"
         source="vehicle">
    <field name="Step Id" abbrev="step_id" type="plaintext" />
    <field name="Stage" abbrev="stage" type="uint8_t"
           unit="Enumerated" prefix="NVL">
    <description>Vehicle result while executing a step.</description>
        <value id="0" name="Control" abbrev="CONTROL" />
        <value id="1" name="Gather" abbrev="GATHER" />
        <value id="2" name="Reduce" abbrev="REDUCE" />
        <value id="3" name="Done" abbrev="DONE" />
    </field>
    <field name="Result" abbrev="result" type="plaintext" />
    <field name="States" abbrev="states" type="message-list"
           message-type="NVLControlResult" />
    <field name="Sync Time" abbrev="synctime" type="fp64_t" unit="s" />
</message>

<message id="831" abbrev="NVLControlResult" name="NVL Control Result"
         source="vehicle">
    <field name="Participant Address" abbrev="participant" type="uint16_t" />
    <field name="Result" abbrev="result" type="plaintext" />
</message>
```

Figura 5.11: Mensagens IMC: NVLStatus e NVLControlResult

5.2.4 Extensões ao IMC

Na Figura 5.11 estão definidas as mensagens IMC NVLControllerStatus e NVLControlResult. A mensagem NVLControllerStatus é a mensagem usada para a sincronização entre veículos. Esta mensagem identifica em que passo (Step_id linha 3) do controlador se encontra o veículo a executar, O campo Stage (linha 4) identifica qual o estado do veículo enquanto executa o passo especificado anterior, enumerando quatro diferentes fases (a fase de controlo [Control], a fase de recolha de resultados dos outros participantes [Gather], a fase de tomada de decisão [Reduce] baseada nos resultados dos outros participantes, obtidos pela fase anterior, e a fase [Done] em casos que o controlador é executado apenas com um veículo e não existe portanto necessidade de sincronização com outros participantes), o campo Result (linha 12) identifica o resultado que do veículo após execução de cada passo, o campo Stages (linha 13) é uma lista de mensagens IMC NVLControlResult, onde estão contidos os endereços (linha 20) e resultados (linha 21) de todos os participantes envolvidos na execução de um controlador.

Para dar inicio à execução e monitorização do estado de execução de controladores pelos veículos, foram também criadas duas mensagens IMC: NVLCommand e NVLNodeState. A mensagem NVLCommand (linha 1) contém o campo Command (linha 3) que identifica qual a operação a ser realizada (Iniciar a execução do controlador [Start], parar a execução do controlador [Stop] ou listar todos os controladores carregados no veículo [List]). Caso a operação seja iniciar a execução de um controlador, o campo Program (linha 10) identifica qual o controlador a ser executado e o campo Role (linha 11) identifica qual o papel do veículo no controlador.

```
<message id="832" abbrev="NVLCommand" name="NVL Command and Control"                1
                source="ccu,vehicle">                                              2
        <field name="Command" abbrev="command" type="uint8_t"                      3
                 unit="Enumerated"       prefix="NVL">                             4
        <description>NVL command.</description>                                    5
                <value id="0" name="Start" abbrev="START" />                       6
                <value id="1" name="Stop" abbrev="STOP" />                         7
                <value id="2" name="List" abbrev="LIST" />                         8
        </field>                                                                   9
        <field name="Program" abbrev="program_id" type="plaintext" />              10
        <field name="Role" abbrev="role_id" type="plaintext" />                    11
    <field name="Participants" abbrev="participants_id" type="plaintext" />        12
</message>                                                                         13
                                                                                   14
<message id="833" abbrev="NVLNodeState" name="NVL Node State"                      15
                source="vehicle">                                                  16
        <field name="Role" abbrev="role_id" type="plaintext" />                    17
        <field name="State" abbrev="state_id" type="uint8_t"                       18
                unit="Enumerated" prefix="NVL">                                    19
        <description>Vehicle state.      </description>                            20
                <value id="0" name="Executing" abbrev="EXECUTING" />               21
                <value id="1" name="Ready" abbrev="READY" />                       22
                <value id="2" name="Notready" abbrev="NOTREADY" />                 23
        </field>                                                                   24
        <field name="Program" abbrev="program_id" type="plaintext" />              25
        <field name="Step Id" abbrev="step_id" type="plaintext" />                 26
        <field name="Stage" abbrev="stage_id" type="uint8_t"                       27
                unit="Enumerated" prefix="NVL">                                    28
        <description>Vehicle result while executing a step.</description>         29
                <value id="0" name="Control" abbrev="CONTROL" />                   30
                <value id="1" name="Gather" abbrev="GATHER" />                     31
                <value id="2" name="Reduce" abbrev="REDUCE" />                     32
                <value id="3" name="Done" abbrev="DONE" />                         33
        </field>                                                                   34
        <field name="Result" abbrev="result_id" type="plaintext" />                35
    <field name="System Type" abbrev="sys_type" type="uint8_t"                     36
        unit="Enumerated" enum-def="SystemType">                                   37
    <description>System type.</description>                                        38
    </field>                                                                       39
</message>                                                                         40
```

Figura 5.12: Mensagens IMC: NVLCommand e NVLNodeState

A mensagem NVLNodeState é usada pelo interpretador para monitorização do estado do veículo. Esta mensagem é criada pelo *ExternalInterface* e contém o campo Role (linha 17) que identifica o papel do veículo no controlador em execução, o campo State (linha 18) que identifica qual o estado do veículo (Em execução [Executing], disponível [Ready] ou indisponível [Notready])), o campo Program (linha 25) identifica qual o controlador em execução, o campo Step_Id (linha 26) identifica em qual passo, na execução do controlador, se encontra o veículo, o campo Stage (linha 27) identifica a fase em que o veículo se encontra durante a execução do passo, Result (linha 35) identifica o resultado com o qual o veículo terminou a execução do passo e por fim o campo System_Type identifica qual o tipo de veículo (UUV, UAV, ASV, ROV, etc).

Capítulo 6

Resultados experimentais

Neste capítulo são apresentados alguns resultados obtidos da execução de diversos cenários definidos por programas NVL, em simulação e em ambiente real de execução com veículos reais.

Foram executados alguns testes de sanidade, tais como verificação de deteção de erro dos controladores, testes de conectividade entre veículos e entre outros, tanto em modo de simulação como em ambiente real, e todos foram executados com sucesso.

6.1 Cenário 1 - Rendezvous

Neste cenário, existe a necessidade de efetuar uma recolha de dados por parte de um UUV, sem que este possua obrigatoriamente conectividade com outros veículos durante um certo período de tempo, após o qual irá reunir com um veículo disponível para efetuarem uma troca de dados.

6.1.0.1 Simulação

Para simular os veículos pretendidos para a execução deste cenário, foi utilizado o software DUNE (Secção 2.1.3.2), e todo o progresso do cenário em execução foi observado com recurso à ferramenta Neptus (Secção 2.1.3.3).

Neste cenário são necessários dois veículos e para tal foram executadas duas instâncias do DUNE, em consolas *unix* diferentes, sendo que o veículo 'lauv-xplore-1' representa o veículo que faz a recolha de dados (survey) numa dada área e o veículo 'lauv-seacon-1' serve para transportar dados (mule) recolhidos pelo primeiro veículo assim que este complete a sua tarefa.

Iniciar o veículo 'lauv-xplore-1' em simulação:

```
./dune -p Simulation -c lauv-xplore-1
```

Iniciar o veículo 'lauv-seacon-1' em simulação:

```
./dune -p Simulation -c lauv-seacon-1
```

Figura 6.1: Planos de *rendezvous* para os dois veículos, no Neptus

De seguida, com os dois veículos já em modo de simulação foi executada a NVL com os seguintes comandos:

Iniciar a NVL no veículo 'lauv-xplore-1':

```
$ nvl_runtime/scripts/rnvl.sh lauv-xplore-1
```

Iniciar a NVL no veículo 'lauv-seacon-2':

```
$ nvl_runtime/scripts/rnvl.sh lauv-seacon-2
```

O interpretador foi iniciado com o ficheiro relativo ao cenário *rendezvous* em anexo A.1.

```
$ nvl_runtime/scripts/interpreter.sh ../cen/RV_twoVehicles.nvl
```

Ambos os veículos executaram o cenário ilustrado na Figura 6.2, com os tempos de execução a serem apresentados na Figura 6.3. Em anexo são apresentados os registos da execução: do interpretador, do 'lauv-xplore-1' e do 'lauv-seacon-1'.

Na Figura 6.4 é possível identificar a área onde foi feita a recolha de dados pelo

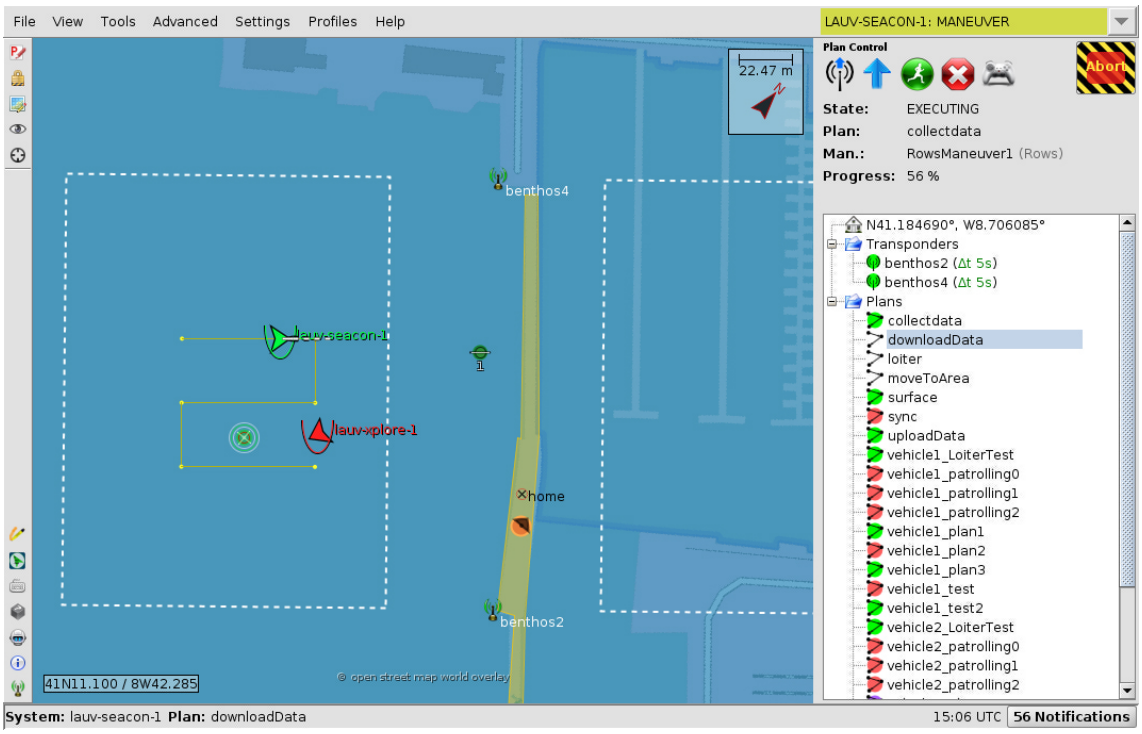


Figura 6.2: Captura de ecrã durante a execução do cenário de *rendezvous*

Instrução	Duração
select	0:20
spawn collectData	4:33
spawn moveToOpArea	1:09
join cRes mRes	4:33
delay 15s	0:15
spawn RV	2:21
join rvRes	2:21
Programa RV	8:29

Figura 6.3: Durações na execução das instruções do programa de *rendezvous*

veículo 'lauv-seacon-1' (a vermelho), e caminho percorrido pelo veículo 'lauv-xplore-1' (tracejado azul) para se deslocar para perto da área de recolha de dados.

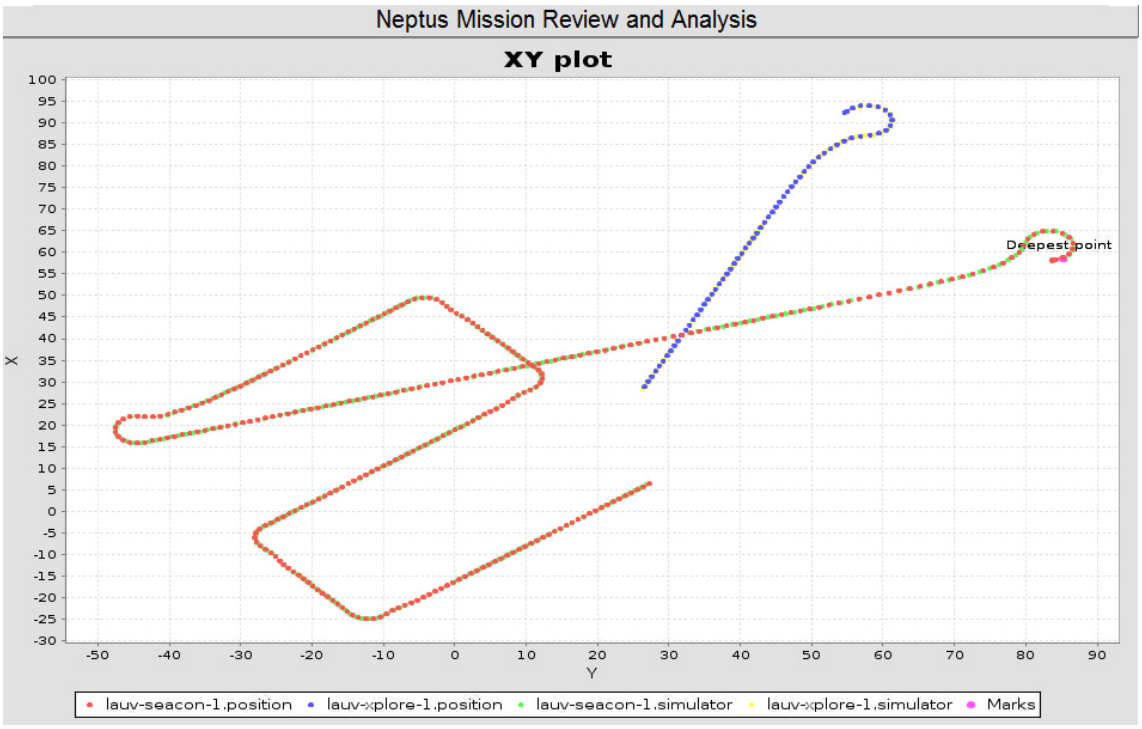


Figura 6.4: Área percorrida no cenário de *rendezvous*

6.2 Cenário 2 - Patrulha

Neste cenário existe uma área extensa, da qual é necessário recolher dados e utilizar apenas um veículo demoraria demasiado tempo para realizar a operação. Desta forma, recorre-se ao uso de três veículos para dividir a área em questão e proceder à recolha de dados de forma dividida.

6.2.1 Simulação

Tal como no cenário anterior, foi utilizado o DUNE para simulação dos veículos e o progresso do cenário em execução foi observado com recurso à ferramenta Neptus.

Foram executadas três instâncias do DUNE, uma para cada um dos três veículos. O veículo 'lauv-seacon-1' representa o veículo que faz a recolha de dados da área Sul, o veículo 'lauv-seacon-2' recolhe na área Centro e o veículo 'lauv-xplore-1' efetua a recolha na área Norte (Figura 6.5)

Iniciar o veículo 'lauv-xplore-1' em simulação:

```
./dune -p Simulation -c lauv-xplore-1
```

Iniciar o veículo 'lauv-seacon-2' em simulação:

```
./dune -p Simulation -c lauv-seacon-2
```

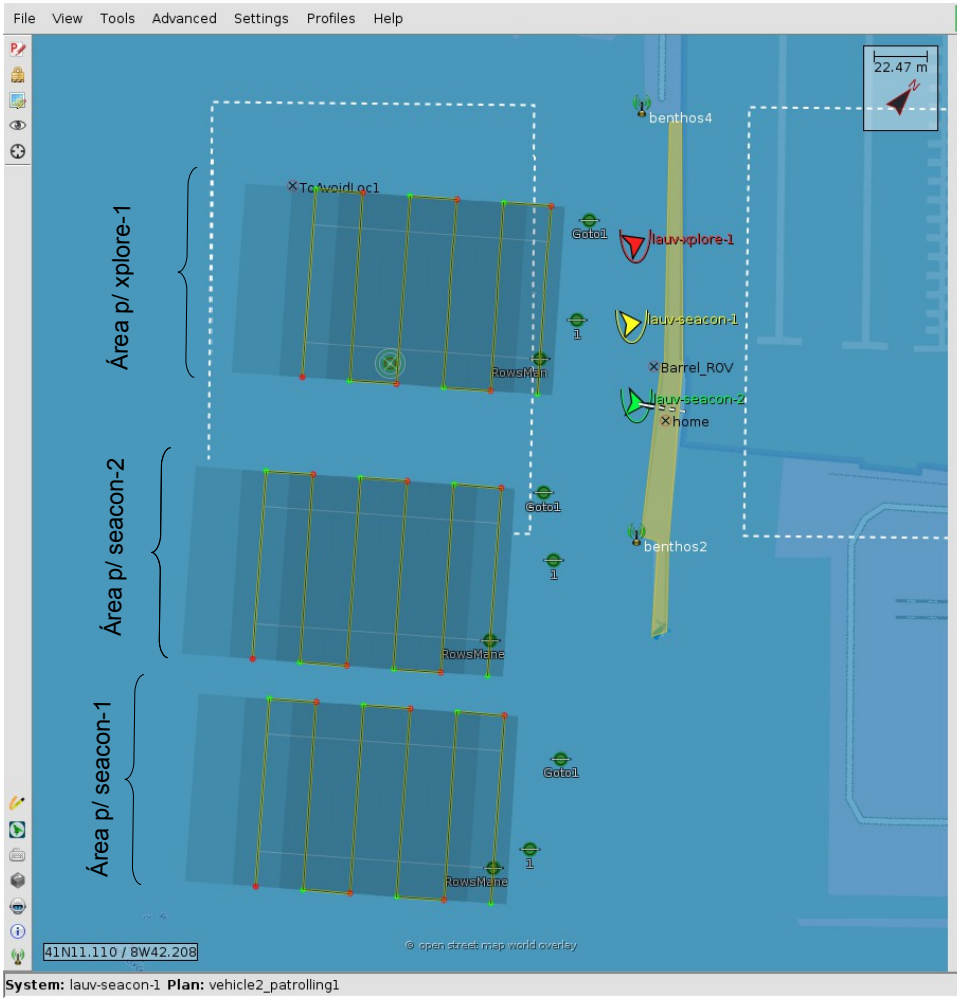


Figura 6.5: Planos de patrulha para os três veículos, no Neptus

Iniciar o veículo 'lauv-seacon-1' em simulação:

```
./dune -p Simulation -c lauv-seacon-1
```

De seguida, com os três veículos já em modo de simulação foi executada a NVL com os seguintes comandos:

Iniciar a NVL no veículo 'lauv-xplore-1':

```
$ nvl_runtime/scripts/rnvl.sh lauv-xplore-1
```

Iniciar a NVL no veículo 'lauv-seacon-2':

```
$ nvl_runtime/scripts/rnvl.sh lauv-seacon-2
```

Iniciar a NVL no veículo 'lauv-seacon-1':

```
$ nvl_runtime/scripts/rnvl.sh lauv-seacon-1
```

De seguida noutra consola, foi executado o interpretador com o comando apresentado a seguir, para executar o interpretador e carregar o ficheiro correspondente ao cenário de patrulha.

Execução do interpretador com cenário de patrulha em anexo A.2.

```
$ nvl_runtime/scripts/interpreter.sh ../cen/patrolling.nvl
```

Foi verificado que os veículos envolvidos executaram o cenário em ambiente de simulação com sucesso e iniciou-se portanto uma execução em ambiente real, que abordamos no próximo capítulo.

6.2.2 Experiência de campo

Com o objetivo de testar a NVL em ambiente real, foram cedidos dois veículos e uma Manta pelo LSTS. Os veículos utilizados foram o 'lauv-seacon-2' e 'lauv-xplore-1' (Figura 6.6).

Figura 6.6: Veículos lauv-seacon-2 e lauv-xplore-1

Especificações dos veículos:
Profundidade máxima: 100 m
Autonomia: até 8 horas @ 3 nós
Velocidade: até 4 nós
Comunicações: WiFi, GSM/HSDPA
Navegação: GPS, AHRS, Sensor profundidade

Para estabelecer as ligações entre operador e veículos foi utilizado um *gatweway* de comunicações, a Manta, que suporta comunicações sem fios wifi e via modems acústicos. A Manta permite que vários operadores controlem e monitorizem os diversos veículos na rede, suportando diferentes plataformas.

Figura 6.7: Manta - *Gateway* de comunicações

Especificações da manta:
Dimensões: 43 x 25 x 34 cm
Peso: 9Kg
Autonomia: 8 horas
Comunicações: WiFi 2.4 e 5 GHz (alcance até 4,5 km), GSM/HSDPA
Comunicação por via acústica: Micro-modem (alcance até 2 km)
Posicionamento: GPS

O local escolhido foi para a realização da missão de teste foi a APDL (Administração dos Portos de Douro e Leixões), no dia 3 de Setembro de 2014.

Para realizar as experiências de campo foram instaladas *BeagleBone's Black* nos dois veículos, onde posteriormente foi carregado todo o sistema NVL. A *BeagleBone Black* é um computador de baixo custo com requisitos de energia reduzidos, equipada com um processador ARM® Cortex A8 a 1GHz, 512MB de memória RAM, 2GB de memória flash, saída de vídeo e audio, ligação *ethernet*, pinos de I/O, desenvolvida pela Texas Instruments. Esta versão surgiu de uma melhoria à original *Beagleboard* que foi desenvolvida por um grupo de engenheiros como uma placa educativa que poderia ser utilizada em faculdades como plataforma de desenvolvimento e aprendizagem em sistemas embebidos, baseada num modelo *open source*.

Para o sistema NVL ser executado é necessário o Java, para tal foi instalado ambiente Linux em todas as placas *BeagleBone* e também o Java JRE-1.7 de forma a ser possível carregar e executar o sistema NVL nos veículos.

Para transferir o sistema NVL para os veículos, efetuou-se uma ligação *SSH* aos veículos e copiado todo o pacote NVL-*runtime* para o sistema de cada um dos veículos.

Neste cenário foram utilizados três veículos, dois reais (lauv-seacon-2 e lauv-xplore-1) e um em simulação (lauv-seacon-1). Usando uma ligação SSH a cada um dos veículos reais, o sistema NVL foi iniciado com os comandos apresentados a seguir, cada um em uma consola *unix* diferente.

Iniciar a NVL no veículo 'lauv-seacon-2':

```
$ nvl_runtime/scripts/rnvl.sh lauv-seacon-2
```

Iniciar a NVL no veículo 'lauv-xplore-1':

```
$ nvl_runtime/scripts/rnvl.sh lauv-xplore-1
```

O veículo 'lauv-seacon-1' foi iniciado em modo de simulação da seguinte forma:

```
./dune -p Simulation -c lauv-xplore-1
```

E de seguida iniciada a NVL para o 'lauv-seacon-1':

```
$ nvl_runtime/scripts/rnvl.sh lauv-seacon-1
```

De seguida noutra consola, foi executado o interpretador com o comando apresentado a seguir, para executar o interpretador e carregar o ficheiro correspondente ao cenário de patrulha.

Execução do interpretador com cenário de patrulha em anexo A.2.

```
$ nvl_runtime/scripts/interpreter.sh ../cen/patrolling.nvl
```

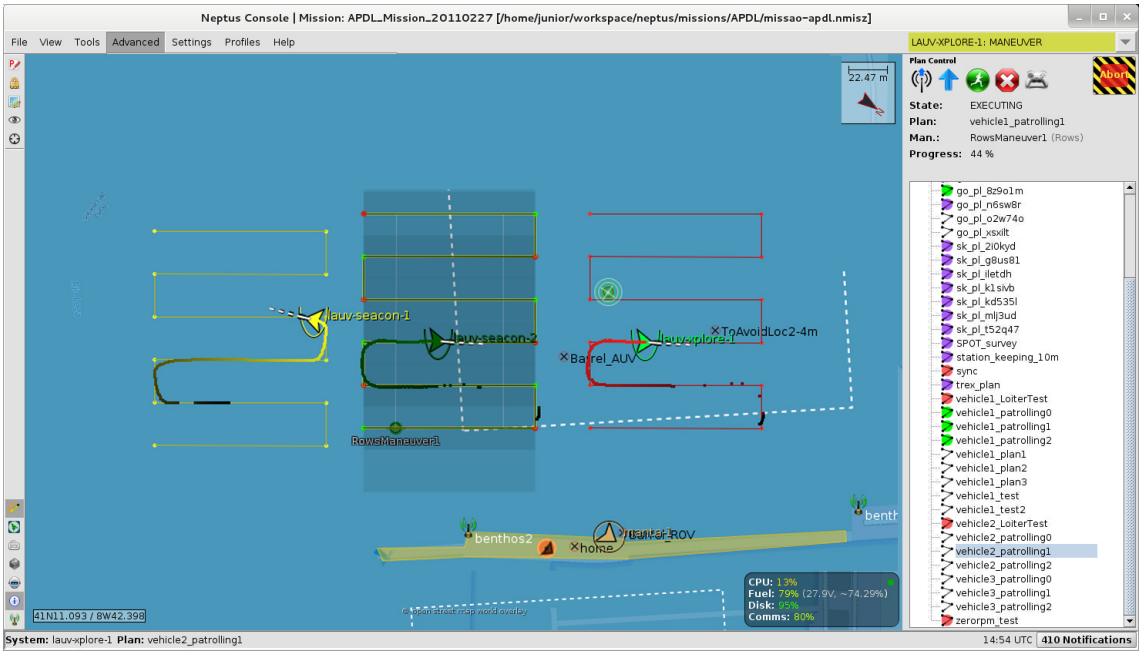


Figura 6.8: Captura de ecrã durante a execução do cenário de patrulha

O conjunto de veículos selecionado (2 reais + 1 simulado) executaram com sucesso o cenário de patrulha (Figura 6.8), com os tempos de execução a serem apresentados na Figura 6.9. Em anexo são apresentados os registos da execução: do interpretador, do 'lauv-xplore-1', do 'lauv-seacon-1' e do 'lauv-seacon-2'.

Na Figura 6.10 é possível identificar as três áreas onde foi efetuada a patrulha. A área a verde representa a área patrulhada pelo veículo 'lauv-xplore-1', a azul a área do 'lauv-seacon-2' e por fim a vermelho a área patrulha por 'lauv-seacon-1'.

Instrução	Duração
select	0:11
spawn Patrolling3	9:11
join cRes mRes	9:11
delay 10s	0:10
Programa Patrulha	9:34

Figura 6.9: Durações na execução das instruções do programa de patrulha

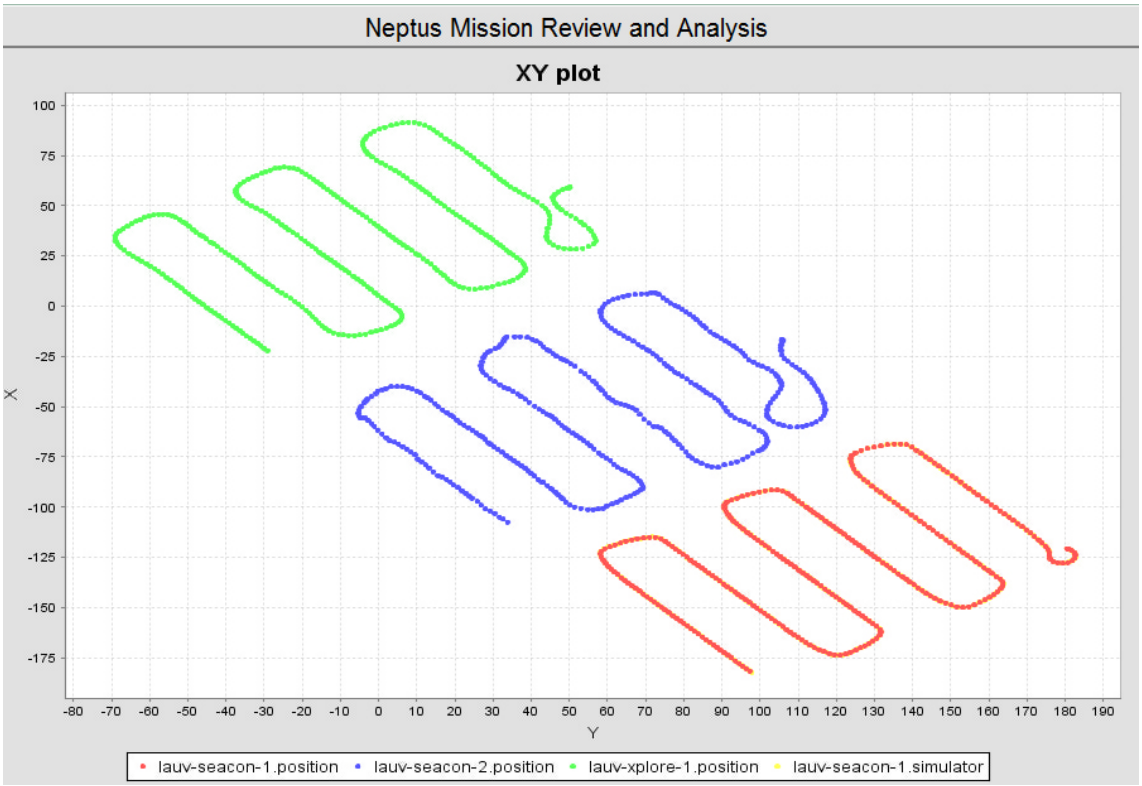


Figura 6.10: Varrimento de área realizado pelos três veículos

Foi também observada a carga de utilização do processador da *BeagleBone* em *idle* que se situava entre os 53% e os 55%. Já durante a execução dos programas NVL a carga de utilização do processador subiu para os 58%, considerando-se portanto que a NVL não sobrecarrega o processador.

Capítulo 7

Conclusão

Esta tese apresentou a linguagem NVL para coordenação de redes de veículos autónomos, em termos da sua definição, desenho, implementação, e avaliação experimental. Embora a linguagem se encontre num estado de protótipo inicial, e muito trabalho se possa considerar no futuro (discutido abaixo), salientam-se os seguintes aspectos:

- A NVL é uma linguagem que permite exprimir cenários de operação multi-veículo com um alto nível de abstracção, endereçando requisitos não-triviais na sua especificação, e provendo à sua execução.
- A NVL envolveu um esforço de desenho e implementação também não-trivial, compreendendo um ambiente de edição/validação de programas, e outro para execução distribuída dos mesmos em interface com o toolchain LSTS.
- Consideramos que a avaliação experimental da linguagem decorreu com sucesso, incluindo experiências de campo com veículos reais. Embora o número de cenários de teste em si tenha sido reduzido, cremos que se logrou o objectivo de demonstrar o potencial inerente à aproximação geral proposta pela tese, e a efectividade do sistema de software desenvolvido.

7.1 Trabalho futuro

Para trabalho futuro na NVL identificamos os seguintes desafios:

- Extensão e refinamento da linguagem: a modelação de cenários multi-veículo coloca requisitos de vária ordem que não são endereçados na linguagem. Por exemplo, tomando como referência a modelação e simulação de cenários complexos em [23], a linguagem não tem abstracções para equipas de veículos, o acoplamento dinâmico de veículos a controladores (tal acontece apenas quando um controlador é activado), ou para hierarquia de controlo.

- Colocam-se problemas típicos de sistemas de distribuidos. como modelar e implementar apropriadamente tolerância a falhas, eventualmente com suporte dado pela própria linguagem. Outro problema reside na execução centralizada de um programa por um interpretador NVL. Não é trivial distribuir a execução do interpretador, quando estão em causa requisitos de sincronização (ex. na selecção de veículos, término de controladores).

- Uma maior integração do software com o toolchain LSTS é necessária, nomeadamente o acoplamento dos ambientes de edição/execução como plugin para a ferramenta Neptus.

Apêndice A

Programas NVL

A.1 Rendezvous

```
controller collectData {
  vehicles survey;
  outputs done, rvError;

  step start {
  control 15m {
        at survey collectdata ;
  }

  supervise {}

  synchronize 100s {

        stop done <- survey:done ;
    stop rvError <- default;

  }
}

}

controller moveToOpArea {
  vehicles mule;
  outputs done rvError;

  step start {
  control 5m {
        at mule moveToArea ;
  }

  supervise {}

  synchronize 100s {

        stop done <- mule:done;
    stop rvError <- default;

  }
}
}
controller RV {
 vehicles survey, mule;
 outputs moreData, rvDone, rvError;

 step start {
   control 15m {
     at survey surface;
     at mule loiter ;
```

```
    }
    supervise {}

    synchronize 2m {
     dataTransfer <- mule:done survey:done ;
     stop rvError <- default;
    }
  }
  step dataTransfer {
   control 10m {
    at survey uploadData;
    at mule downloadData;
   }
   supervise {}

   synchronize 2m{
     stop rvDone <- mule:done survey:done;
     stop moreData <- mule:done survey:getMoreData;
     stop rvError <- default;
   }
  }
}
// program procedures
procedure main {
 // select vehicles
 select 5m
  uuv ( type: "UUV",
          id: "lauv-seacon-1"),
  uav ( type: "UUV"
          id: "lauv-xplore-1"
  )
 then {
  // spawn survey controller for uuv
  spawn cRes collectData (survey: uuv)
  // spawn moveToOpArea controller for uav
  spawn mRes moveToOpArea (mule: uav)
  join 1h cRes mRes
  delay 15s
  // spawn RV controller for uuv and uav
  spawn rvRes RV(survey: uuv, mule: uav)
  join 15m rvRes
  // evaluate controller output
  choose rvRes {
   case moreData {
    // execute again
    continue main
   }
   default {
    // do nothing
   }
  }
 }
 or {
  // selection failed, retry 1 hour later
  delay 1h
  continue main
 }
 // end the program
 message "terminated"
}
```

A.2 Patrulha

```
controller Patrolling3 {
  vehicles a b c;
  outputs done error;
  subsystems {}

  step start {
    control 180s {
      at a vehicle1_patrolling0 ;
      at b vehicle2_patrolling0 ;
      at c vehicle3_patrolling0 ;
    }

    supervise { }

    synchronize 60s {
      //a sync0;  //to define other syncplan
      step2 <- a:done b:done c:done;
      start <- a:timeout;
      start <- b:timeout;
      start <- c:timeout;
      stop error <- default;
    }
  }
  step step2 {
    control 600s {
      at a vehicle1_patrolling1 ;
      at b vehicle2_patrolling1 ;
      at c vehicle3_patrolling1 ;
    }

    supervise { }

    synchronize 60s {
      rendezvous <- a:done b:done c:done;
      step2 <- a:timeout;
      step2 <- b:timeout;
      step2 <- c:timeout;
      stop error <- default;
    }
  }
  step rendezvous {
    control 120 s {
      at a vehicle1_patrolling2 ;
      at b vehicle2_patrolling2 ;
      at c vehicle3_patrolling2 ;
    }
    supervise { }

    synchronize 60s {
      stop done <- a:done b:done;
      rendezvous <- a:timeout;
      rendezvous <- b:timeout;
      rendezvous <- c:timeout;
      stop error <- default;
    }
  }
}

procedure main {
  select 35s
    auv0 ( type: "UUV", id: "lauv-xplore-1")
    auv1 ( type: "UUV", id: "lauv-seacon-2" )
    auv2 ( type: "UUV", id: "lauv-seacon-1")
  then {
    spawn r1 Patrolling3 ( a:auv0, b:auv1, c:auv2 )
    join 1000s r1
    choose r1 {
      case error {
```

```
        message "error"
      }
      case done {
        message "done"
      }
    }
    release auv0 auv1 auv2
    delay 10s
    message "end patrolling"
  }
}
```

Apêndice B

Registos de execução

B.1 Rendezvous

B.1.1 Interpretador

```
27 Sep 2014 15:23:12 GMT | Interpreter : Executing 'select' instruction ...
27 Sep 2014 15:23:32 GMT | Interpreter : Vechicle 'ccu-nvl-lauv-xplore-1' bound to 'uav'
    ...
27 Sep 2014 15:23:32 GMT | Interpreter : Vechicle 'ccu-nvl-lauv-seacon-1' bound to 'uuv'
    ...
27 Sep 2014 15:23:32 GMT | Interpreter : Executing 'spawn' instruction ...
27 Sep 2014 15:23:32 GMT | Interpreter : Executing 'spawn' instruction ...
27 Sep 2014 15:23:32 GMT | Interpreter : Executing 'join' instruction ...
27 Sep 2014 15:29:05 GMT | Interpreter : Sending stop command to ccu-nvl-lauv-seacon-1
27 Sep 2014 15:29:05 GMT | Interpreter : Sending stop command to ccu-nvl-lauv-xplore-1
27 Sep 2014 15:29:05 GMT | Interpreter : Join results: '{cRes=done, mRes=done}'
27 Sep 2014 15:29:05 GMT | Interpreter : Executing 'delay' instruction ...
27 Sep 2014 15:29:20 GMT | Interpreter : Executing 'spawn' instruction ...
27 Sep 2014 15:29:20 GMT | Interpreter : Executing 'join' instruction ...
27 Sep 2014 15:31:41 GMT | Interpreter : Sending stop command to ccu-nvl-lauv-seacon-1
27 Sep 2014 15:31:41 GMT | Interpreter : Sending stop command to ccu-nvl-lauv-xplore-1
27 Sep 2014 15:31:41 GMT | Interpreter : Join results: '{cRes=done, rvRes=rvDone, mRes=
    done}'
27 Sep 2014 15:31:41 GMT | Interpreter : Executing 'choose' instruction ...
27 Sep 2014 15:31:41 GMT | Interpreter : Released 'uuv' (vehicle 'ccu-nvl-lauv-seacon
    -1').
27 Sep 2014 15:31:41 GMT | Interpreter : Released 'uav' (vehicle 'ccu-nvl-lauv-xplore
    -1').
27 Sep 2014 15:31:41 GMT | Interpreter : Executing 'message' instruction ...
27 Sep 2014 15:31:41 GMT | Interpreter : terminated
27 Sep 2014 15:31:41 GMT | Interpreter : Will terminate: Program terminated
27 Sep 2014 15:31:41 GMT | InterpreterLink : InterpreterLink is dead.
27 Sep 2014 15:31:41 GMT | Announcer : Announcer is dead.
27 Sep 2014 15:31:41 GMT | Heart : Heart is dead.
27 Sep 2014 15:31:41 GMT | ShutdownThread : Program termination requested.
```

B.1.2 Veículo *lauv-xplore-1*

```
27 Sep 2014 15:23:31 GMT | ExternalInterface : Activating transmission to node '
    interpreter-nvl'
27 Sep 2014 15:23:32 GMT | ExternalInterface : [Vehicle] lauv-xplore-1 - READY [READY]
27 Sep 2014 15:23:32 GMT | ExternalInterface : Node defined as 'mule' ...
27 Sep 2014 15:23:32 GMT | ExternalInterface : Requesting 'moveToOpArea_mule' to start
    ...
27 Sep 2014 15:23:32 GMT | Heart : Launching organ 'moveToOpArea_mule' ...
27 Sep 2014 15:23:32 GMT | Heart : Creating organ 'moveToOpArea_mule' ...
27 Sep 2014 15:23:32 GMT | Heart : Organ 'moveToOpArea_mule' has been started.
27 Sep 2014 15:23:32 GMT | moveToOpArea_mule : Node: mule | Vehicle: lauv-xplore-1 |
    Address: 30
27 Sep 2014 15:23:32 GMT | moveToOpArea_mule : moveToOpArea_mule is born.
27 Sep 2014 15:23:32 GMT | moveToOpArea_mule : > Executing step: 'start'
27 Sep 2014 15:23:32 GMT | moveToOpArea_mule : control deadline 300 - sync deadline 100
27 Sep 2014 15:24:41 GMT | moveToOpArea_mule : R: done
27 Sep 2014 15:24:41 GMT | moveToOpArea_mule : Sync. stage over -- normal completion; 69
    seconds execution
27 Sep 2014 15:24:41 GMT | moveToOpArea_mule : Results: {mule=done}
27 Sep 2014 15:24:41 GMT | moveToOpArea_mule : > Execution done; next step is 'stop'
27 Sep 2014 15:24:41 GMT | moveToOpArea_mule : > Stopped with result: done ...
27 Sep 2014 15:24:41 GMT | moveToOpArea_mule : > End controller ...
```

B.1.3 Veículo *lauv-seacon-1*

```
27 Sep 2014 15:23:31 GMT | ExternalInterface : Activating transmission to node '
    interpreter-nvl'
27 Sep 2014 15:23:32 GMT | ExternalInterface : Node defined as 'survey' ...
27 Sep 2014 15:23:32 GMT | ExternalInterface : Requesting 'collectData_survey' to start
    ...
27 Sep 2014 15:23:32 GMT | Heart : Launching organ 'collectData_survey' ...
27 Sep 2014 15:23:32 GMT | Heart : Creating organ 'collectData_survey' ...
27 Sep 2014 15:23:32 GMT | Heart : Organ 'collectData_survey' has been started.
27 Sep 2014 15:23:32 GMT | collectData_survey : Node: survey | Vehicle: lauv-seacon-1 |
    Address: 21
27 Sep 2014 15:23:32 GMT | collectData_survey : collectData_survey is born.
27 Sep 2014 15:23:32 GMT | collectData_survey : > Executing step: 'start'
27 Sep 2014 15:23:32 GMT | collectData_survey : control deadline 900 - sync deadline 100
27 Sep 2014 15:29:05 GMT | collectData_survey : R: done
27 Sep 2014 15:29:05 GMT | collectData_survey : Sync. stage over -- normal completion;
    333 seconds execution
27 Sep 2014 15:29:05 GMT | collectData_survey : Results: {survey=done}
27 Sep 2014 15:29:05 GMT | collectData_survey : > Execution done; next step is 'stop'
27 Sep 2014 15:29:05 GMT | collectData_survey : > Stopped with result: done ...
27 Sep 2014 15:29:05 GMT | collectData_survey : > End controller ...
```

B.2 Patrulha

B.2.1 Interpretador

```
3 Sep 2014 16:48:02 GMT | main : Parsing configuration ...
3 Sep 2014 16:48:02 GMT | main : Creating organ 'Announcer' ...
3 Sep 2014 16:48:02 GMT | main : Interface vehicle: 'interpreter-nvl'.
3 Sep 2014 16:48:02 GMT | main : Shared object registered: 'pt.lsts.nvl.organ.imc.
    IMCInformation' of type 'class pt.lsts.nvl.organ.imc.IMCInformation'.
3 Sep 2014 16:48:02 GMT | main : Creating organ 'Interpreter' ...
3 Sep 2014 16:48:02 GMT | main : Creating organ 'InterpreterLink' ...
3 Sep 2014 16:48:03 GMT | Heart : Organ 'Announcer' has been started.
3 Sep 2014 16:48:03 GMT | Heart : Organ 'Interpreter' has been started.
3 Sep 2014 16:48:03 GMT | Heart : Organ 'InterpreterLink' has been started.
3 Sep 2014 16:48:03 GMT | Heart : Heart is born.
3 Sep 2014 16:48:03 GMT | Announcer : Trying to create socket on port 30102 ...
3 Sep 2014 16:48:03 GMT | InterpreterLink : Trying to create socket on port 10005 ...
3 Sep 2014 16:48:03 GMT | InterpreterLink : Bound to port 10005.
3 Sep 2014 16:48:03 GMT | Announcer : Trying to create socket on port 30103 ...
3 Sep 2014 16:48:03 GMT | Announcer : Trying to create socket on port 30104 ...
3 Sep 2014 16:48:03 GMT | Announcer : Bound to port 30104.
3 Sep 2014 16:48:03 GMT | Announcer : Announcer is born.
3 Sep 2014 16:48:03 GMT | InterpreterLink : InterpreterLink is born.
3 Sep 2014 16:48:03 GMT | Announcer : Node 'lauv-xplore-1' is alive at 127.0.0.1:6004
3 Sep 2014 16:48:04 GMT | Interpreter : ----Structure Validation----
3 Sep 2014 16:48:04 GMT | Interpreter : Errors : 0
3 Sep 2014 16:48:04 GMT | Interpreter : Warnings : 0
3 Sep 2014 16:48:04 GMT | Interpreter : ----------
3 Sep 2014 16:48:04 GMT | Interpreter : ----Model Validation----
3 Sep 2014 16:48:04 GMT | Interpreter : ----------
3 Sep 2014 16:48:04 GMT | Interpreter : Interpreter is born.
3 Sep 2014 16:48:04 GMT | Interpreter : Executing procedure 'main' ...
3 Sep 2014 16:48:04 GMT | Interpreter : Executing 'select' instruction ...
3 Sep 2014 16:48:04 GMT | Announcer : Node 'ccu-nvl-local-lauv-seacon-2' is alive at
    192.168.1.11:9002
3 Sep 2014 16:48:04 GMT | Announcer : Node 'ccu-nvl-lauv-seacon-2' is alive at
    192.168.1.11:10002
3 Sep 2014 16:48:05 GMT | Announcer : Node 'ccu-junior-1-11' is alive at
    192.168.1.11:6001
3 Sep 2014 16:48:06 GMT | Announcer : Node 'ccu-nvl-local-lauv-seacon-1' is alive at
    192.168.1.11:9003
3 Sep 2014 16:48:06 GMT | Announcer : Node 'ccu-nvl-lauv-seacon-1' is alive at
    192.168.1.11:10003
3 Sep 2014 16:48:06 GMT | Announcer : Node 'ccu-nvl-lauv-xplore-1' is alive at
    192.168.1.11:10001
3 Sep 2014 16:48:06 GMT | Announcer : Node 'ccu-nvl-local-lauv-xplore-1' is alive at
    192.168.1.11:9001
3 Sep 2014 16:48:09 GMT | Announcer : Node 'lauv-seacon-1' is alive at 127.0.0.1:6002
3 Sep 2014 16:48:12 GMT | Announcer : Node 'lauv-seacon-2' is alive at 127.0.0.1:6003
3 Sep 2014 16:48:12 GMT | Announcer : Node 'interpreter-nvl' is alive at
    192.168.1.11:10005
3 Sep 2014 16:48:13 GMT | Interpreter : Vechicle 'ccu-nvl-lauv-seacon-1' bound to 'auv2'
    ...
3 Sep 2014 16:48:14 GMT | Interpreter : Vechicle 'ccu-nvl-lauv-xplore-1' bound to 'auv0'
    ...
3 Sep 2014 16:48:15 GMT | Interpreter : Vechicle 'ccu-nvl-lauv-seacon-2' bound to 'auv1'
    ...
3 Sep 2014 16:48:15 GMT | Interpreter : Executing 'spawn' instruction ...
3 Sep 2014 16:48:15 GMT | Interpreter : Executing 'join' instruction ...
3 Sep 2014 16:57:26 GMT | Interpreter : Sending stop command to ccu-nvl-lauv-xplore-1
3 Sep 2014 16:57:26 GMT | Interpreter : Sending stop command to ccu-nvl-lauv-seacon-2
3 Sep 2014 16:57:26 GMT | Interpreter : Sending stop command to ccu-nvl-lauv-seacon-1
3 Sep 2014 16:57:26 GMT | Interpreter : Join results: '{r1=done}'
3 Sep 2014 16:57:26 GMT | Interpreter : Executing 'choose' instruction ...
3 Sep 2014 16:57:26 GMT | Interpreter : Executing 'message' instruction ...
3 Sep 2014 16:57:26 GMT | Interpreter : done
3 Sep 2014 16:57:26 GMT | Interpreter : Executing 'release' instruction ...
3 Sep 2014 16:57:26 GMT | Interpreter : Released 'auv0' (vehicle 'ccu-nvl-lauv-xplore
    -1')
```

```
3 Sep 2014 16:57:26 GMT | Interpreter : Released 'auv1' (vehicle 'ccu-nvl-lauv-seacon
    -2')
3 Sep 2014 16:57:26 GMT | Interpreter : Released 'auv2' (vehicle 'ccu-nvl-lauv-seacon
    -1')
3 Sep 2014 16:57:26 GMT | Interpreter : Executing 'delay' instruction ...
3 Sep 2014 16:57:36 GMT | Interpreter : Executing 'message' instruction ...
3 Sep 2014 16:57:36 GMT | Interpreter : end patrolling
3 Sep 2014 16:57:36 GMT | Interpreter : Will terminate: Program terminated
3 Sep 2014 16:57:36 GMT | Announcer : Announcer is dead.
3 Sep 2014 16:57:36 GMT | InterpreterLink : InterpreterLink is dead.
3 Sep 2014 16:57:36 GMT | Heart : Heart is dead.
3 Sep 2014 16:57:36 GMT | ShutdownThread : Program termination requested.
```

B.2.2 Veículo 'lauv-seacon-1'

```
3 Sep 2014 16:48:12 GMT | ExternalInterface : Activating transmission to node '
    interpreter-nvl'
3 Sep 2014 16:48:13 GMT | ExternalInterface : [Vehicle] lauv-seacon-1 - READY [READY]
3 Sep 2014 16:48:15 GMT | ExternalInterface : Node defined as 'c' ...
3 Sep 2014 16:48:15 GMT | ExternalInterface : Requesting 'Patrolling3_c' to start ...
3 Sep 2014 16:48:15 GMT | Heart : Launching organ 'Patrolling3_c' ...
3 Sep 2014 16:48:15 GMT | Heart : Creating organ 'Patrolling3_c' ...
3 Sep 2014 16:48:15 GMT | Heart : Organ 'Patrolling3_c' has been started.
3 Sep 2014 16:48:15 GMT | Patrolling3_c : Node: c | Vehicle: lauv-seacon-1 | Address: 21
3 Sep 2014 16:48:15 GMT | Patrolling3_c : Patrolling3_c is born.
3 Sep 2014 16:48:15 GMT | Patrolling3_c : > Executing step: 'start'
3 Sep 2014 16:48:15 GMT | Patrolling3_c : control deadline 180 - sync deadline 60
3 Sep 2014 16:49:06 GMT | Patrolling3_c : R: done
3 Sep 2014 16:49:10 GMT | Patrolling3_c : FINAL STAGE
3 Sep 2014 16:49:15 GMT | Patrolling3_c : Sync. stage over -- normal completion; 59
    seconds execution
3 Sep 2014 16:49:15 GMT | Patrolling3_c : Results: {b=done, c=done, a=done}
3 Sep 2014 16:49:15 GMT | Patrolling3_c : > Execution done; next step is 'step2'
3 Sep 2014 16:49:15 GMT | Patrolling3_c : > Executing step: 'step2'
3 Sep 2014 16:49:15 GMT | Patrolling3_c : control deadline 600 - sync deadline 60
3 Sep 2014 16:54:54 GMT | Patrolling3_c : R: done
3 Sep 2014 16:55:20 GMT | Patrolling3_c : FINAL STAGE
3 Sep 2014 16:55:26 GMT | Patrolling3_c : Sync. stage over -- normal completion; 370
    seconds execution
3 Sep 2014 16:55:26 GMT | Patrolling3_c : Results: {b=done, c=done, a=done}
3 Sep 2014 16:55:26 GMT | Patrolling3_c : > Execution done; next step is 'rendezvous'
3 Sep 2014 16:55:26 GMT | Patrolling3_c : > Executing step: 'rendezvous'
3 Sep 2014 16:55:26 GMT | Patrolling3_c : control deadline 120 - sync deadline 60
3 Sep 2014 16:57:05 GMT | Patrolling3_c : R: done
3 Sep 2014 16:57:21 GMT | Patrolling3_c : FINAL STAGE
3 Sep 2014 16:57:26 GMT | Patrolling3_c : Sync. stage over -- normal completion; 120
    seconds execution
3 Sep 2014 16:57:26 GMT | Patrolling3_c : Results: {b=done, c=done, a=done}
3 Sep 2014 16:57:26 GMT | Patrolling3_c : > Execution done; next step is 'stop'
3 Sep 2014 16:57:26 GMT | Patrolling3_c : > Stopped with result: done ...
3 Sep 2014 16:57:26 GMT | Patrolling3_c : > End controller ...
3 Sep 2014 16:57:26 GMT | ExternalInterface : Stopped execution of running program.
3 Sep 2014 16:57:27 GMT | ExternalInterface : [Vehicle] lauv-seacon-1 - READY [READY]
3 Sep 2014 16:57:35 GMT | ShutdownThread : Program termination requested.
3 Sep 2014 16:57:35 GMT | Announcer : Announcer is dead.
3 Sep 2014 16:57:35 GMT | VehicleLink : VehicleLink is dead.
3 Sep 2014 16:57:35 GMT | ExternalInterface : ExternalInterface is dead.
3 Sep 2014 16:57:35 GMT | StatusDiffusion : StatusDiffusion is dead.
3 Sep 2014 16:57:35 GMT | Heart : Heart is dead.
```

B.2.3 Veículo 'lauv-seacon-2'

```
3 Sep 2014 16:48:12 GMT | ExternalInterface : Activating transmission to node '
    interpreter-nvl'
3 Sep 2014 16:48:15 GMT | ExternalInterface : [Vehicle] lauv-seacon-2 - READY [READY]
3 Sep 2014 16:48:15 GMT | ExternalInterface : Node defined as 'b' ...
3 Sep 2014 16:48:15 GMT | ExternalInterface : Requesting 'Patrolling3_b' to start ...
3 Sep 2014 16:48:15 GMT | Heart : Launching organ 'Patrolling3_b' ...
3 Sep 2014 16:48:15 GMT | Heart : Creating organ 'Patrolling3_b' ...
3 Sep 2014 16:48:15 GMT | Heart : Organ 'Patrolling3_b' has been started.
3 Sep 2014 16:48:15 GMT | Patrolling3_b : Node: b | Vehicle: lauv-seacon-2 | Address: 22
3 Sep 2014 16:48:15 GMT | Patrolling3_b : Patrolling3_b is born.
3 Sep 2014 16:48:15 GMT | Patrolling3_b : > Executing step: 'start'
3 Sep 2014 16:48:15 GMT | Patrolling3_b : control deadline 180 - sync deadline 60
3 Sep 2014 16:49:08 GMT | Patrolling3_b : R: done
3 Sep 2014 16:49:10 GMT | Patrolling3_b : FINAL STAGE
3 Sep 2014 16:49:15 GMT | Patrolling3_b : Sync. stage over -- normal completion; 59
    seconds execution
3 Sep 2014 16:49:15 GMT | Patrolling3_b : Results: {b=done, c=done, a=done}
3 Sep 2014 16:49:15 GMT | Patrolling3_b : > Execution done; next step is 'step2'
3 Sep 2014 16:49:15 GMT | Patrolling3_b : > Executing step: 'step2'
3 Sep 2014 16:49:15 GMT | Patrolling3_b : control deadline 600 - sync deadline 60
3 Sep 2014 16:55:18 GMT | Patrolling3_b : R: done
3 Sep 2014 16:55:20 GMT | Patrolling3_b : FINAL STAGE
3 Sep 2014 16:55:25 GMT | Patrolling3_b : Sync. stage over -- normal completion; 369
    seconds execution
3 Sep 2014 16:55:25 GMT | Patrolling3_b : Results: {b=done, c=done, a=done}
3 Sep 2014 16:55:25 GMT | Patrolling3_b : > Execution done; next step is 'rendezvous'
3 Sep 2014 16:55:25 GMT | Patrolling3_b : > Executing step: 'rendezvous'
3 Sep 2014 16:55:25 GMT | Patrolling3_b : control deadline 120 - sync deadline 60
3 Sep 2014 16:57:19 GMT | Patrolling3_b : R: done
3 Sep 2014 16:57:21 GMT | Patrolling3_b : FINAL STAGE
3 Sep 2014 16:57:26 GMT | Patrolling3_b : Sync. stage over -- normal completion; 121
    seconds execution
3 Sep 2014 16:57:26 GMT | Patrolling3_b : Results: {b=done, c=done, a=done}
3 Sep 2014 16:57:26 GMT | Patrolling3_b : > Execution done; next step is 'stop'
3 Sep 2014 16:57:26 GMT | Patrolling3_b : > Stopped with result: done ...
3 Sep 2014 16:57:26 GMT | Patrolling3_b : > End controller ...
3 Sep 2014 16:57:26 GMT | ExternalInterface : Stopped execution of running program.
3 Sep 2014 16:57:28 GMT | ExternalInterface : [Vehicle] lauv-seacon-2 - READY [READY]
3 Sep 2014 16:57:37 GMT | ShutdownThread : Program termination requested.
3 Sep 2014 16:57:37 GMT | ExternalInterface : ExternalInterface is dead.
3 Sep 2014 16:57:37 GMT | VehicleLink : VehicleLink is dead.
3 Sep 2014 16:57:37 GMT | Announcer : Announcer is dead.
3 Sep 2014 16:57:37 GMT | StatusDiffusion : StatusDiffusion is dead.
3 Sep 2014 16:57:37 GMT | Heart : Heart is dead.
```

B.2.4 Veículo 'lauv-xplore-1'

```
3 Sep 2014 16:48:12 GMT | ExternalInterface : Activating transmission to node '
    interpreter-nvl'
3 Sep 2014 16:48:14 GMT | ExternalInterface : [Vehicle] lauv-xplore-1 - READY [READY]
3 Sep 2014 16:48:15 GMT | ExternalInterface : Node defined as 'a' ...
3 Sep 2014 16:48:15 GMT | ExternalInterface : Requesting 'Patrolling3_a' to start ...
3 Sep 2014 16:48:15 GMT | Heart : Launching organ 'Patrolling3_a' ...
3 Sep 2014 16:48:15 GMT | Heart : Creating organ 'Patrolling3_a' ...
3 Sep 2014 16:48:15 GMT | Heart : Organ 'Patrolling3_a' has been started.
3 Sep 2014 16:48:15 GMT | Patrolling3_a : Node: a | Vehicle: lauv-xplore-1 | Address: 30
3 Sep 2014 16:48:15 GMT | Patrolling3_a : Patrolling3_a is born.
3 Sep 2014 16:48:15 GMT | Patrolling3_a : > Executing step: 'start'
3 Sep 2014 16:48:15 GMT | Patrolling3_a : control deadline 180 - sync deadline 60
3 Sep 2014 16:48:52 GMT | Patrolling3_a : R: done
3 Sep 2014 16:49:10 GMT | Patrolling3_a : FINAL STAGE
3 Sep 2014 16:49:15 GMT | Patrolling3_a : Sync. stage over -- normal completion; 59
    seconds execution
3 Sep 2014 16:49:15 GMT | Patrolling3_a : Results: {b=done, c=done, a=done}
3 Sep 2014 16:49:15 GMT | Patrolling3_a : > Execution done; next step is 'step2'
3 Sep 2014 16:49:15 GMT | Patrolling3_a : > Executing step: 'step2'
3 Sep 2014 16:49:15 GMT | Patrolling3_a : control deadline 600 - sync deadline 60
3 Sep 2014 16:55:12 GMT | Patrolling3_a : R: done
3 Sep 2014 16:55:20 GMT | Patrolling3_a : FINAL STAGE
3 Sep 2014 16:55:25 GMT | Patrolling3_a : Sync. stage over -- normal completion; 370
    seconds execution
3 Sep 2014 16:55:25 GMT | Patrolling3_a : Results: {b=done, c=done, a=done}
3 Sep 2014 16:55:25 GMT | Patrolling3_a : > Execution done; next step is 'rendezvous'
3 Sep 2014 16:55:25 GMT | Patrolling3_a : > Executing step: 'rendezvous'
3 Sep 2014 16:55:25 GMT | Patrolling3_a : control deadline 120 - sync deadline 60
3 Sep 2014 16:56:39 GMT | Patrolling3_a : R: done
3 Sep 2014 16:57:21 GMT | Patrolling3_a : FINAL STAGE
3 Sep 2014 16:57:26 GMT | Patrolling3_a : Sync. stage over -- normal completion; 120
    seconds execution
3 Sep 2014 16:57:26 GMT | Patrolling3_a : Results: {b=done, c=done, a=done}
3 Sep 2014 16:57:26 GMT | Patrolling3_a : > Execution done; next step is 'stop'
3 Sep 2014 16:57:26 GMT | Patrolling3_a : > Stopped with result: done ...
3 Sep 2014 16:57:26 GMT | Patrolling3_a : > End controller ...
3 Sep 2014 16:57:26 GMT | ExternalInterface : Stopped execution of running program.
3 Sep 2014 16:57:27 GMT | ExternalInterface : [Vehicle] lauv-xplore-1 - READY [READY]
3 Sep 2014 16:57:34 GMT | ShutdownThread : Program termination requested.
3 Sep 2014 16:57:34 GMT | ExternalInterface : ExternalInterface is dead.
3 Sep 2014 16:57:34 GMT | Announcer : Announcer is dead.
3 Sep 2014 16:57:34 GMT | VehicleLink : VehicleLink is dead.
3 Sep 2014 16:57:34 GMT | StatusDiffusion : StatusDiffusion is dead.
3 Sep 2014 16:57:34 GMT | Heart : Heart is dead.
```

Bibliografia

[1] G. Agha. Actors: a model of concurrent computation in distributed systems. In *MIT Press*, USA, 1986.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2014.

[4] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.

[5] Matthew Dunbabin, Peter Corke, Iuliu Vasilescu, and Daniela Rus. Data Muling over Underwater Wireless Sensor Networks using an Autonomous Underwater Vehicle. In *Robotics and Automation. ICRA*, 2006.

[6] Eclipse, `http://eclipse.org`.

[7] M. Faria, J. Pinto, F. Py, J. Fortuna, H. Dias, R. Martins, F. Leira, T.A. Johansen, J. Sousa, and K. Rajan. Coordinating UAVs and AUVs for Oceanographic Field Experiments: Challenges and Lessons Learned. In *Proc. ICRA*, 2014.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[9] A. Girard, J. Sousa, and K. Hedrick. A selection of recent advances in networked multi-vehicle systems. *Journal of Systems and Control Engineering*, 2004.

[10] J. González, I. Masmitjà, S. Gomáriz, E. Molino, J. Del Río, A. Mànuel, J. Busquets, A. Guerrero, F. López, M. Carreras, D. Ribas, A. Carrera, C. Candela, P. Ridao, J. Sousa, P. Calado, J. Pinto, A. Sousa, R. Martins, D. Borrajo, A. Olaya, B. Garau, I. González, S. Torres, K. Rajan, M. McCann, and J. Gilabert. AUV based multi-vehicle collaboration: Salinity studies in Mar Menor Coastal lagoon. In *Proc. NGCUV*. IFAC, 2012.

[11] Joshua Love, Jerry Jariyasunant, Eloi Pereira, Marco Zennaro, Karl Hedrick, Christoph Kirsch, and Raja Sengupta. CSL: A Language to Specify and Re-Specify Mobile Sensor Network Behaviors. In *15th IEEE Real-Time and Embedded Technology and Applications Symposium. LNCS*, 2009.

[12] Eduardo R. B. Marques, José Pinto, Sean Kragelund, Paulo S. Dias, Luís Madureira, Alexandre Sousa, Márcio Correia, Hugo Ferreira, Rui Gonçalves, Ricardo Martins, Anthony J. Healey Douglas P. Horner, Gil M. Gonçalves, and João B. Sousa. AUV control and communication using underwater acoustic networks. In *Proc. IEEE Oceans Europe*. IEEE, 2007.

[13] R. Martins, J.B. Sousa, and C.C. Afonso. Shallow-water surveys with a fleet of heterogeneous autonomous vehicles. *Sea Technology*, 52(11):27–31, 2011.

[14] Ricardo Martins, Paulo Sousa Dias, Eduardo R. B. Marques, José Pinto, João B. Sousa, and Fernando L. Pereira. IMC: A Communication Protocol for Networked Vehicles and Sensors. In *Proc. IEEE Oceans Europe (OCEANS'09)*. IEEE, 2009.

[15] Ricardo Martins, Paulo Sousa Dias, José Pinto, P. B. Sujit, and João Borges Sousa. Multiple Underwater Vehicle Coordination for Ocean Exploration. In *Proceedings of IJCAI*, 2009.

[16] R. Milner. Bigraphical reactive systems. In *2001 - Concurrency Theory. CONCUR*, pages 16–35, 2001.

[17] Eloi Pereira, Christoph M. Kirsch, Raja Sengupta, and João Borges de Sousa. BigActors - A Model for Structure-aware Computation. In *ACM/IEEE 4th International Conference on Cyber-Physical Systems*, 2013.

[18] José Pinto, Pedro Calado, José Braga, Paulo Dias, Ricardo Martins, Eduardo Marques, and J.B. Sousa. Implementation of a Control Architecture for Networked Vehicle Systems*. In *Proceedings of the IFAC Workshop on Navigation, Guidance and Control of Underwater Vehicles*. IFAC, 2012.

[19] José Pinto, Paulo S. Dias, Ricardo Martins, João Fortuna, Eduardo Marques, and João Sousa. The LSTS Toolchain for Networked Vehicle Systems*. In *Proceedings of MTS/IEEE Oceans Europe (Oceans'13)*. IEEE, 2013.

[20] José Pinto, Margarida Faria, João Fortuna, Ricardo Martins, João Sousa, Nuno Queiroz, Frederic Py, and Kanna Rajan. Chasing Fish: Tracking and control in a autonomous multi-vehicle real-world experiment. In *Proceedings of Oceans'13 MTS/IEEE SAN DIEGO*, 2013.

[21] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 Language Specification. 2013.

[22] Hamed Yaghoubi Shahir, Uwe Glässer, Roozbeh Farahbod, Piper Jackson, and Hans Wehn. Generating test cases for marine safety and security scenarios: a composition framework. In *Security Informatics*, 2012.

[23] J. Sousa, T. Simsek, and P. Varaiya. Task planning and execution for UAV teams. In *Proc. CDC*. IEEE, 2004.

[24] J.B. Sousa, B. Maciel, and F.L. Pereira. Sensor systems on networked vehicles. *Networks and Heterogeneous Media*, 2009.

[25] João Sousa, Tunc Simsek, and Pravin Varaiya. Task planning and execution for uav teams. In *Proceedings of 43rd IEEE Conference on Decision and Control*. IEEE, 2004.

[26] P. B. Sujit, João Borges Sousa, and Pereira F.L. UAV and AUVs coordination for ocean exploration. In *Proceedings of Oceans'09*, 2009.

[27] Xtext, `http://eclipse.org/Xtext`.