INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Pedro Elardenberg Sousa e Souza

ESTRUTURAÇÃO DE UMA ARQUITETURA DE SIMULAÇÃO DISTRIBUÍDA PARA SISTEMAS AEROESPACIAIS

Trabalho de Graduação
2021

Curso de Engenharia Aeroespacial

CDU 629.78

Pedro Elardenberg Sousa e Souza

ESTRUTURAÇÃO DE UMA ARQUITETURA DE SIMULAÇÃO DISTRIBUÍDA PARA SISTEMAS AEROESPACIAIS

Orientador

Prof. Dr. Christopher Shneider Cerqueira (ITA)

Coorientador

Maj. Eng. Daniel Lelis Baggio (CCA-SJ)

ENGENHARIA AEROESPACIAL

São José dos Campos
Instituto Tecnológico de Aeronáutica

2021

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

Sousa e Souza, Pedro Elardenberg
Estruturação de uma Arquitetura de Simulação Distribuída para Sistemas Aeroespaciais / Pedro Elardenberg Sousa e Souza.
São José dos Campos, 2021.
57f.

Trabalho de Graduação – Curso de Engenharia Aeroespacial– Instituto Tecnológico de Aeronáutica, 2021. Orientador: Prof. Dr. Christopher Shneider Cerqueira. Coorientador: Maj. Eng. Daniel Lelis Baggio.

1. Simulador. 2. Simulações Aeroespaciais. 3. Simulação Distribuída. 4. HLA. 5. RTI. I. Instituto Tecnológico de Aeronáutica. II. Título.

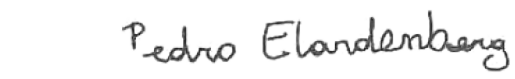
REFERÊNCIA BIBLIOGRÁFICA

SOUSA E SOUZA, Pedro Elardenberg. **Estruturação de uma Arquitetura de Simulação Distribuída para Sistemas Aeroespaciais**. 2021. 57f. Trabalho de Conclusão de Curso (Graduação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: Pedro Elardenberg Sousa e Souza
TITULO DO TRABALHO: Estruturação de uma Arquitetura de Simulação Distribuída para Sistemas Aeroespaciais.
TIPO DO TRABALHO/ANO: Trabalho de Conclusão de Curso (Graduação) / 2021

É concedida ao Instituto Tecnológico de Aeronáutica permissão para reproduzir cópias deste trabalho de graduação e para emprestar ou vender cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte deste trabalho de graduação pode ser reproduzida sem a autorização do autor.



Pedro Elardenberg Sousa e Souza
Rua H8B, 221
12.228-461 – São José dos Campos–SP

ESTRUTURAÇÃO DE UMA ARQUITETURA DE SIMULAÇÃO DISTRIBUÍDA PARA SISTEMAS AEROESPACIAIS

Essa publicação foi aceita como Relatório Final de Trabalho de Graduação

Pedro Elardenberg

Pedro Elardenberg Sousa e Souza

Autor

Christopher Shneider Cerqueira (ITA)

Orientador

Daniel Lélis Baggio

Daniel Lelis Baggio (CCA-SJ)

Coorientador

Cristiane Aparecida Martins

Assinado digitalmente por Cristiane Aparecida Martins
DN: C=BR, OU=Instituto Tecnológico de Aeronáutica, O=Cristiane Martins, CN=Cristiane Aparecida Martins, E=cmartins@ita.br
Razão: Eu sou o autor deste documento
Localização:
Data: 2021-11-17 08:54:08

Prof. Dr. Cristiane Aparecida Martins
Coordenadora do Curso de Engenharia Aeroespacial

São José dos Campos, 10 de novembro de 2021.

Aos amigos da Graduação do ITA, por me proporcionarem grandes momentos de aprendizado acadêmico, fazerem-me enxergar novos horizontes de oportunidades e partilharem de suas companhias por todos esses anos.

Agradecimentos

Agradeço ao Prof. Christopher, por ser meu professor orientador e conselheiro durante a graduação, por me abrir as portas do Centro Espacial ITA e, desde 2018 me proporciona trabalhos nos quais eu mais me identifico.

Agradeço ao Centro de Computação da Aeronáutica de São José dos Campos (CCA-SJ), em particular nas figuras do Major Ednelson Silva de Oliveira e do Major Daniel Lélis Baggio, os quais ofereceram cordialmente o espaço que dispunha das ferramentas tecnológicas necessárias para o andamento do trabalho, bem como muito do conhecimento que aqui foi aplicado.

Agradeço aos meus colegas, sem os quais eu não teria chegado até aqui, seja pelos bons momentos compartilhados, seja pelas dúvidas tiradas e pela solicitude em ajudar independentemente da ocasião.

Agradeço a minha mãe, que superou todas as dificuldades para que eu sempre pudesse ter uma educação de qualidade, e por ela mesma ter me educado da melhor forma possível.

Acima de tudo, agradeço a Deus, por ter-me dado forças, conhecimento e por colocar tantas pessoas na minha vida que me inspiraram e não me fizeram desistir.

"O tempo é o único capital daqueles que têm apenas sua inteligência por fortuna"
— Honoré de Balzac

Resumo

Missões espaciais, devido a sua alta precisão requerida em parâmetros orbitais e de lançamento, bem como seu alto custo operacional, requerem simulações para que sejam feitas verificações, validações e que sejam previstas algumas rotinas de operações. Tais missões, no entanto, possuem agentes que se comunicam com agentes de outras missões, de modo que uma única simulação não abarca toda a complexidade e escalabilidade de um ambiente complexo. Neste trabalho, foi proposta a utilização de um *software* de simulação de cenários no qual foi utiizada uma infraestrutura baseada em *High Level Architecture* (HLA) de modo a integrar sistemas de simulação de aplicações aeroespaciais.

Para isso, foi desenvolvido um *software* de simulação que se comunica com o primeiro, enviando e recebendo sinais de veículos espaciais representados no cenário, de modo a demonstrar a viabilidade desta arquitetura de simulação distribuída.

Abstract

Space missions are, due to its high accuracy in orbits and launching needed, as well as its high operational cost, require simulations in order to verify, validate and to preview some operation routines. Such missions, nevertheless, have agents that communicate with other missions' agents so that a single simulation does not embrace all of the complexity and scalability of a complex ambient. In this course assignment, it was proposed the utilization of a scenery simulation software in which it was used an infrastructure based on High Level Architecture (HLA) in order to integrate simulation systems of aerospace applications.

Therefore, it was developed a simulation software that communicates to the first one, sending and receiving signals from spacecrafts represented in the scenario, in order to demonstrate the viability of this distributed simulation architecture.

Lista de Figuras

Lista de Abreviaturas e Siglas

DS	Distributed simulation
DIS	Distributed Interactive Simulation
ALSP	Aggregate Level Simulation Protocol
HLA	High Level Architecture
RTI	Runtime Infrastructure
IDE	Integrated Development Environment
FOM	Federate Object Models
OMT	Object-Modeling Technique
RPR	Real-Time Platform Reference
GUI	Graphical User Interface
TLE	Two-Line Element
CRC	Central RTI Component
ITA	Instituto Tecnológico de Aeronáutica
CCA-SJ	Centro de Computação de Aeronáutica de São José dos Campos

Sumário

1 Introdução

1.1 Motivação

Os sistemas complexos requerem ferramentas de exploração e gerenciamento de complexidade para possibilitar aos engenheiros:

- A visão do domínio do problema, identificando as necessidades dos stakeholders e seus requisitos;
- A visão do domínio da solução, objetivando identificar alternativas de arquiteturas que possam vir a se tornar soluções candidatas à concretização do projeto.

Com o auxílio de um simulador, torna-se possível realizar a variação de arquiteturas e sua futura operação.

No setor aeroespacial, é necessário que vários sistemas estejam operando em conjunto para que o objetivo fim possa ser realizado. Para uma missão espacial, por exemplo, tem-se o veículo espacial (satélite), veículo lançador (foguete) e a estação solo como principais agentes. Esses agentes, por sua vez, possuem seus próprios subsistemas. Esse sistema complexo, como o próprio nome sugere, apresenta uma grande dificuldade em ser testado em simulação, uma vez que cada um desses sistemas tem suas devidas particularidades. Por isso, surge a ideia de utilizar Simulação Distribuída para simular esses casos.

Uma demanda observada dentro desse contexto é a necessidade de testar, em um ambiente unificado, cenários de simulação de determinados institutos dentro do Campus do DCTA que possuem seus próprios simuladores, como na figura 1.1, a qual apresenta um cenário hipotético no qual simulações de sistemas do Centro Espacial ITA (CEI), Instituto Nacional de Pesquisas Espaciais (INPE) e Centro de Operações Espaciais (COPE) são integradas numa única simulação. Por sua vez, cada sistema pode ser modelado de modo independente, isto é, sem as condições dos demais cenários.

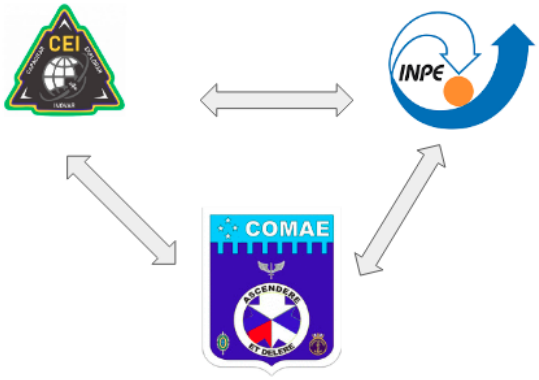


FIGURA 1.1 – Cenário hipotético de intercomunicação de simuladores

1.2 Objetivo

O objetivo deste projeto é explorar a utilização de Simulação Distribuída para integrar aplicações de modo a simular a robustez de uma arquitetura conceitual de um sistema complexo dentro do âmbito aeroespacial.

1.3 Organização do trabalho

1.3.1 Introdução

O capítulo 1 contém a introdução do trabalho, onde são expostos o objetivo, sua motivação, a descrição sistema e a formulação do problema com a nomenclatura utilizada; além de uma revisão bibliográfica da literatura relacionada ao tema do trabalho.

1.3.2 Revisão Bibliográfica

No capítulo 2, são apresentados os principais conceitos utilizados para a formulação do problema: o que é simulação distribuída, como ela surgiu, o que é *High Level Architecture* e como ela pode ser aplicada na engenharia e, por fim, como isso pode ser usado dentro do contexto Aeroespacial.

1.3.3 Tecnologias pesquisadas e utilizadas

No capítulo 3, são apresentados os *softwares* nos quais este trabalho operou. São mostradas as formas de integração de simuladores, utilizando *Runtime Infrastructure* (RTI) em um simulador de cenários chamado VR-Forces.

1.3.4 Resultados e Discussão

No capítulo 4, são descritas as atividades realizadas ao longo do trabalho de graduação, que incluem a aplicação dos conceitos vistos no capítulo 2, a configuração dos *softwares* vistos no capítulo 3, a intercomunicação entre eles e a implementação do cenário modelo no qual o trabalho final é apresentado.

1.3.5 Conclusão

No capítulo 5, são apresentadas as conclusões do trabalho, além de sugestões de desenvolvimento futuro.

2 Simulação Distribuída e HLA

2.1 Simulação Distribuída

De um modo simplificado, simulação distribuída, ou *distributed simulation* (DS) é uma tecnologia que permite que um programa de simulação seja executado em computadores paralelos ou distribuídos (FUJIMOTO, 2000). Uma simulação computacional é um programa que modela o comportamento de algum sistema existente ou inexistente ao longo do tempo.

O que distingue simulação paralela de simulação distribuída é a ênfase na velocidade de execução (TOPÇU; OĞUZTÜZÜN, 2017). A principal preocupação de uma simulação paralela é aumentar a velocidade de processamento por meio de tecnologias de alta performance. Simulação distribuída, por sua vez, foca em tratar sub-partes da simulação de modo separado. Por exemplo, dois simuladores de voo, operando num exercício de voo coordenado constituem uma simulação distribuída.

Simulação paralela ou distribuída referem-se a tecnologias que permitem a uma simulação executar em um sistema computacional contendo múltiplos processadores, tais como computadores pessoais, interconectados por uma rede de comunicação.

São benefícios do uso de arquiteturas de simulação (FUJIMOTO, 2000):

- Redução do tempo de resposta, na qual uma simulação complexa pode ter suas tarefas subdivididas em subtarefas sendo executadas em diferentes processadores independentes;
- Distribuição geográfica, permitindo que se crie um mundo virtual com vários participantes geograficamente distantes uns dos outros, mas interagindo entre si;
- Integração de simuladores que são executados em diferentes plataformas operacionais;
- Tolerância a falhas, uma vez que a distribuição da simulação entre vários processadores permite que caso um processador sofra uma avaria, os demais possam continuar

o processamento assumindo o trabalho do processador com problemas, fazendo com que a simulação não seja interrompida.

Quanto ao uso de simuladores distribuídos na área espacial, para além das vantagens apontadas - escalabilidade, paralelismo, etc., pode-se citar vantagens na incorporação de hardware na malha da simulação (AZEVEDO, 2014). Desta forma, vários elementos físicos de um laboratório podem estar geograficamente distantes e serem incorporados na simulação como elementos distribuídos, o que se mostra bastante interessante, uma vez que os equipamentos de teste são bastante específicos, caros, e tendem a estar instalados em laboratórios ou ambientes de integração e teste isolados.

2.2 Distributed Interactive Simulation (DIS)

Nas décadas de 70 e 80, o departamento de defesa americano criou um protocolo de simulação em rede (Simulation Networking), o qual foi usado para desenvolver uma nova geração de simuladores conectados (TOPÇU; OĞUZTÜZÜN, 2017). Esse projeto foi utilizado para desenvolver o que foi chamado de *distributed interactive simulation*.

DIS é um protocolo de envio de mensagem que define o modo que os simuladores vão se comunicar entre si, por meio de pacotes de mensagem chamados *protocol data units* (PDUs). Esse protocolo foi aprovado em 1992 pelo Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE) e utilizado para mandar e receber dados via rede. Um conjunto de PDUs eram definidos como padrão e, se um novo tipo de pacote tivesse que ser adicionado a esse padrão.

2.3 Aggregate Level Simulation Protocol (ALSP)

No projeto americano, surgiu a necessidade de fazer simulações para simular combates de agregados para o treinamento dos comandantes (TOPÇU; OĞUZTÜZÜN, 2017). Essas simulações tinham o desafio de serem distribuídas e não terem um número predefinido de agentes. Deste modo, foi criado o *Aggregate Level Simulation Protocol* (ALSP), o qual incluía desafios como gerenciamento de tempo, gerenciamento de dados e independência de arquitetura. ALSP desenvolveu vários conceitos que tornaram-se base para a construção da *High Level Architecture* como objetos, atributos, interações, federações, modelos de federações e infraestrutura de software.

2.4 High Level Architecture (HLA)

Como uma evolução do DIS e do ALSP, surge o padrão HLA o qual, devido a seu alto grau de abstração no que tange a aspectos de distribuição e de serviços fornecidos pela infraestrutura, tem sido pesquisado e usado em diferentes áreas (AZEVEDO, 2014), e um dos objetivos deste trabalho é avaliar a viabilidade desta abordagem. Ademais, as Forças Armadas, por meio da Portaria Normativa Nº1.814/MD, de 13 de junho de 2013, determinou que os simuladores desenvolvidos ou comprados deverão utilizar HLA (BRASIL, 2013).

HLA é uma arquitetura de sistemas que facilita o reúso por meio de interoperabilidade e compatibilidade de simulações (TOPÇU *et al.*, 2016). Interoperabilidade pode ser definida como a capacidade de simulações de trocar informações de um modo útil e significativo. Compatibilidade é definido como a capacidade que permite selecionar e montar componentes de várias maneiras a fim de atingir o objetivo do sistema. Para que essas características sejam melhor preservadas, a topologia de integração é feita através de um barramento compartilhado. Dessa forma, é mais fácil inserir ou retirar uma entidade, e não é necessário se preocupar com a compatibilidade por cada par de entidades.

2.4.1 Federação e Federados

Um federado pode ser definido como uma aplicação que é implementada ou conforma-se a um padrão HLA (TOPÇU *et al.*, 2016). Federados interagem com interfaces especificadas em HLA, e portanto podem participar de uma execução de simulação distribuída. Uma aplicação de federado pode ingressar na mesma execução várias vezes ou pode entrar em múltiplas execuções, criando um novo federado sempre que isso ocorrer. O federado é a aplicação que se conecta à RTI, tipicamente um simulador.

Federação é o conjunto de federados que compartilham uma especificação de dados em comum na qual é capturado o *Federation Object Model* (FOM). A federação é o grupo de sistemas que operam entre si.

Federation Object Model é o arquivo que contém uma descrição da troca de dados na federação. Pode ser visto como a linguagem em que a federação opera (MöLLER, 2012).

Execução de federado *Federation Execution* é uma instanciação em tempo de execução de uma federação. Ela é a própria execução de simulação.

2.5 Runtime Infrastructure (RTI)

A maioria das arquiteturas requerem infraestruturas que permitam suas premissas (TOPÇU *et al.*, 2016). HLA também vem com uma infraestrutura que permite comunicação entre federados. RTI pode ser definido como a infraestrutura de software fundamental. Federados interagem com RTI por meio de serviços padrão e interfaces para participar da simulação distribuída e trocar dados. Ela suporta as regras HLA com os serviços que fornecem as interfaces especificadas. A figura 2.1 mostra como é a arquitetura de execução de uma aplicação HLA.

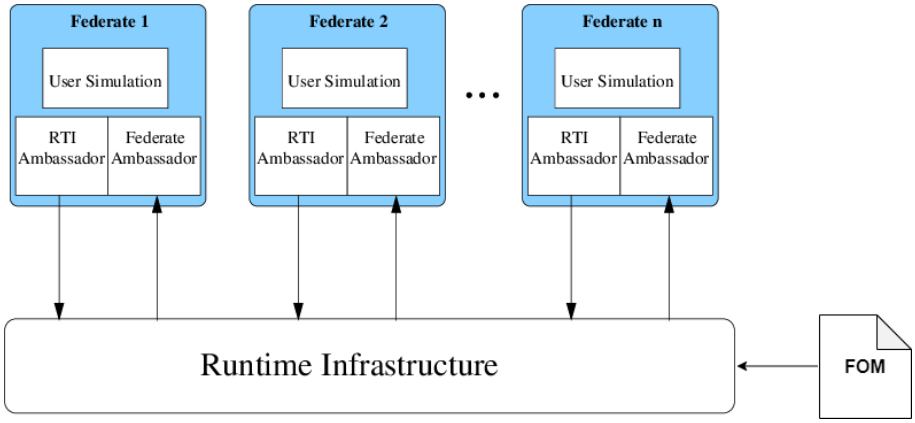


FIGURA 2.1 – Arquitetura de federados em uma aplicação HLA. Fonte: (LEES *et al.*, 2006)

2.6 Conteúdo de um FOM

Os três principais componentes de um FOM são as classes Objeto, Interação e Dados.

- Um objeto é um componente que persiste ao longo do tempo e possui atributos que podem ser atualizados. Um satélite é um exemplo da classe objeto. Nome, posição e velocidade são atributos típicos.
- Uma interação é algo que não persiste ao longo do tempo. Iniciar, parar, enviar mensagens de rádio são exemplos da classe interação. Uma interação geralmente possui parâmetros.
- Os tipos de dados descrevem a semântica e a representação técnica dos atributos de um objeto e dos parâmetros de uma interação.

Um FOM pode ser gradualmente estendido ao longo do tempo sem comprometer as simulações existentes. A necessidade de aumentar o nível de interoperabilidade é algo que, em geral, acontecerá nas aplicações desenvolvidas.

3 Tecnologias Pesquisadas e Utilizadas

Uma parte importante deste trabalho foi a escolha das teclonogias e dos programas que foram utilizadas. A execução final do trabalho foi concebida de modo a existir um programa que simulasse um ambiente espacial conectado via RTI a um programa desenvolvido para interagir com esse cenário, utilizando HLA. Embora tanto o protocolo HLA quanto o modo como o caso modelo fossem adaptados para funcionar com várias máquinas em rede, por viabilidade e praticidade decidiu-se executar todos os *softwares* num mesmo computador.

Para a simulação de ambiente + RTI, foram pesquisadas algumas opções *open source,* como Portico (PORTICO..., 2021) e CERTI (CERTI, 2021). Entretanto, dado à maior praticidade no uso, à existência de manuais e materiais de apoio e à acessibilidade do programa, dado que o CCA-SJ possui a licença do VR-Forces e cedeu para o trabalho a máquina em questão, o simulador de ambiente (VR-Forces) e a RTI da Mak Technologies (MAK..., 2021) foram escolhidos. Outra empresa especializada nesses produtos é a Pitch Technologies (PITCH..., 2021), da qual foi utilizada o visualizador de FOMs.

Para a criação do federado, foi escolhida a linguagem *java* por já haver um programa que utilizava o protocolo HLA para ligar-se a uma simulação, o qual foi modificado para antender à necessidade deste trabalho.

3.1 Mak VR-Forces

VR-Forces é uma aplicação de *computer generared forces* e uma caixa de ferramentas (TECHNOLOGIES, 2020b). VR-Forces é implementado utilizando front-end (interface gráfica com o usuário) e back-end (*engine* de simulação) separados que permitem uma grande flexibilidade para manipular certas necessidades de simulação.

VR-Forces fornece visões em 2D e 3D do mundo simulado (figuras 3.1 e 3.2) integradas à interface gráfica. A visão 2D fornece o entendimento de uma batalha simulada ao inserir objetos e informação de dados sobre tática, estratégia e visão. A visão 2D responde a questões acerca do posicionamento de unidades e como o terreno pode afetar a batalha.

A visão em 3D fornece perspectivas de terreno e modelos das entidades de forma tridimensional. Esta visão é mais indicada para posicionar entidades de modo mais preciso e para funcionar em terrenos nos quais entidades podem estar em várias altitudes possíveis, como em aplicações aeroespaciais.

VR-Forces é compatível com os padrões de simulação *Distributed Interactive Simulation* (DIS) e com *High Level Architecture* (HLA). Ele suporta vários formatos de banco de dados e múltiplos servidores de terreno.

VR-Forces é interoperável com Mak RTI.

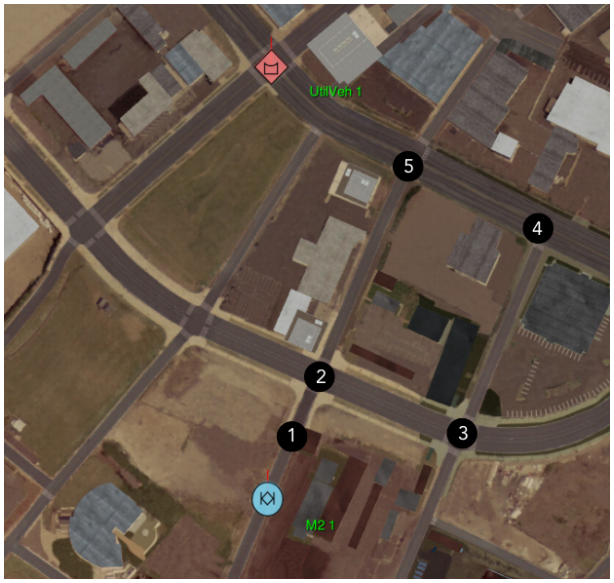


FIGURA 3.1 – Visão 2D de um cenário do VR-Forces

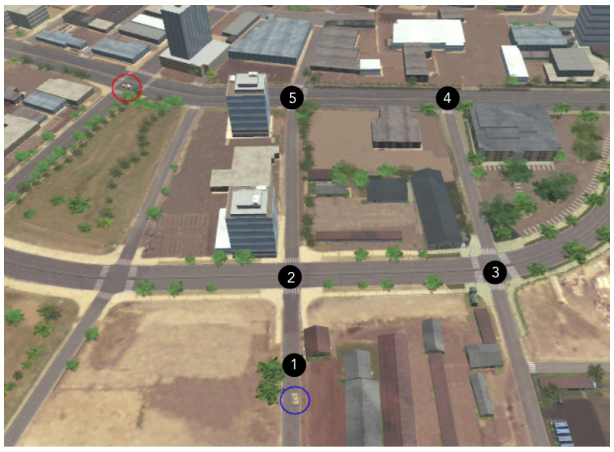


FIGURA 3.2 – Visão 3D de um cenário do VR-Forces

3.2 Mak RTI

Mak RTI (Runtime Infrastructure) é uma biblioteca de software que implementa o protocolo HLA 1.3, 1516 (SISO DLC HLA API 1516), e HLA Evolved (IEEE 1516-2010) (TECHNOLOGIES, 2018). Em HLA, aplicações trocam Federation Object Models (FOMs) por meio de requisições da RTI, ou seja, aplicações RTI sempre são utilizadas em HLA.

A figura 3.3 mostra a interface com usuário deste programa.

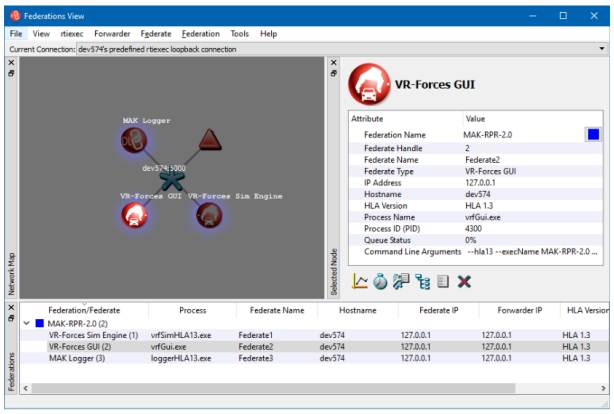


FIGURA 3.3 – Interface gráfica do Mak RTI

3.3 Pitch Visual OMT

Pitch Visual OMT é a ferramenta base para desenvolvimento e manutenção de modelos de objeto para o padrão HLA (TECHNOLOGIES, 2013). Dentre outras funcionalidades, ela permite a visualização de módulos de *Federation Object Models* (FOMs), como objetos (figura 3.4), interações e tipos de dados por meio de uma interface gráfica.

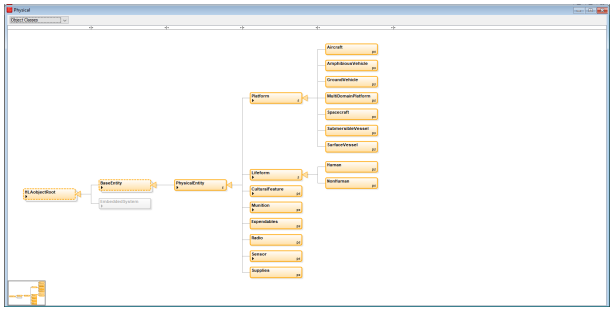


FIGURA 3.4 – Interface gráfica do Pitch Visual OMT - Classe Objeto do RPR FOM 2.0

Árvores de classes de objetos e grafos em miniatura são utilizados para mostrar as hierarquias das classes. Vários módulos FOM podem ser sobrepostos para mostrar a combinação de vários módulos. Existe, também, uma visão geral do projeto inteiro, permitindo que se explore quais módulos FOM são utilizados e quais tabelas os constituem.

3.3.1 RPR FOM 2.0

Real-Time Platform Reference Federation Object Model (RPR FOM) é o FOM mais amplamente utilizado em simulações de defesa (MöLLER *et al.*, 2014), e por padrão é compatível com o VR-Forces. O principal foco é em simulação em tempo real de cenários de guerra, nos quais estão presentes plataformas, veículos, humanos e armas, bem como interações de rádio e programação de logística.

É comum utilizar FOMs padronizados (ou FOMs de referência), e modificá-los para atender aos requisitos de um projeto ou programa particular. Eles ajudam a reduzir custos e tempo, por permitir a reutilização de trabalhos anteriores. Também reduzem riscos, pois o modelo já foi testado.

Para este trabalho, utilizar esse modelo tornou possível a interação do *software* VR-Forces com o simulador de satélite criado, baseado no programa *Federado Aeronave*.

3.4 Eclipse IDE

A IDE Eclipse é um Ambiente de Desenvolvimento Integrado de código aberto e gratuito, que reúne ferramentas para apoiar o desenvolvimento de softwares em diversas linguagens de programação, como por exemplo, Java, JavaScript/TypeScript, PHP e entre outras (IDE..., 2021).

A Eclipse IDE é utilizada para auxiliar no processo de desenvolvimento de softwares e aplicações de forma ágil. Com a IDE, é possível fazer a edição do código do software que está sendo desenvolvido, realizar a compilação ou interpretação e o debug (depuração), processo que contempla o mapeamento e testes da aplicação para encontrar possíveis bugs.

Dessa forma, foi possível editar os códigos e testá-los com maior eficiência. A figura 3.5 mostra o código do Federado Aeronave RPR 2.0 aberto no ambiente da IDE Eclipse.

3.5 Federado Aeronave

Este foi o programa utilizado como modelo para o desenvolvimento do simulador deste trabalho. Ele foi originalmente dado em um curso sobre HLA ministrado por Björn Möller, cofundador da empresa Pitch Technologies no Comando de Operações Terrestres em Agosto de 2009, do qual o CCA participou e o concedeu para este trabalho (CCA-SJ, 2009). O programa usa o RPR FOM 2.0 para inserir um objeto do tipo PhysicalEntity.Plaftorm.Aircraft na simulação e modificar sua posição a um determinado espaço de tempo. A figura 3.6 mostra o comportamento original do programa. Utilizando um si-

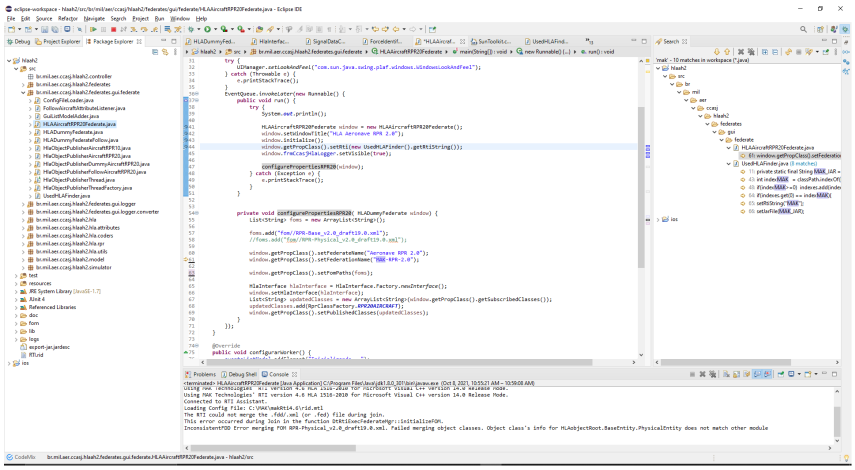


FIGURA 3.5 – Ambiente de desenvolvimento de *software* do Eclipse IDE

mulador de ambiente, denominado IOS, uma interface gráfica construída em *java* e que também utiliza o RPR FOM 2.0 como protocolo HLA, a aeronave (ponto vermelho) é iniciada no centro do mapa e desloca-se segundo uma cinemática descrita no código 3.1.

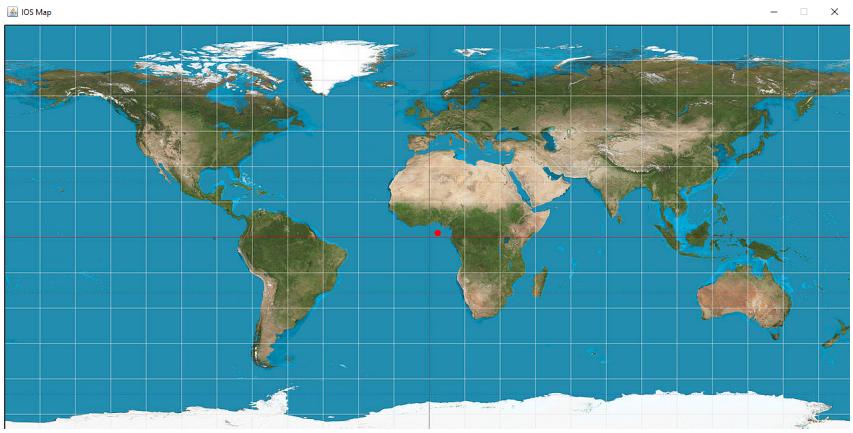


FIGURA 3.6 – Simulador de ambiente com um simulador de aeronave inserido, conectados por RTI

```
1    // um Boeing 777 faz 1 grau/400 s, no equador, aproximadamente
2    // vamos fazer 1 grau/10 s, aqui, timescale de 40
3    lon+=0.01;
4    lat+=0.005;
5    if(lon>180) lon=-180;
6    if(lat>90)  lat=-90;
7    double longitudeRadiano = ConversorGeo.grauDecimalParaRadiano(
        lon);
8    double latitudeRadiano  = ConversorGeo.grauDecimalParaRadiano(
        lat);
9    CoordenadaGeocentrica geocentrica = ConversorGeo.fromGeodetic(
        latitudeRadiano, longitudeRadiano, 400000);
```

Trecho de código 3.1 – Trecho que descreve o movimento da aeronave no mapa

4 Resultados e Discussão

Este estudo de caso consiste de simulação espacial na qual foram inseridos vários satélites como entidades do VR-Forces (VR-FORCES..., 2019). Dentre essas entidades, uma foi configurada para, ao detectar uma unidade inimiga (neste caso, uma aeronave), enviar uma mensagem de texto para outro satélite. Essa mensagem de texto é trafegada por meio de um protocolo HLA e, portanto, pode ser entendida por um federado inserido na simulação. Este federado, por sua vez, lê a mensagem e mostra-a no ambiente determinado que, neste caso, é o *console* da IDE *Eclipse* (figura 4.21). O programa, então, envia um pacote de dados de volta para uma entidade do VR-Forces, contendo outra mensagem de texto, demonstrando, assim, a bidirecionalidade da interação por HLA via RTI. Semelhantemente, o federado insere um objeto na simulação, mostrando como ocorre a comunicação tanto pelas classes *Object* quanto pelas classes *Interaction*, conforme as normas do *Federation Object Model* (FOM).

4.1 VR-Forces: Preparando o cenário

4.1.1 Iniciando o VR-Forces

Nesta seção, é demonstrado como foi feita a criação e preparação do cenário espacial dentro do *software* VR-Forces, no qual o trabalho foi realizado. A explicação de como utilizar o *software* é feita conforme o manual VR-Forces First Experience da MÄK (TECHNOLOGIES, 2020a).

1. No menu inicial, selecione **MAK Technologies → VR-Forces GUI + Simulation Engine (64 bit)**. A janela das configurações de simulação é aberta (figura 4.1).

2. Na janela das configurações de simulação, selecione a configuração HLA 1516 Evolved RPR 2.0 with MAK extensions.

3. Clique em *Launch*. A Janela de configuração de conexão com o MAK RTI é aberta (figura 4.2). Escolha **rtiexec connections → VRFORCES's predefined rtiexec connection**.

4. Clique em *Connect*. O rtiexec é iniciado.

5. Uma segunda janela de configuração de conexão com o MAK RTI é aberta. Isso acontece porque são inseridos dois federados ao RTI: o VR-Forces GUI e o Simulation Engine. Selecione a mesma conexão do rtiexec e clique novamente em *Launch*.

6. No VR-Forces GUI, vá em **File → New Scenario**. Uma janela com diversos cenários predefinidos é aberta.

7. Selecione o cenário WorldGeocentric.mtf. Um ambiente do planeta Terra é carregado (figura 4.3).

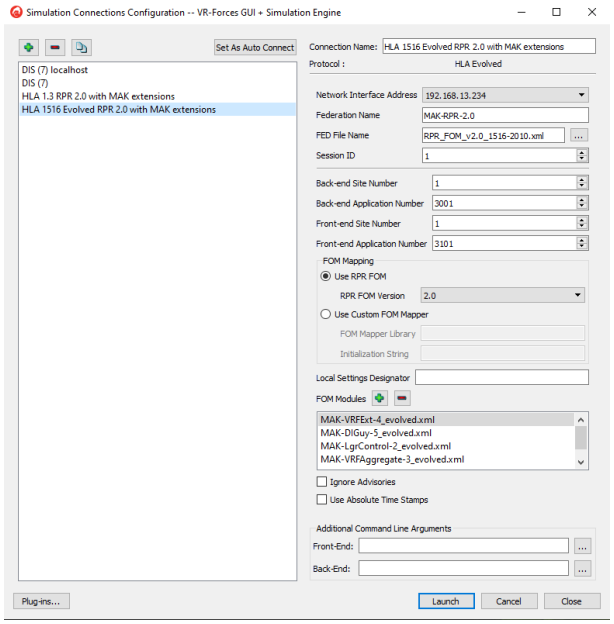


FIGURA 4.1 – Configurações de conexão do VR-Forces. O programa oferece suporte para os protocolos DIS e HLA

4.1.2 Criando entidades

Nesta seção, é mostrado como se inserem os satélites (com órbita definida por um padrão conhecido) e outras entidades manualmente. Além disso, alguns modos de exibição de certas propriedades de uma entidade foram ligados, para elucidar determinadas funcionalidades.

1. No VR-Forces GUI, vá em **File → Import Scenario Objects**. Uma janela com os tipos de objetos é aberta.

2. Na pasta *tle*, selecione para exibir arquivos do tipo TLE Scenario File (*.tle) (*). O VR-Forces tem, por padrão, um arquivo de texto com este formato, embora arquivos

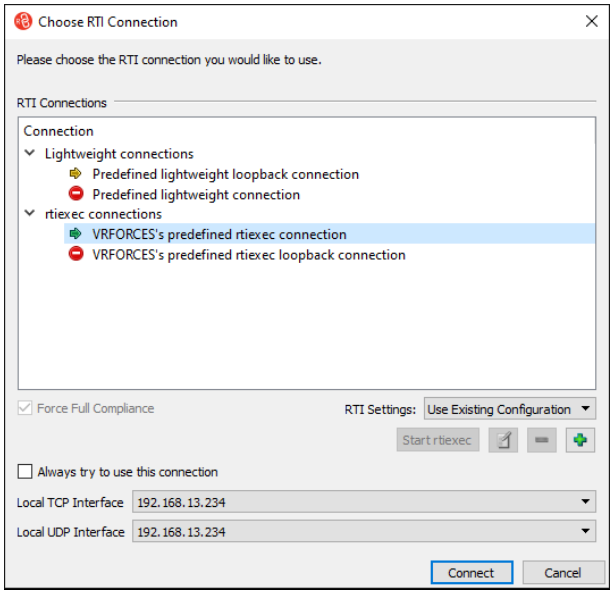


FIGURA 4.2 – Janela de conexão com o MAK RTI. O VR-Forces utiliza este programa como padrão RTI

com este formato possam ser encontrados em alguns sites, como (SPACE-TRACK. . . , 2021).

3. Selecione o arquivo desejado e clique em Open. Uma janela será aberta, na qual se pode selecionar o tipo de unidade a ser importada para o cenário (figura 4.4). Selecione a unidade desejada (**). Os satélites são carregados na simulação

4. No lado direito do VR-Forces GUI, clique na aba Simulation Objects. A janela de objetos de simulação é aberta (figura 4.5).

5. No campo *Force*, selecione o botão vermelho (força inimiga). Utilizar a força inimiga faz o satélite identificar a unidade quando ela está em seu campo de visão.

6. Selecione a unidade desejada (***). Clique na posição do mapa na qual se deseja inserir a unidade. Para inserir mais de uma unidade, clique novamente com o botão esquerdo do mouse.

7. Para sair do modo de criação de unidades, clique com o botão direito do mouse.

8. Para adicionar modos de exibição do satélite, vá em **Settings** → **Display**. A janela de configuração de modos de exibição é aberta.

9. Clique na aba *Observer Settings* (ícone do binóculo). A janela de configurações de observação é aberta (figura 4.6).

10. No campo *Model Set*, selecione *3D Models*. Os objetos de simulação mudam para o modo 3D.

FIGURA 4.3 – Cenário WorldGeocentric.mtf do VR-Forces

11. Na lista *Visible Visual Types*, selecione o campo *Range Rings*. A esfera de alcance de detecção das unidades (neste caso, o satélite), é mostrada quando se clica na unidade.

12. Na lista *Visible Visual Types*, selecione o campo *Sensor Contact Lines*. Os objetos identificados pela unidade são marcados por uma linha que os liga, quando se clica sobre ela.

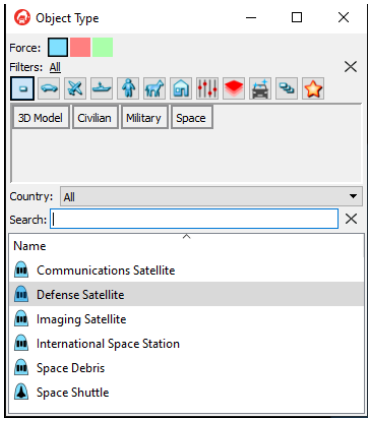


FIGURA 4.4 – Tipos de unidades compatíveis com o formato TLE

***** Um conjunto do tipo ***two-line element*** **(TLE)** é um formato de dados que contém uma lista de elementos orbitais de um objeto que orbita ao redor da Terra para um dado tempo (TWO-LINE..., 2021a). A figura 4.8 mostra um exemplo de uma órbita de satélite descrita no formato TLE. Uma explicação detalhada sobre cada campo de dados do formato pode ser vista em (TWO-LINE..., 2021b).

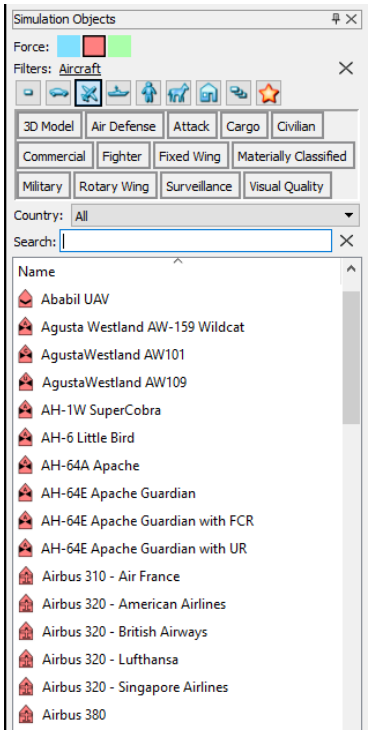


FIGURA 4.5 – Lista de objetos de simulação

** Para este caso, a principal diferença entre as unidades é o modo como elas detectam uma entidade inimiga. Foi escolhido um objeto do tipo Defense Satellite, pois seu modo de detecção é por infra-vermelho, o que se mostrou mais rápido e prático para o exemplo criado.

*** Para este caso, foram escolhidas aeronaves do tipo Boeing 777, mas poderia ser qualquer unidade que pudesse ser detectada pelo satélite.

4.1.3 Escrevendo um plano

Para controlar os objetos dentro da simulação, é possível designar-lhes tarefas (ou *tasks*). *Tasks* são comandos simples que envolvem atividades como mover-se ou enviar uma mensagem. Entretanto, se deseja-se controlar múltiplas situações ou determinada situação em circunstâncias específicas, deve-se designar-lhe um plano.

Nesta seção, é mostrado como foi configurado o plano de um dos satélites para enviar uma mensagem de rádio para um outro objeto após ele ter detectado uma entidade inimiga.

1. Clique com o botão direito no satélite. Uma janela de opções é exibida.
2. Clique em **Plan...**. A janela de plano para a unidade é exibida.
3. no espaço da lista de passos do plano, clique com o botão direito. Em seguida, escolha a opção **Conditions → Wait Until**. A janela de condições é aberta.

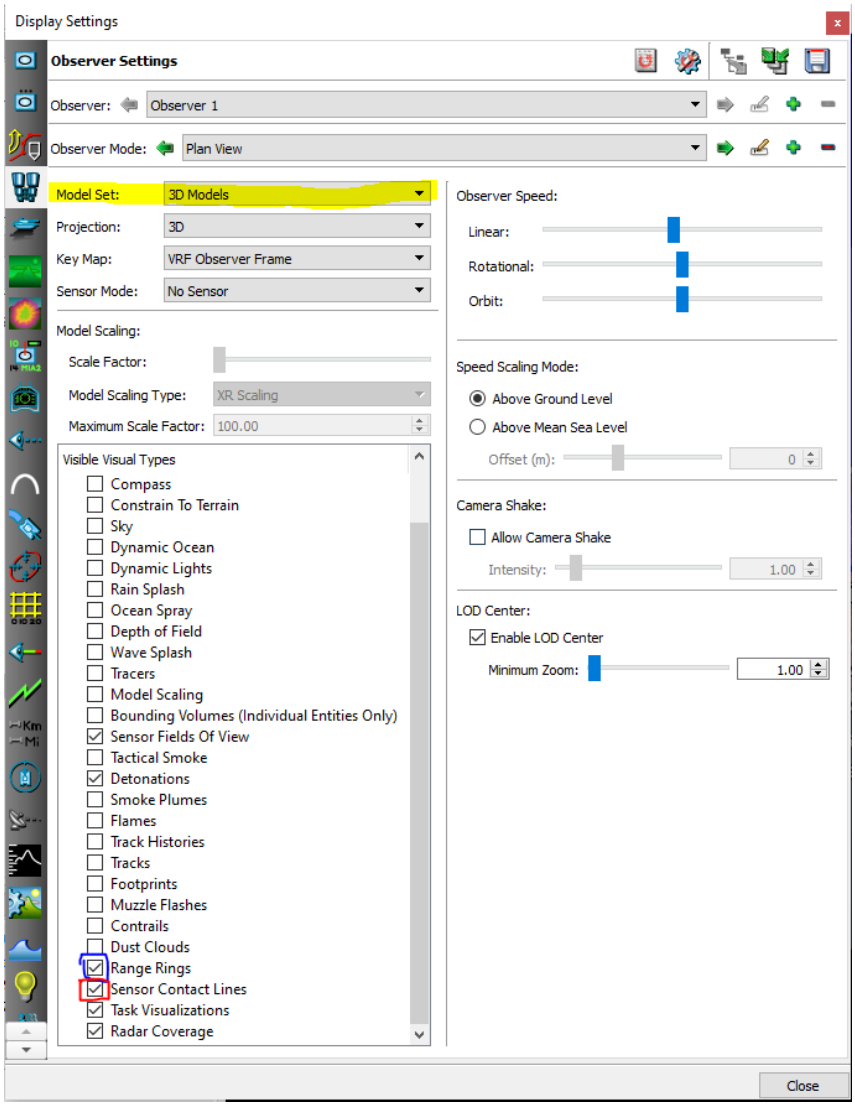


FIGURA 4.6 – Janela de configuração de modos de exibição: configuração do observador

4. No campo **<choose a conditional>** , selecione a opção Detect Entity (figura 4.9). Uma janela com a lista de entidades do cenário aparece.

5. Escolha uma unidade inimiga. Clique em Ok.

6. De volta à janela de planos do satélite, vá em **Task → Radio → Send Text Message**. Uma lista de entidades do cenário aparece.

7. Escolha uma unidade aliada. Se um federado estiver conectado à simulação, ele também poderá ser escolhido.

8. Digite a mensagem desejada. Clique em Ok. O plano deste satélite deverá ser similar ao da figura 4.10.

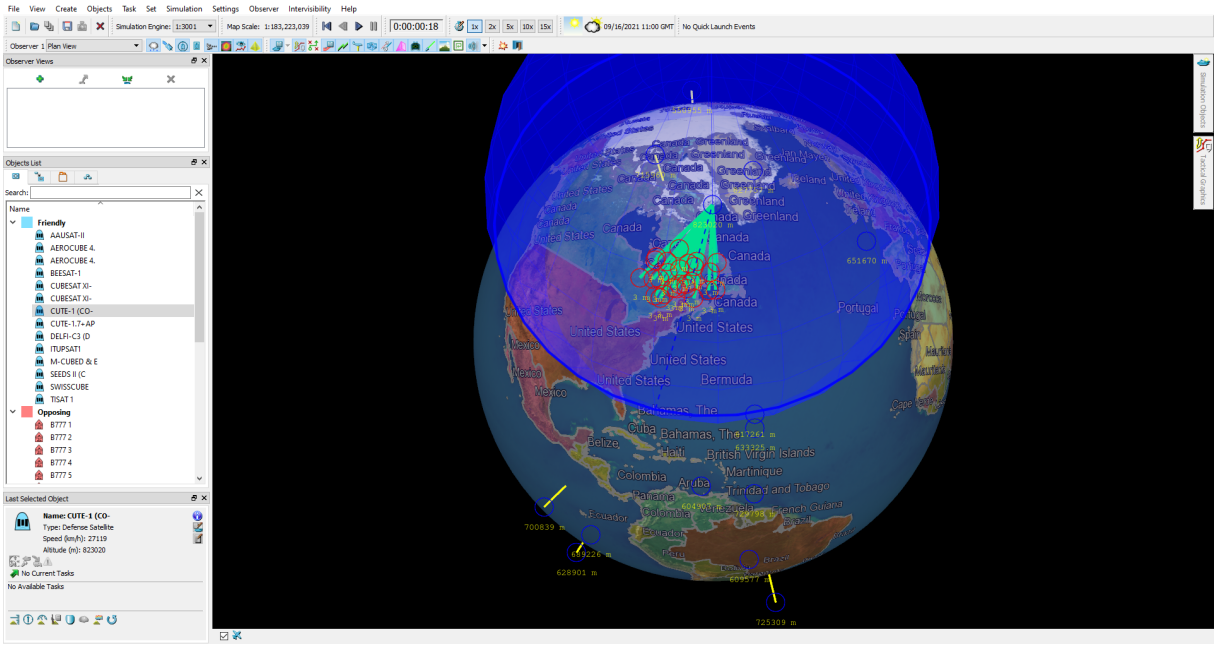


FIGURA 4.7 – Cenário do trabalho configurado, conforme mostrado na subseção 4.1.2

```
CUBESAT XI-V
1 28895U 05043F   21259.60439225  .00000203  00000-0  47831-4 0  9996
2 28895  98.0712  30.9147 0015411 284.3306  75.6186 14.63958440847346
```

FIGURA 4.8 – Exemplo de formato TLE: Cubesat XI-V (fonte: (SPACE-TRACK..., 2021))

4.1.4 Verificando os resultados da simulação

Nesta seção, é mostrada uma forma de verificar os resultados da simulação. Neste caso, uma entidade alvo recebe uma mensagem de texto via rádio de um objeto satélite (entidade dentro do VR-Forces) e uma mensagem de texto via rádio do federado conectado.

1. Clique com o botão direito no satélite. Uma janela de opções é exibida.
2. Clique em **Information...**. A janela de informações do satélite é exibida.
3. No campo **Object Console**, clique na seta (à direita) para expandir as opções de console.
4. No campo **Notify Level**, selecione Debug. A figura 4.11 mostra a tela desejada.

Com este modo, é possível ver as mensagens de texto que o satélite recebe.

4.2 Conectando um federado

Na seção 4.1, foi visto que o VR-Forces já é configurado para conectar-se ao MAK RTI. Nesta seção, por sua vez, é mostrado como um federado, já implementado segundo o

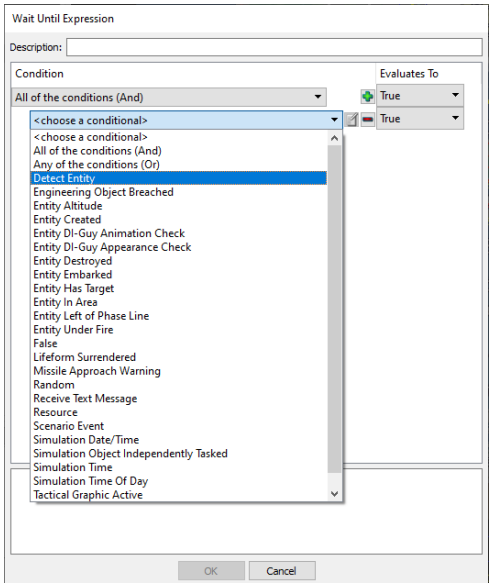


FIGURA 4.9 – Condicional do satélite: mandar a mensagem apenas após detectar uma unidade inimiga

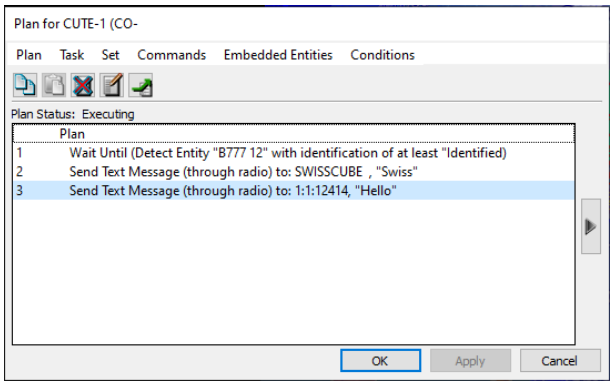


FIGURA 4.10 – Janela de plano do satélite

protocolo HLA e utilizando por padrão o RPR FOM 2.0 é inserido na mesma simulação. A conexão pode ser verificada no rtiexec, ou interface gráfica do MAK RTI, como na figura 3.3.

4.2.1 Compatibilidade com o *java*

O simulador *Federado Aeronave RPR 2.0* possui como programa principal o *HLAAircraftRPR20Federate.java*. No *Eclipse*, a opção **Run → Run Configurations** abre a janela de configurações de execução de código (figura 4.12).

Na aba *Main*, foi configurado *HLAAircraftRPR20Federate* como o código principal. Na aba *Classpath*, foi adicionado o arquivo *jar* correspondente ao MAK RTI.

O sistema operacional, por sua vez, necessita saber como utilizar corretamente os processos que irá executar. Para isso, as variáveis de ambiente precisam estar devidamente configuradas. No *Windows*, as variáveis de ambiente são mostradas conforme a figura 4.13.

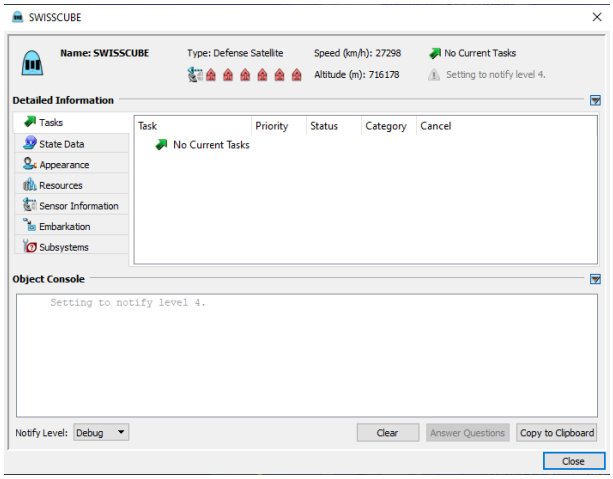


FIGURA 4.11 – Informações de *debug* para um objeto no VR-Forces

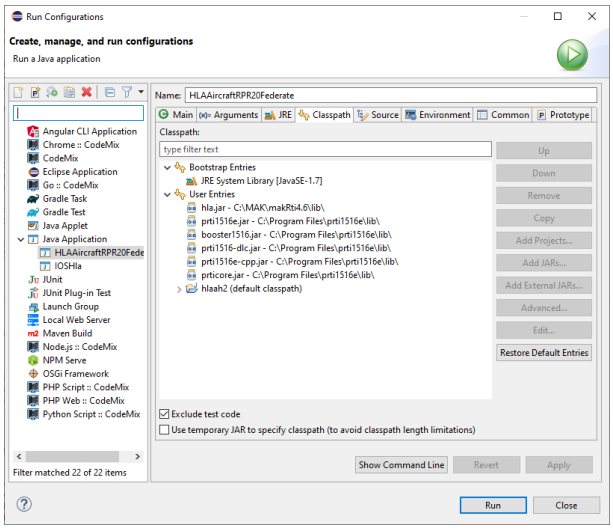


FIGURA 4.12 – Janela de configuração de execução do Eclipse

Na variável *java*, foi inserido o caminho do arquivo executável java jdk-16-02. Na variável *Path*, foi adicionado o caminho do executável RTI da MAK. A figura 4.14 mostra a janela por onde se pode inserir e editar uma variável de ambiente pelo Windows.

O programa Federado Aeronave RPR 2.0 foi escrito baseado na versão 1.8 do *java*, e utilizar uma versão diferente pode causar problemas de compatibilidade no momento da execução do código. Para resolver este problema, é necessário ter esta versão do *java* na máquina com a qual o programa será executado, e configurar a IDE para que ela utilize o compilador correto.

No ambiente de desenvolvimento *Eclipse*, na aba *Window*, clique em *Preferences*. A janela de preferências da IDE *Eclipse* é aberta (figura 4.15).

Na aba *Java* → *Compiler* há um aviso que diz para "certificar-se de ter um JRE compatível instalado e ativado". Clique em *Configure*. Uma janela para inserir a versão do compilador é aberta (figura 4.16).

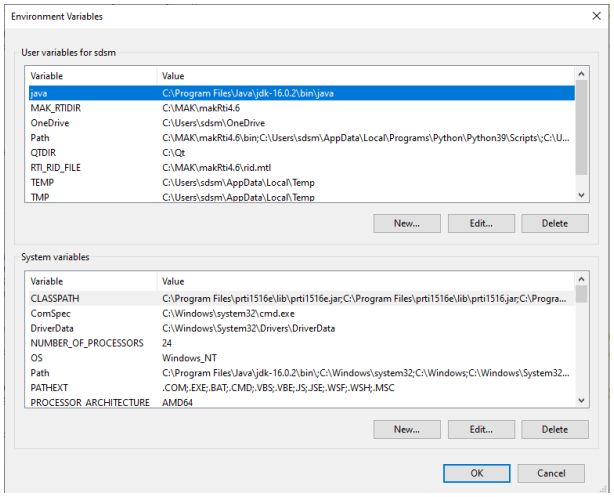


FIGURA 4.13 – Janela de configuração de variáveis de ambiente no Windows

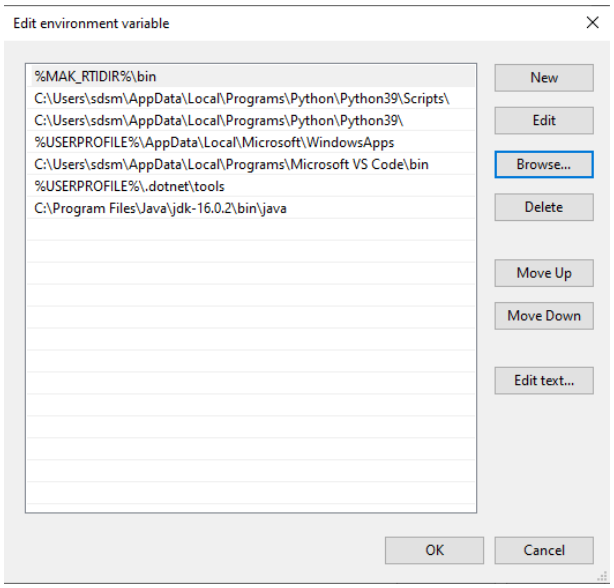


FIGURA 4.14 – Variável de ambiente *Path* no Windows

Nesta janela, clique em *Edit* e coloque o caminho do compilador *java* versão 1.8, conforme a imagem 4.16.

4.2.2 Compatibilidade do código com RTI / Demais nós da simulação

No método *configurePropertiesRPR20* do código *HLAAircraftRPR20Federate.java* (listado em 4.1), são definidos o nome do federado na simulação e em qual federação ele é conectado. Na linha 5, o método *setFederationName* deve conter o mesmo nome da federação que o MAK RTI (ver figura 3.3). Além disso, ele define as propriedades básicas concernentes ao federado e adiciona as configurações da classe *RPR20AIRCRAFT* (linha 12).

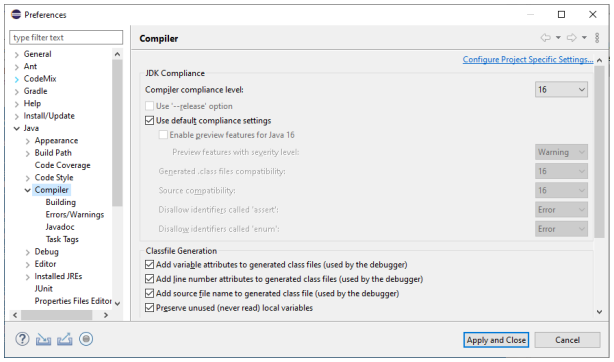


FIGURA 4.15 – Janela de Preferências do *Eclipse* (aba *Java* → *Compiler*

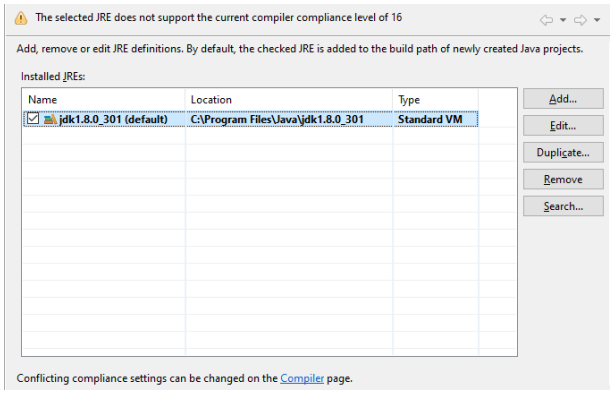


FIGURA 4.16 – Ambiente de edição da versão *java* para o compilador da IDE

```
1     private void configurePropertiesRPR20(HLADummyFederate window)
          {
2         List<String> foms = new ArrayList<String>();
3
4         window.getPropClass().setFederateName("Espaconave RPR 2.0"
              );
5         window.getPropClass().setFederationName("MAK-RPR-2.0");
6
7         window.getPropClass().setFomPaths(foms);
8
9         HlaInterface hlaInterface = HlaInterface.Factory.
              newInterface();
10        window.setHlaInterface(hlaInterface);
11        List<String> updatedClasses = new ArrayList<String>(window
              .getPropClass().getSubscribedClasses());
12        updatedClasses.add(RprClassFactory.RPR20AIRCRAFT);
13        window.getPropClass().setPublishedClasses(updatedClasses);
14    }
```

Trecho de código 4.1 – Código de configuração da conexão do federado

A figura 4.17 mostra a interface gráfica do federado. Para iniciar a simulação, clica-

se o botão *Começar*, momento a partir do qual o programa conecta-se ao RTI, de um modo similar ao descrito pela subseção 4.1.1 e pela imagem 4.2. Os campos do programa correspondem às seguintes propriedades:

- RTI: A qual RTI o programa se conecta. Este programa é compatível, além do MAK RTI, ao Pitch RTI e ao Portico RTI.
- CrcAddress: Apenas necessário para conectar-se ao Pitch RTI. Corresponde à porta na qual o Central RTI Component (CRC) da Pitch está configurada.
- Federation name: Nome da federação. Deve ser o mesmo para todos os federados que se desejam conectar à simulação.
- Federate name: Nome do federado. Deve ser único dentro de uma federação.
- Published classes: Classes que são publicadas no início da execução, com seus respectivos atributos e *data types*. O trecho 4.2 mostra o código, contido no arquivo *RPRClassFactory.java*, na qual à classe RPR20SPACECRAFT são adicionados esses valores.
- Published events: Eventos que serão publicados no início da execução.
- FOMs: FOMs necessários para executar o programa. Para este programa, foram necessários os FOMs "RPR-Base_v2.0_draft19.0", "RPR-Physical_v2.0_draft19.0"e "RPR-Communication_v2.0_draft19.0"

```
1   if(rprClass.equals(RPR20SPACECRAFT)){
2     List<String> attributes = new ArrayList<String>();
3     attributes.add("EntityType");
4     attributes.add("EntityIdentifier");
5     attributes.add("IsPartOf");
6     attributes.add("Spatial");
7     attributes.add("RelativeSpatial");
8     attributes.add("ForceIdentifier");
9     attributes.add("Marking");
10
11    List<String> datatypes = new ArrayList<String>();
12    datatypes.add("EntityTypeStruct");
13    datatypes.add("EntityIdentifierStruct");
14    datatypes.add("IsPartOfStruct");
15    datatypes.add("SpatialVariantStruct");
16    datatypes.add("SpatialVariantStruct");
17    datatypes.add("ForceIdentifierEnum8");
```

FIGURA 4.17 – Interface gráfica do programa Aeronave RPR 2.0

```
18        datatypes.add("MarkingStruct");
19
20        HlaGenericClass hlaGenericClass = new HlaGenericClass("
              HLAobjectRoot.BaseEntity.PhysicalEntity.Platform.
              Spacecraft", attributes, datatypes);
21        return hlaGenericClass;
22      }
23    }
```

Trecho de código 4.2 – Criação de uma classe HLA com os parâmetros do objeto Spacecraft segundo o RPR FOM 2.0

4.3 Publicações e Subscrições: Colocando um objeto na simulação

Nesta seção, é visto como se cria uma classe do tipo *Object* na simulação. Com isso, é possível inserir o federado como uma entidade do VR-Forces, porém controlado externamente pelo código do programa. A figura 4.18 mostra um fluxograma das principais tarefas que são necessárias para se publicar esse objeto. São mostrados os principais trechos de código no qual a classe é criada e nos quais os comandos ao *Object* são enviados.

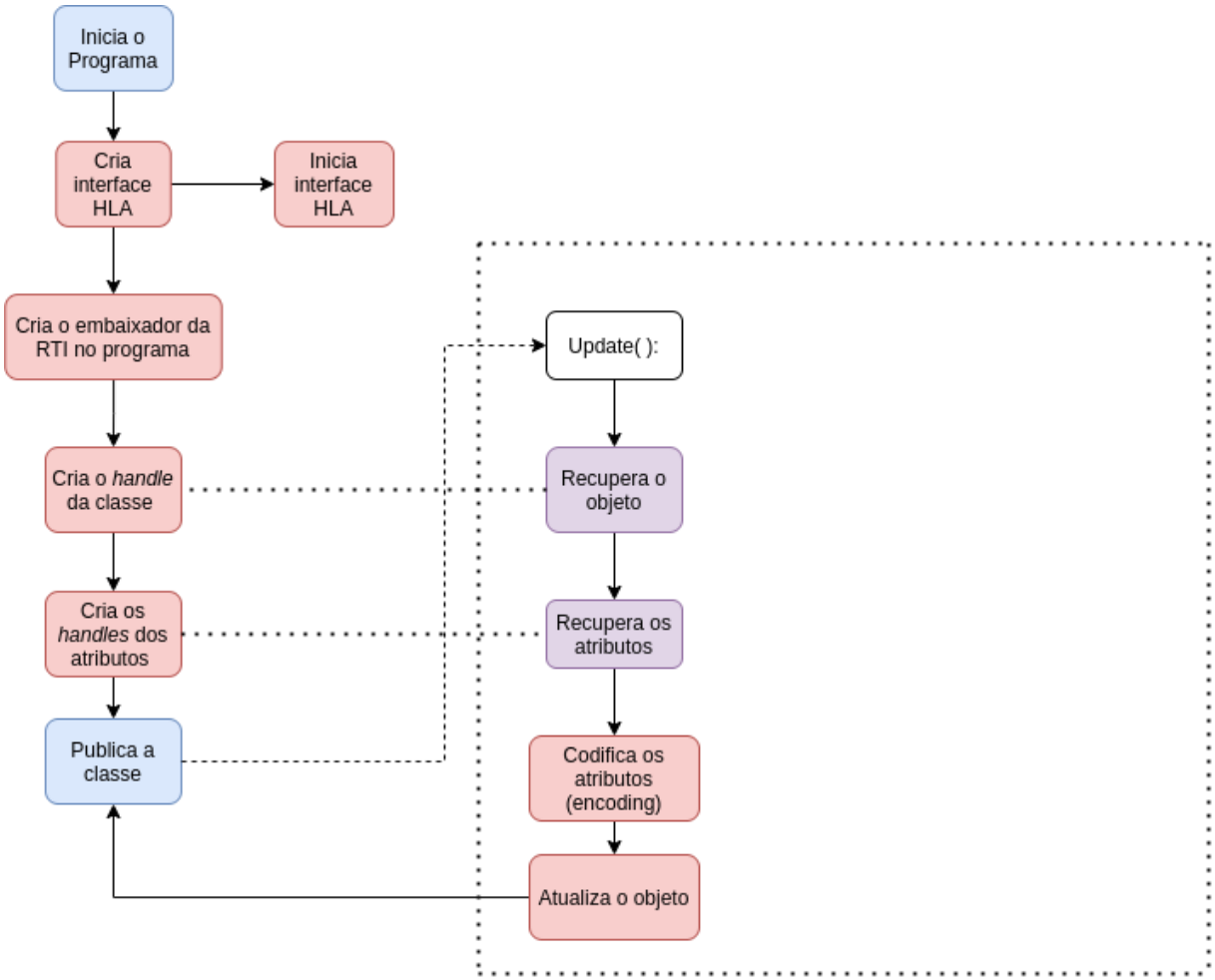


FIGURA 4.18 – Fluxograma de publicação e atualização de um objeto na simulação

O código listado 4.3 mostra como é criado e publicado um *Object*. O objeto em questão é o *Spacecraft,* conforme o RPR FOM 2.0 (figura 3.4. No arquivo *HLADummyFederate.java,* a *thread WorkerThread* é chamada pelo método *initialize()* desse mesmo arquivo. Uma *thread* é um processo paralelo dentro de um programa (ORACLE..., 2021). O método *doInBackground(),* então, cria uma interface HLA e chama o **embaixador** *(ambassador)* da RTI. Em seguida, a interface HLA é iniciada (linha 10), passando os parâmetros necessários (FOMs, nome do federado e da federação, conforme descrito na seção 4.2).

O embaixador é a classe responsável num federado de fazer a sua conexão com o RTI (MöLLER, 2012). Comandos de entrar e sair da simulação, bem como todas as publicações e subscrições do federado são feitas pelo embaixador. Nesta função, o embaixador é usado para criar um objeto da classe *HlaObjectClassFactory* (linha 12) o qual, por sua vez, cria o objeto *hoc* da classe *HlaObjectClass* (linha 17).

Factory é um padrão de projeto feito para criar objetos encapsulando a lógica de criação em outra classe ou método (PADRãO..., 2021). Deste modo, a criação do objeto *hoc* da classe *HlaObjectClass* é delegado à classe *HlaObjectClassFactory* (linha 17). Este padrão é utilizado ao longo de todo o código do programa Federado Aeronave RPR 2.0.

Após criar o objeto da classe *HLAObject,* o programa publica esse objeto na interface HLA (linha 19). Para se publicar e subscrever objetos, entretanto, a HLA exige que se criem manuseadores *(handles)* para a classe (linha 20) e para os atributos (código 4.5). Em seguida, é criada e iniciada a *thread PublisherThread* (linhas 21 e 22).

```
1   public class WorkerThread extends SwingWorker<Void, Void> {
2   @Override
3   protected Void doInBackground() throws Exception {
4     HlaInterface hlaInterface = HlaInterface.Factory.
        newInterface();
5     HLADummyFederate.this.setHlaInterface(hlaInterface);
6
7     RTIambassador ambassador = hlaInterface.getAmbassador();
8     PropClass propClass = HLADummyFederate.this.getPropClass();
9
10    hlaInterface.start("", propClass.getFomPaths(), propClass.
        getFederationName(), propClass.getFederateName());
11
12    HlaObjectClassFactory hocfactory = new HlaObjectClassFactory
        (ambassador);
13
14    for(String classToBePublished : propClass.
        getPublishedClasses()){
15      HlaGenericClass hlaClass = RprClassFactory.create(
          classToBePublished);
16      // Ponto de criacao do objeto
17      HlaObjectClass hoc = hocfactory.create(hlaClass);
18      // Ponto de publicacao do objeto
19      hlaInterface.publish(hoc);
20      ObjectInstanceHandle object = hoc.createObject("TestObject
          " + System.currentTimeMillis());
```

```
21          createPublisherThread(classToBePublished, hoc, object);
22          publisherThread.start();
23        }
24
25        eventsListModel.addElement("Inicializado.");
26
27        return null;
28      }
```

Trecho de código 4.3 – *WorkerThread:* criação e publicação de uma classe tipo *Object* utilizando HLA

Na linha 19 do código em 4.3, é chamado o método *publish* da classe *HLAInterface*. O código desse método é mostrado em 4.4, que encontra-se no arquivo *HlaInterfaceImpl.java*. Aqui, o objeto HLA é inserido na lista de classes publicadas, e o embaixador publica os atributos da classe com o método *getAttributeHandleSet()*. Os atributos desse objeto estão listados em 4.5, e estão no arquivo *RprClassFactory.java*.

```
1
2   @Override
3   public void publish(HlaObjectClass hoc) throws
       AttributeNotDefined, ObjectClassNotDefined, SaveInProgress,
       RestoreInProgress, FederateNotExecutionMember, NotConnected,
       RTIinternalError {
4     federatePublishedClasses.put(hoc.getClassHandle(), hoc);
5     ambassador.publishObjectClassAttributes(hoc.getClassHandle(),
         hoc.getAttributeHandleSet());
6   }
```

Trecho de código 4.4 – Publicação dos atributos da classe *Spacecraft* do RPR FOM 2.0

```
1     List<String> attributes = new ArrayList<String>();
2       attributes.add("EntityType");
3       attributes.add("EntityIdentifier");
4       attributes.add("IsPartOf");
5       attributes.add("Spatial");
6       attributes.add("RelativeSpatial");
7       attributes.add("ForceIdentifier");
8       attributes.add("Marking");
9
10      List<String> datatypes = new ArrayList<String>();
11      datatypes.add("EntityTypeStruct");
```

```
12      datatypes.add("EntityIdentifierStruct");
13      datatypes.add("IsPartOfStruct");
14      datatypes.add("SpatialVariantStruct");
15      datatypes.add("SpatialVariantStruct");
16      datatypes.add("ForceIdentifierEnum8");
17      datatypes.add("MarkingStruct");
18
19      HlaGenericClass hlaGenericClass = new HlaGenericClass("
           HLAobjectRoot.BaseEntity.PhysicalEntity.Platform.
           Spacecraft", attributes, datatypes);
20      return hlaGenericClass;
```

Trecho de código 4.5 – Listagem de todos os atributos da classe *Spacecraft* do RPR FOM 2.0

Uma vez publicado o objeto, ele começa a participar da publicação. O VR-Forces cria uma entidade do tipo satélite, a qual é controlada pelo federado. Como descrito na seção 3.5, essa entidade tem por atividade deslocar-se pelo mapa, variando sua posição de latitude e longitude para um dado intervalo de tempo.

Isso é feito com o método *Update(),* do arquivo *HlaObjectPublisherDummyAircraftRPR20*. Esse método é chamado pela *thread publisherThread,* chamada na linha 22 do código 4.3. O método chamado na linha 18 leva o conjunto objeto/atributos para a função *updateObject* (código listado em 4.7). Dentre esses atributos, aquele que muda seus valores conforme o passo da simulação é o *SpatialVariantStruct,* e o trecho de código em que ele faz isso foi mostrado em 3.1.

Após atualizar o objeto, o programa cria uma *string* contendo as suas informações e o mostra na tela da interface gráfica do programa (campo em branco da figura 4.17).

```
1     @Override
2   protected String update() {
3       String message = "";
4       Map<String,Object> values = new HashMap<String,Object>();
5       EntityTypeStruct ets = createEntityTypeStruct();
6       SpatialVariantStruct spatial = createSpatialVariantStruct();
7       EntityIdentifierStruct eis = createIdentifierStruct();
8       byte forceIdentifier = (byte) 1;//friendly
9       MarkingStruct markingStruct = createMarkingStruct();
10
11      values.put("EntityType",ets);
12      values.put("Spatial",spatial);
13      values.put("EntityIdentifier",eis);
```

```
14        values.put("ForceIdentifier",forceIdentifier);
15        values.put("Marking",markingStruct);
16
17        try {
18          hoc.updateObject(object, values);
19          message += ets.toString();
20          message += spatial.toString();
21          message += eis.toString();
22          message += "[Force: " + forceIdentifier + "]";
23          message += markingStruct;
24
25        } catch (FederateNotExecutionMember | NotConnected |
              AttributeNotOwned
26            | AttributeNotDefined | ObjectInstanceNotKnown |
                SaveInProgress
27            | RestoreInProgress | RTIinternalError e) {
28
29          String updatedMessage = "Erro:";
30          Runnable updateList = new GuiListModelAdder(model,
                updatedMessage, new Date());
31          SwingUtilities.invokeLater(updateList);
32
33          e.printStackTrace();
34        }
35        return message;
36    }
```

Trecho de código 4.6 – Implementação do método *update,* que atualiza o estado do objeto conforme o passo da simulação

No arquivo *HlaObjectClass,* a função *updateObject* atualiza o estado do objeto na simulação. Conforme dito anteriormente, modificações dessa natureza são realizadas através do embaixador. Fornecendo todos os *handles* da classe e dos parâmetros, o embaixador executa o método *updateAttributeValues* (linha 11 do código 4.7).

```
1     public void updateObject(ObjectInstanceHandle objectHandle,
          Map<String,Object> values) throws
          FederateNotExecutionMember, NotConnected, AttributeNotOwned
          , AttributeNotDefined, ObjectInstanceNotKnown,
          SaveInProgress, RestoreInProgress, RTIinternalError{
2     int size = values.size();
3     AttributeHandleValueMap attributeHandleValueMap = ambassador.
```

```
        getAttributeHandleValueMapFactory().create(size);
4     for(String value : values.keySet()){
5       try{
6         attributeHandleValueMap.put(attributeMap.get(value),
             attributeCodersMap.get(value).encode(values.get(value))
             );
7       }catch(Exception e){
8         System.out.println("Invalid coder for attribute "+ value);
9       }
10    }
11    ambassador.updateAttributeValues(objectHandle,
         attributeHandleValueMap, null);
12  }
```

Trecho de código 4.7 – Atualização do objeto na RTI

Os parâmetros, ou atributos, por sua vez, estão descritos na *struct* dos seus respectivos códigos. Para que os dados sejam enviados corretamente por HLA, é necessário transformar a *struct* do código numo formato padrão do protocolo. O nome desta ação é *encode*. Na linha 6 do código 4.7, o método *encode* é chamado para cada um dos parâmetros do objeto. O código 4.8 mostra o exemplo do atributo *WorldLocation*. O método *WorldLocationStructCoder* cria as posições x y z do objeto como uma variável do tipo *HLAfloat64BE*. Em seguida, o programa guarda esses valores na lista *hlaFixedRecord*. O *encode* deste parâmetro consiste em devolver essa lista com as posições do objeto.

```
1     public WorldLocationStructCoder(EncoderFactory ef) {
2     super(ef);
3
4     hlaFixedRecord = encoderFactory.createHLAfixedRecord();
5
6     x = encoderFactory.createHLAfloat64BE();
7     y = encoderFactory.createHLAfloat64BE();
8     z = encoderFactory.createHLAfloat64BE();
9
10    hlaFixedRecord.add(x);
11    hlaFixedRecord.add(y);
12    hlaFixedRecord.add(z);
13  }
14
15    @Override
16  public byte[] encode(Object objectToEncode) {
```

```
17      return hlaFixedRecord.toByteArray();
18    }
```

Trecho de código 4.8 – *Encoding* um atributo de um *Object*

Do mesmo modo que, para enviar os parâmetros por HLA é preciso fazer o *encoding,* para ler esses parâmetros no federado é preciso fazer o processo inverso, *decoding.* Este programa não faz leitura de classes do tipo *Object,* porém o método *decode* do parâmetro *WorldLocation* é implementado (código 4.9). O programa recebe um *array* de *bytes* que é decodificado para a forma da *struct* deste atributo (linha 5), por meio da variável *hlaFixedRecord*. Em seguida, os valores x y z do tipo *HLAfloat64BE* são extraídos da lista *hlaFixedRecord*. Por fim, a variável *wls* monta a *Struct* definindo as posições x y z em forma de *floats* (linhas 8, 11 e 14).

```
1   @Override
2   public WorldLocationStruct decode(byte[] bytes) throws
       DecoderException {
3     WorldLocationStruct wls =new WorldLocationStruct();
4
5     hlaFixedRecord.decode(bytes);
6
7     HLAfloat64BE x = (HLAfloat64BE) hlaFixedRecord.get(0);
8     wls.setX(x.getValue());
9
10    HLAfloat64BE y = (HLAfloat64BE) hlaFixedRecord.get(1);
11    wls.setY(y.getValue());
12
13    HLAfloat64BE z = (HLAfloat64BE) hlaFixedRecord.get(2);
14    wls.setZ(z.getValue());
15
16    return wls;
17  }
```

Trecho de código 4.9 – *Decoding* um atributo de um *Object*

4.4 Publicações e Subscrições: Criando uma interação na simulação

Nesta seção, é visto como se cria uma classe do tipo *Interaction* na simulação. Com isso, é possível enviar e receber informações de e para outro federado (que, neste caso, é o VR-Forces GUI). A figura 4.19 mostra um fluxograma de como essa informação tramita entre os nós da simulação e quais são os principais passos que o *Federado* executa para que isso aconteça.

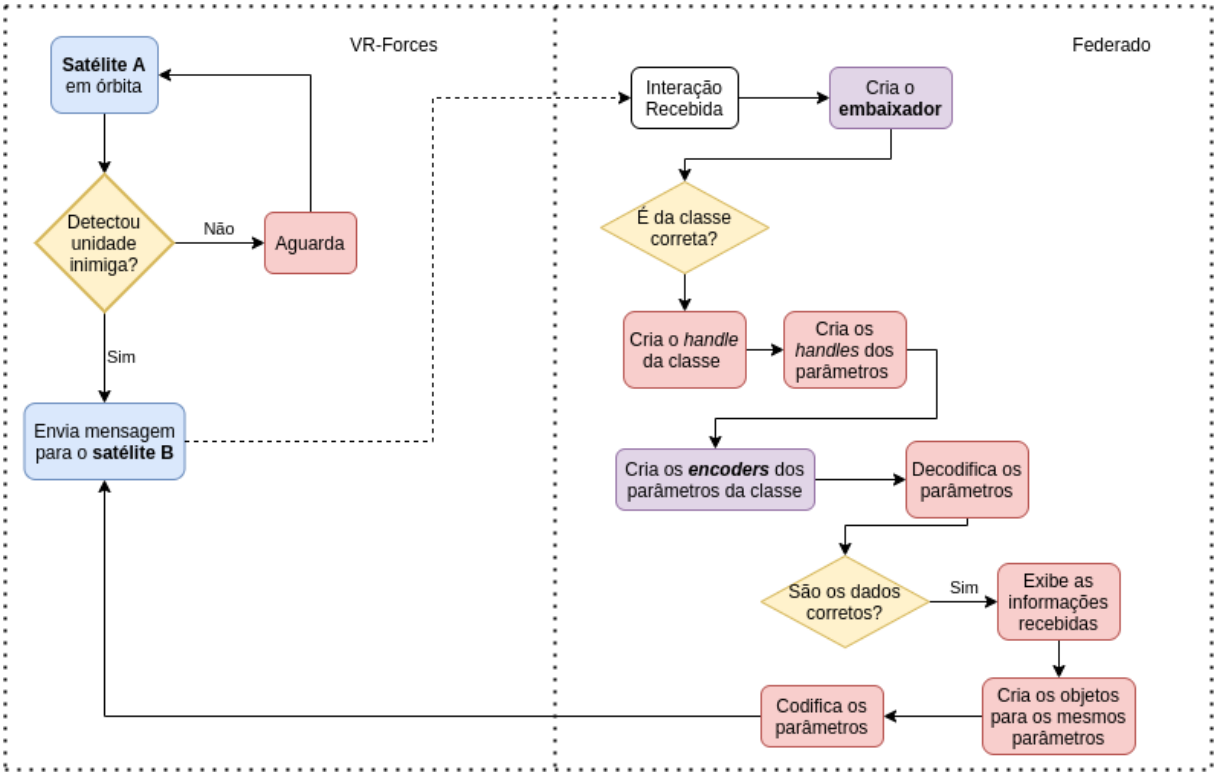


FIGURA 4.19 – Fluxograma do funcionamento das interações entre os nós da simulação

A interação utilizada para fazer a comunicação entre os federados corresponde à classe *ApplicationSpecificRadioSignal*. Ela possui seus respectivos atributos que, por sua vez, possuem seus respectivos tipos de variáveis. Essas informações são dadas pelo FOM, e é com base nelas que o programa é escrito. A figura 4.20 mostra o FOM dessa classe, vista por meio do *software* Pitch Visual OMT.

Aqui, são mostrados os principais trechos de código no qual a classe é criada e nos quais os pacotes são manipulados. Para se publicar e subscrever interações, a HLA exige que se criem manuseadores *(handles)* para a classe e todos os seus atributos. O trecho de código 4.10 mostra a criação dos *handles* no arquivo *HLAInterfaceImpl.java*.

```
1     private InteractionClassHandle
          applicationSpecificRadioSignalInteractionClassHandle;
2   private ParameterHandle     hostRadioIndexParameterHandle;
```

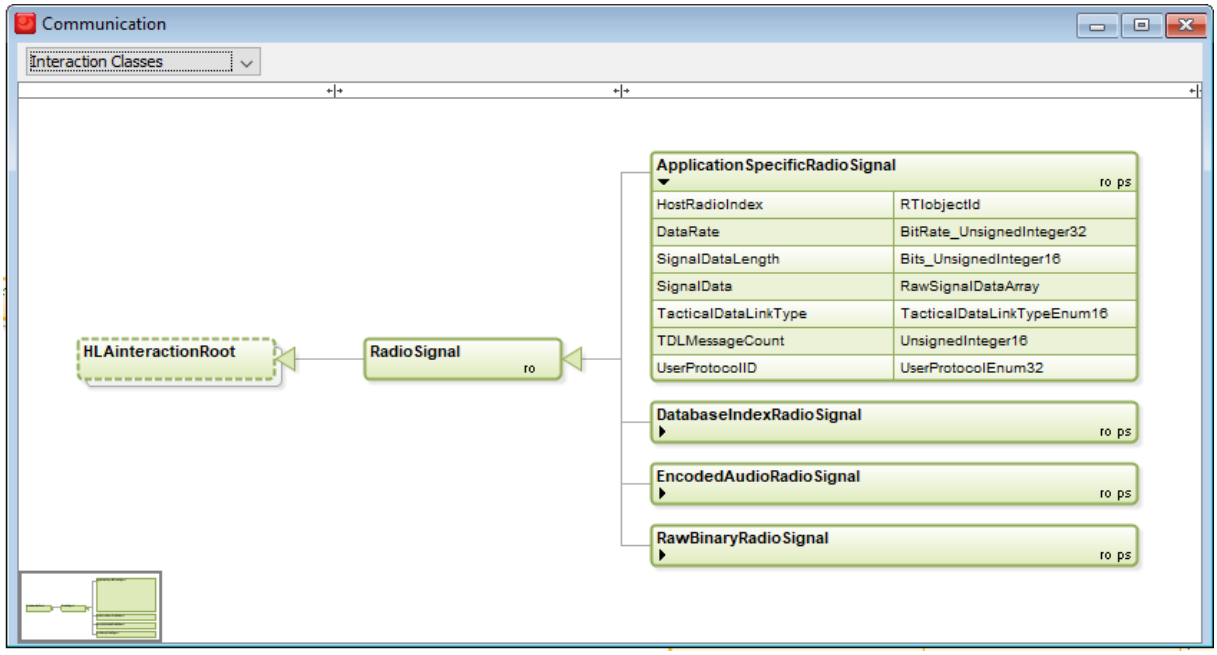


FIGURA 4.20 – Diagrama da classe ApplicationSpecificRadioSignal do RPR FOM 2.0

```
3  private ParameterHandle     dataRateParameterHandle;
4  private ParameterHandle     signalDataLengthParameterHandle;
5  private ParameterHandle     signalDataParameterHandle;
6  private ParameterHandle     userProtocolIDParameterHandle;
7  private ParameterHandle     tacticalDataLinkTypeParameterHandle;
8  private ParameterHandle     tdlMessageCountParameterHandle;
```

Trecho de código 4.10 – Criação do *Interaction Class Handle* e dos *Parameter Handles* no federado

Uma vez declaradas as variáveis, é necessário que elas correspondam às respectivas *handles*. Para isso, utiliza-se o embaixador *(ambassador)*, que realiza as chamadas *(calls)* e os *callbacks* na RTI. As funções chamadas no método *getHandles()* do trecho 4.11 que realizam isso são *getInteractionClassHandle* e *getParameterHandle*.

```
1    private void getHandles() throws RTIinternalError,
        FederateNotExecutionMember, NotConnected {
2    try {
3      applicationSpecificRadioSignalInteractionClassHandle =
          ambassador.getInteractionClassHandle("HLAinteractionRoot.
          RadioSignal.ApplicationSpecificRadioSignal");
4      hostRadioIndexParameterHandle = ambassador.
          getParameterHandle(
          applicationSpecificRadioSignalInteractionClassHandle, "
          HostRadioIndex");
5      dataRateParameterHandle = ambassador.getParameterHandle(
```

```
            applicationSpecificRadioSignalInteractionClassHandle, "
            DataRate");
        /* Trecho de codigo omitido. Contem as chamadas de todos
            Parameter Handles contidos no trecho anterior. A
            estrutura segue conforme a linha de codigo acima. */

    } catch (NameNotFound e) {
        System.out.println("Erro getHandles():" + e);
        throw new RTIinternalError("HlaInterfaceFailure", e);
    } catch (InvalidInteractionClassHandle e) {
        throw new RTIinternalError("HlaInterfaceFailure", e);
    }
}
```

Trecho de código 4.11 – Função que determina os *handles* em suas respectivas variáveis

A HLA realiza as interações entre os simuladores por meio de publicações *(publishings)* e subscrições *(subscribings)*, as quais são mediadas pela RTI. Como esta interação recebe e manda mensagens de texto, ela precisa que ambas as funções estejam implementadas no código. O trecho 4.12 mostra os métodos *subscribeInteractions()* e *publishInteractions()*, nas quais o *ambassador* sinaliza à RTI que interações serão mandadas (publish) e recebidas (subscribe) pelo federado.

```
 private void subscribeInteractions() throws
    FederateNotExecutionMember, RestoreInProgress,
    SaveInProgress, NotConnected, RTIinternalError,
    FederateServiceInvocationsAreBeingReportedViaMOM {
  try {
    ambassador.subscribeInteractionClass(
        applicationSpecificRadioSignalInteractionClassHandle);
  } catch (InteractionClassNotDefined e) {
    throw new RTIinternalError("HlaInterfaceFailure", e);
  }
}

private void publishInteractions() throws
   FederateNotExecutionMember, RestoreInProgress, SaveInProgress
   , NotConnected, RTIinternalError {
  try {
    ambassador.publishInteractionClass(
        applicationSpecificRadioSignalInteractionClassHandle);
```

```
12      } catch (InteractionClassNotDefined e) {
13        throw new RTIinternalError("HlaInterfaceFailure", e);
14      }
15
16    }
```

Trecho de código 4.12 – Funções publish e subscribe interaction, a partir das quais é possível se comunicar utilizando a classe ApplicationSpecificRadioSignal

Com isso, as interações são tratadas na função *receiveInteraction()*. Como a classe de interesse é *applicationSpecificRadioSignal*, a função irá realizar alguma tarefa apenas se a interação recebida for dessa classe. A RTI recebe e envia dados por meio de um *array* de *bytes*. Para transformar esses *bytes* na informação desejada, é necessário criar um decodificador *coder / decoder*. Para criar um objeto da classe *coder,* é necessário um objeto do tipo *rtiFactory,* com o qual se pode criar instâncias com o tipo de variável correspondentes ao protocolo HLA, conforme listado no FOM (ver figura 4.20).

```
1     private void receiveInteraction(InteractionClassHandle
         interactionClass, ParameterHandleValueMap theParameters) {
2     if (interactionClass.equals(
         applicationSpecificRadioSignalInteractionClassHandle)) {
3       RtiFactory rtiFactory;
4       try {
5         rtiFactory = RtiFactoryFactory.getRtiFactory();
6         EncoderFactory factory = rtiFactory.getEncoderFactory();
7         SignalDataLengthCoder sdlCoder = new SignalDataLengthCoder
             (factory);
8         SignalDataCoder sdCoder = new SignalDataCoder(factory);
9         HostRadioIndexCoder hriCoder = new HostRadioIndexCoder(
             factory);
```

Trecho de código 4.13 – Função *receiveInteraction()*: criação dos *coders* da interação *applicationSpecificRadioSignalInteraction*

O trecho de código 4.14 mostra como é o codificador / decodificador do parâmetro *signalData*, que é aquela que contém a mensagem de texto enviada. A classe *encode* transforma a informação do parâmetro num pacote de dados tal que possa ser enviado via RTI e a classe *decode* transforma o pacote de dados recebido numa informação interpretável pelo programa.

```
1     public byte[] encode(byte[] data, String textMessageIdentifier
         , String newText) {
2         String s = new String(data);
3         s = s.substring(0, s.indexOf(textMessageIdentifier) +
             textMessageIdentifier.length() + 1) + newText + "\0";
```

```
4         return s.getBytes();
5       }
6
7          public String decode(byte[] data) throws DecoderException
               {
8             String dataString = new String(data, StandardCharsets.
                 UTF_8);
9             return dataString;
10      }
```

Trecho de código 4.14 – Codificador / decodificador das informações do parâmetro *signalData*

Voltando à função *receiveInteraction(),* após criar os objetos *coders,* utiliza-se os métodos *decode* implementados em cada uma das classes para que se extraia as informações corretas dos parâmetros (código 4.15). Para essa interação, o parâmetro mais importante é o *signalData*. A variável mensagem (linha 6) recebe uma *string* na qual cada caractere corresponde a um *byte* recebido do parâmetro *signalData*.

```
1           try {
2           int signalDataLength = sdlCoder.decode(theParameters.get
               (signalDataLengthParameterHandle));
3           int tdlMessageCount  = sdlCoder.decode(theParameters.get
               (tdlMessageCountParameterHandle));
4           String hostRadioIndex = hriCoder.decode(theParameters.
               get(hostRadioIndexParameterHandle));
5           byte[] signalData = theParameters.get(
               signalDataParameterHandle);
6           String mensagem = sdCoder.decode(signalData);
```

Trecho de código 4.15 – Função *receiveInteraction()*: utilização do método *decode* para os parâmetros da interação

Além de mensagens de texto, a interação *ApplicationSpecificRadioSignal* é utilizada pelo VR-Forces não apenas para troca de mensagens de rádio, mas para outras atividades, como relatórios de identificação de alvos *(spot reports)*. Ademais, a mensagem enviada no parâmetro *signalData* não mostra, mesmo no caso em que se trata de uma mensagem de texto, apenas a mensagem, mas também traz consigo um cabeçalho cheio de informações redundantes e valores de bytes que estão fora da tabela ASCII e, portanto, ao passar esses dados para a *string,* não apresentam um conteúdo legível.

Por inspeção, descobriu-se que a *substring* contendo a mensagem de rádio desejada aparecia no final da *string,* logo após a substring *"text-report"*. O código da função *receiveInteraction(),* portanto, utiliza essas informações para apenas captar a mensagem de rádio de fato, utilizando as funções *substring()* e *indexOf* da classe *String* do *java.* (trecho

de código 4.16). O programa, então, exibe no *console* da aplicação que o executa a dita mensagem mais alguns parâmetros, conforme a figura 4.21.

```
1           String textMessageIdentifier = "text-report";
2
3           /* Caso a interacao seja uma mensagem de texto, o
               programa mostra os dados recebidos e envia outra
               mensagem de texto de volta (mensagemRecebida) */
4           if(mensagem.indexOf(textMessageIdentifier) > 0) {
5             System.out.println("Signal Data Length: " +
                 signalDataLength);
6             System.out.println("TDL Message Count: " +
                 tdlMessageCount);
7             System.out.println("Host Radio Index: " +
                 hostRadioIndex);
8             System.out.print("Radio Data: " + mensagem.substring(
                 mensagem.indexOf(textMessageIdentifier) +
                 textMessageIdentifier.length() + 1));
9             System.out.println("Fim da msg.");
```

Trecho de código 4.16 – Função *receiveInteraction()*: manipulação de strings para extrair a mensagem de rádio enviada pelo VR-Forces. Mostra também outros parâmetros relevantes

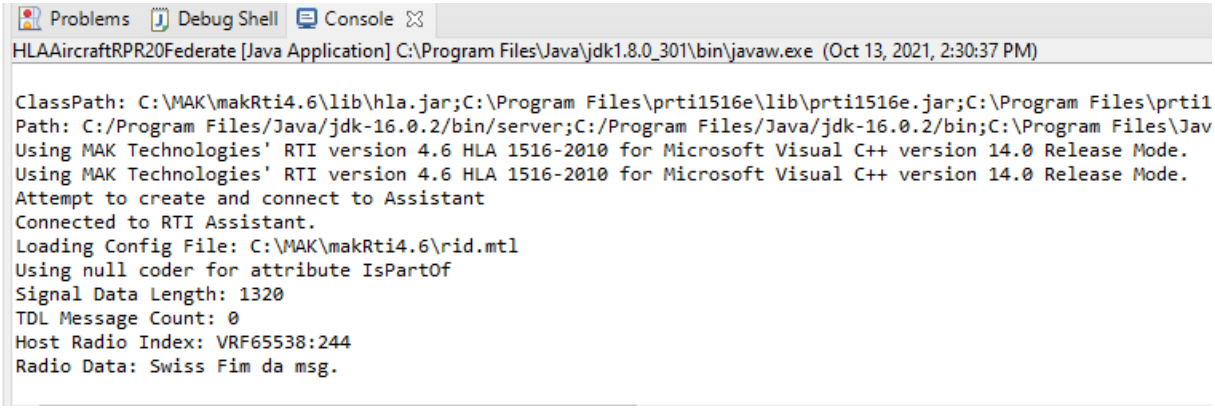


FIGURA 4.21 – Console da *Eclipse IDE* no qual é mostrada a informação recebida pelo VR-Forces

A condição escolhida para fazer o caminho reverso, isto é, mandar uma mensagem de rádio do Federado de volta para uma entidade do VR-Forces foi simplesmente receber uma mensagem do mesmo tipo. Logo, no código da função *receiveInteraction(),* após os comandos de mostrar os resultados em *console,* ele cria as variáveis dos parâmetros que serão enviados de volta. Do mesmo modo que a primeira mensagem, todos os parâmetros devem estar de acordo com o padrão HLA daquela classe estabelecidos pelo FOM correspondente. Logo, as variáveis criadas deverão ser codificadas (função *encode* das res-

pectivas classes). Para este caso, entretanto, os únicos parâmetros modificados em relação à mensagem que chega no federado são *signalData* e *signalDataLength*.

O trecho de código 4.17 mostra como foram feitas essas mudanças. É criada uma nova mensagem, "Mensagem enviada pelo federado". Em seguida, o *Array* de *bytes modifiedDataBytes* (linha 5) recebe o resultado do *encoding*, que está mostrado no trecho 4.14. Uma vez que a documentação do VR-Forces não apresenta uma descrição extensiva da função responsável por criar o conteúdo do cabeçalho do parâmetro *signalData*, a solução encontrada foi trocar apenas os últimos *bytes* do parâmetro que correspondiam ao texto da mensagem de fato. Para fazer o *encoding* da função *signalDataLength*, portanto, passou-se como argumento o próprio tamanho da mensagem modificada (multiplicada por 8, pois a função dá o tamanho dos dados em *bits*).

Em seguida, foi chamada a função *sendRadioMessage,* que junta todos os parâmetros da classe e envia a interação via RTI (código 4.19).

```
1                   try {
2               String mensagemModificada = new String(signalData);
3               mensagemModificada = "Mensagem enviada pelo federado
                    ";
4
5               byte[] modifiedDataBytes = sdCoder.encode(signalData
                    , textMessageIdentifier, mensagemModificada);
6
7               byte[] modifiedSignalDataLengthBytes = sdlCoder.
                    encode( (short) (modifiedDataBytes.length*8));
8
9               sendRadioMessage(modifiedSignalDataLengthBytes,
                    theParameters.get(tdlMessageCountParameterHandle)
                    ,
10                   theParameters.get(hostRadioIndexParameterHandle)
                        , modifiedDataBytes,
11                   theParameters.get(
                        tacticalDataLinkTypeParameterHandle),
                        theParameters.get(dataRateParameterHandle),
                        theParameters.get(
                        userProtocolIDParameterHandle));
12          }
```

Trecho de código 4.17 – Função *receiveInteraction()*: enviando a mensagem do federado para uma entidade no VR-Forces

Concluindo a função *receiveInteraction()*, o código 4.18 trata as exceções e os erros que podem haver ao longo deste método. O primeiro *catch* reporta erros sobre o estado do

federado na simulação, o segundo diz respeito às chamadas dos métodos *decode* mostrados no trecho 4.15 e o terceiro reporta erros concernentes à RTI.

```
                    catch (FederateNotExecutionMember |
                       NotConnected | RestoreInProgress |
                       SaveInProgress e) {
              e.printStackTrace();
            }
          }

        } catch (DecoderException e) {
          e.printStackTrace();
        }

      } catch (RTIinternalError e) {
        e.printStackTrace();
      }
    }
  }
```

Trecho de código 4.18 – Função *receiveInteraction()*: Tratamento de exceções e erros do método

A função mostrada no trecho 4.19 é a responsável por mandar a mensagem do federado para a entidade no VR-Forces. Como todas as operações envolvendo RTI, esta também é feita pelo embaixador, aqui utilizando o método *sendInteraction*, o qual envia como argumento o *handle* da classe correspondente à interação desejada e um *map* com todos os *handles* dos parâmetros dessa classe.

```
    public void sendRadioMessage(byte[] sDL, byte[] tdlMC, byte[]
       hRI, byte[] sD, byte[] tDLT, byte[] dR, byte[] uPId) throws
        FederateNotExecutionMember,
  NotConnected, RestoreInProgress, SaveInProgress,
     RTIinternalError {
    try {
          ParameterHandleValueMap theParameters = ambassador.
             getParameterHandleValueMapFactory().create(7);
          theParameters.put(hostRadioIndexParameterHandle, hRI);
          theParameters.put(dataRateParameterHandle, dR);
          theParameters.put(signalDataLengthParameterHandle, sDL)
             ;
          theParameters.put(signalDataParameterHandle, sD);
          theParameters.put(tacticalDataLinkTypeParameterHandle,
```

```
                tDLT);
10              theParameters.put(tdlMessageCountParameterHandle, tdlMC
                );
11              theParameters.put(userProtocolIDParameterHandle, uPId);
12              ambassador.sendInteraction(
                applicationSpecificRadioSignalInteractionClassHandle
                , theParameters, null);
13          } catch (InteractionClassNotDefined e) {
14              throw new RTIinternalError("HlaInterfaceFailure", e);
15              } catch (InteractionClassNotPublished e) {
16        // TODO Auto-generated catch block
17        e.printStackTrace();
18      } catch (InteractionParameterNotDefined e) {
19        // TODO Auto-generated catch block
20        e.printStackTrace();
21      }
22    }
```

Trecho de código 4.19 – Função *sendRadioMessage():* Envia para o RTI a classe applicationSpecificRadioSignalInteraction com os respectivos atributos (parâmetros)

Como a mensagem, no VR-Forces, foi mandada de uma entidade específica para outra entidade específica, e esses dados estão encriptados dentro do parâmetro *signalData* e, portanto, não foram modificados nesta operação, a entidade alvo das mensagens de rádio VR-Forces também será a entidade alvo da mensagem do federado.

A figura 4.22 mostra o satélite *SWISSCUBE* da simulação depois que as mensagens foram enviadas a ele, mostrando que as informações foram enviadas e recebidas a contento pelo protocolo HLA.

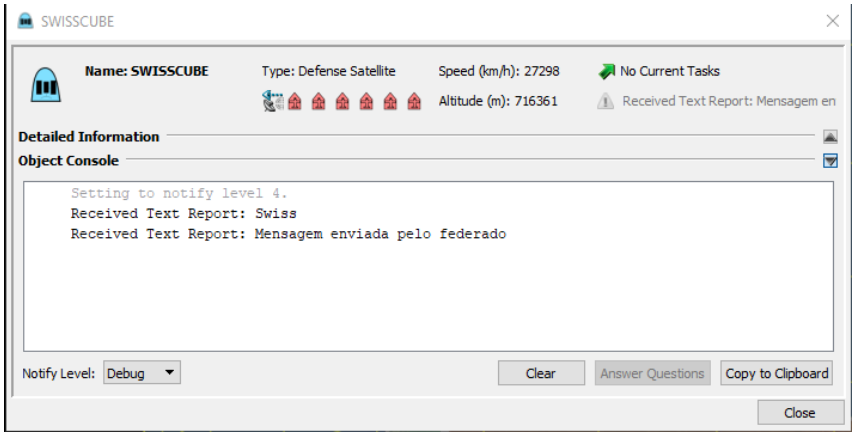


FIGURA 4.22 – Entidade do VR-Forces recebendo a mensagem de outro satélite e do Federado Aeronave RPR 2.0

5 Conclusão

Neste trabalho realizaram-se os primeiros passos para a viabilização da integração de simulações de modo a testar a robustez de uma arquitetura conceitual utilizando simulação distribuída. Foram realizados testes em um simulador de cenários, VR-Forces, o qual possui integração com RTI e, consequentemente, é possível escalar a aplicação para mais federados.

O trabalho foi realizado, por sua vez, utilizando um cenário base criado para que se pudesse testar os modos de conexão do HLA. Para tanto, foi adicionado à simulação um federado que interage com o VR-Forces com publicações e subscrições de objetos *(Objects)* e interações *(Interactions)*. A publicação do objeto cria uma entidade na simulação parecida com as unidades do VR-Forces, embora controlado externamente. A publicação de interações permite que haja troca de informações entre os dois programas.

Com este caso base, portanto, foi possível averiguar como funciona uma simulação distribuída utilizando HLA. O protocolo HLA mostra-se consistente para juntar diversos simuladores que foram feitos ou podem funcionar de modo independente, mas que possam fazer parte em um cenário mais complexo.

Algumas sugestões para aprimorar o trabalho seriam: fazer subscrições do tipo *Object* do VR-Forces para um federado, adicionar federados diferentes e realizar interações entre eles e testar uma simulação macro em rede, com uma máquina central executando o *rtiexec* e as demais conectadas a ela. A integração de simuladores numa simulação distribuída utilizando HLA seria uma grande contribuição para o melhor uso de simuladores dentro da Força Aérea, bem como em qualquer ambiente em que haja viabilidade de isso ser implementado.

Referências

AZEVEDO, D. N. R. **Benchmarking de resiliência para infraestruturas de simuladores de satélites baseadas em hla**. Doutorado em Engenharia e Tecnologia Espaciais — Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2014.

BRASIL. **Portaria nº1.814/MD, de 13 de junho de 2013**. Brasília: Ministério da Defesa. Secretaria-Geral do Exército, 2013.

CCA-SJ: Projeto de conferência de hla no simulador do mi-35 (a-h2). São José dos Campos, SP: CCA-SJ, 2009. Available at: http://svn.ccasj.intraer/60-AH2/trunk/hlaah2/. Accessed on: 30/08/2021.

CERTI. 2021. Available at: https://savannah.nongnu.org/projects/certi. Accessed on: 03/11/2021.

FUJIMOTO, R. M. Parallel and distributed simulation systems. *In*: _____. United States of America: John Wiley & Sons, Inc., 2000.

IDE Eclipse: o que é e sua importância para desenvolvedores. 2021. Available at: https://www.remessaonline.com.br/blog/ide-eclipse/. Accessed on: 13/10/2021.

LEES, M.; LORGAN, B.; KING, J. **The Architecture and Implementation of the BacGrid Simulator**. United States of America: ResearchGate, 2006. Available at: https://www.researchgate.net/publication-/265748597_The_Architecture_and_Implementation_of_the_BacGrid_Simulator. Accessed on: 25/06/2021.

MAK Technologies. 2021. Available at: //https://www.mak.com/. Accessed on: 03/11/2021.

MöLLER, B. **The HLA Tutorial - A practical guide for developing Distributed Simulations**. 1.0. ed. Swden, 2012. 101 p.

MöLLER, B.; DUBOIS, A.; LEYDOUR, P. le; VERHAGE, R. Rpr fom 2.0: A federation object model for defense simulations. p. 15, 2014. Available at: https://www.sisostds.org/DesktopModules/Bring2mind/DMX/API/Entries-/Download?Command=Core_DownloadEntryId=42390PortalId=0TabId=105. Accessed on: 29 set. 2021.

ORACLE - Class Thread documentation. 2021. Available at: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html. Accessed on: 19/10/2021.

PADRãO Factory. 2021. Available at: https://www.devmedia.com.br/como-usar-o-pattern-factory-na-plataforma-java-ee/32814. Accessed on: 22/10/2021.

PITCH Technologies. 2021. Available at: https://pitchtechnologies.com/. Accessed on: 03/11/2021.

PORTICO Project. 2021. Available at: https://porticoproject.org/. Accessed on: 03/11/2021.

SPACE-TRACK.ORG. 2021. Available at: https://www.space-track.org/. Accessed on: 05/10/2021.

TECHNOLOGIES, M. **Mak RTI Reference Manual**. 4.5. ed. United States of America, 2018. 291 p.

TECHNOLOGIES, M. **VR-Forces First Experience Guide**. 4.8. ed. United States of America, 2020. 92 p.

TECHNOLOGIES, M. **VR-Forces User's Guide**. 4.8. ed. United States of America, 2020. 185 p.

TECHNOLOGIES, P. **Pitch Visual OMT™**. 1.0. ed. United States of America, 2013. 48 p.

TOPÇU, O.; DURAK, U.; OĞUZTÜZÜN, H.; YILMAZ, L. Distributed simulation - a model driven engineering approach. *In*: _____. Cham, Switzerland: Springer, 2016.

TOPÇU, O.; OĞUZTÜZÜN, H. Guide to distributed simulation with hla. *In*: _____. Cham, Switzerland: Springer, 2017.

TWO-LINE Element. 2021. Available at: https://en.wikipedia.org/wiki/Two-line_element_set. Accessed on: 10/06/2021.

TWO-LINE Element Format. 2021. Available at: https://ai-solutions.com/_help_Files/two-line_element_set_file.htm. Accessed on: 10/06/2021.

VR-FORCES in Space. 2019. Available at: https://vimeo.com/313460119. Accessed on: 06/10/2021.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO Nº	4. Nº DE PÁGINAS
TC	24 de novembro de 2021	DCTA/ITA/TC-114/2021	57

5. TÍTULO E SUBTÍTULO:

Estruturação de uma Arquitetura de Simulação Distribuída para Sistemas Aeroespaciais

6. AUTOR(ES):

Pedro Elardenberg Sousa e Souza

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

Simulador, Simulações Aeroespaciais, Simulação Distribuída, HLA, RTI

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Sistemas aeroespaciais; Simuladores espaciais; Missões espaciais; Arquitetura (computadores); Engenharia aeroespacial

10. APRESENTAÇÃO: (**X**) **Nacional** ( ) **Internacional**

ITA, São José dos Campos. Curso de Graduação em Engenharia Aeroespacial. Orientador: Prof. Dr. Christopher Shneider Cerqueira; coorientador: Maj. Eng. Daniel Lelis Baggio. Publicado em 2021.

11. RESUMO:

Missões espaciais, devido a sua alta precisão requerida em parâmetros orbitais e de lançamento, bem como seu alto custo operacional, requerem simulações para que sejam feitas verificações, validações e que sejam previstas algumas rotinas de operações. Tais missões, no entanto, possuem agentes que se comunicam com agentes de outras missões, de modo que uma única simulação não abarca toda a complexidade e escalabilidade de um ambiente complexo. Neste trabalho, foi proposta a utilização de um *software* de simulação de cenários no qual foi utiizada uma infraestrutura baseada em *High Level Architecture* (HLA) de modo a integrar sistemas de simulação de aplicações aeroespaciais.

Para isso, foi desenvolvido um *software* de simulação que se comunica com o primeiro, enviando e recebendo sinais de veículos espaciais representados no cenário, de modo a demonstrar a viabilidade desta arquitetura de simulação distribuída.

12. GRAU DE SIGILO:

(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **SECRETO**