GIOVANI AMIANTI

ARQUITETURA DE SOFTWARE AVIÔNICO DE UM VANT COM REQUISITOS DE HOMOLOGAÇÃO.

SÃO PAULO

2008

GIOVANI AMIANTI

ARQUITETURA DE SOFTWARE AVIÔNICO DE UM VANT COM REQUISITOS DE HOMOLOGAÇÃO.

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

SÃO PAULO

2008

GIOVANI AMIANTI

ARQUITETURA DE SOFTWARE AVIÔNICO DE UM VANT COM REQUISITOS DE HOMOLOGAÇÃO.

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

Área de Concentração: Controle e Automação

Orientador: Prof. Dr. Ettore Apolônio de Barros

SÃO PAULO

2008

DEDICATÓRIA

Dedico este trabalho à XMobots.

AGRADECIMENTOS

Primeiramente a minha mãe (Márcia Regina Miniello Amianti) que sempre me incentivou a todas as minhas aspirações profissionais, incondicionalmente.

A meu pai (Cláudio Antônio Amianti) que sempre foi uma referência nos momentos de inspiração.

A minha irmã (Gislaine Amianti) que nos diversos momentos de dificuldade sempre estava ao meu lado para me ouvir e acolher.

A minha namorada (Daiana Vianna) que sempre me trazia de volta a realidade tornando a vida mais confortável.

Ao meu orientador (Prof. Dr. Ettore Apolônio de Barros) que sempre me mostrou as oportunidades no mercado de veículos não tripulados e que me deu a oportunidade de desbravar este nicho.

À XMobots, empresa de veículo não tripulados desenvolvedora do VANT Apoena I, que deu grande suporte a esta pesquisa.

Aos amigos da Xmobots: Fábio Henrique de Assis, João Lucas Dozzi Dantas, Elcio Ricardo Lora e Cleber Miniello Roma que sempre estiveram ao meu lado na construção de um ideal.

Ao amigo Fábio Zanoni pela parceria no desenvolvimento do filtro de kalman estendido.

Ao amigo (Prof. Dr. Antônio Luiz de Campos Mariani) que deste os tempos de iniciação científica me iluminou com conselhos valiosos.

EPÍGRAFE

As pequenas oportunidades

são frequentemente o início

de grandes empreendimentos.

Demóstenes

RESUMO

Recentemente, um crescente número de institutos de pesquisa pelo mundo tem focado seus estudos em veículos aéreos não tripulados (VANT ou, em inglês, UAV - *Unmanned Aerial Vehicle*), que se revelam muito úteis tanto em aplicações militares quanto civis, pois suas principais vantagens são: a alta confiabilidade, baixo risco à vida, reduzido custo de implantação e manutenção. A pesquisa apresentada neste trabalho integra-se ao projeto BR-UAV em desenvolvimento na empresa Xmobots Sistemas Robóticos LTDA e no Laboratório de Veículos Não Tripulados (LVNT) da Escola Politécnica da USP. O projeto BR-UAV visa a contribuir para a inserção desta tecnologia no país e, para tanto, desenvolve atualmente a plataforma, aviônica e sistema de controle autônomo voltados ao objetivo de monitoramento no espectro visível e infravermelho. O principal requisito do projeto BR-UAV é o desenvolvimento de um sistema aéreo não tripulado capaz de voar dentro do espaço aéreo controlado e para tanto este deve ser homologado.

Esta pesquisa foca no desenvolvimento do software embarcado, assim este software deve ser desenvolvido de acordo com uma metodologia direcionada a homologação. Por isso, este trabalho propõe uma metodologia que foi baseada em cinco elementos: processo de desenvolvimento, normas, ferramentas de sistema operacional, ferramentas de aplicação e ferramentas matemáticas.

Após o estabelecimento dos objetivos, de uma análise do estado da arte em sistemas aviônicos, e da metodologia de certificação, o processo de desenvolvimento foi inicializado. Na fase de engenharia de sistemas, os requisitos de sistema foram capturados. Então a arquitetura de sistema (hardware e software) foi modelada e analisada. A partir desta modelagem de sistema, os requisitos funcionais e temporais de software puderam ser capturados na etapa de análise da fase de engenharia de software. Na etapa de projeto, tanto arquiteturas de software reativo-deliberativas quanto seguro-confiáveis foram analisadas e então foi apresentado o desenvolvimento de uma arquitetura híbrida baseada em agentes batizada de GESAM (*Generic and Extensible Software Architecture for Mobots*). A GESAM foi projetada para atingir principalmente generalidade e extensibilidade bem como segurança, confiabilidade, deliberação e reatividade. Nesta etapa, somente foi modelada a estrutura de relacionamento e de comportamento dos agentes. Os principais componentes da arquitetura foram representados pelos seguintes agentes: **Manager**, **Observer**, **Actuator**, **Communicator**, **Autopilot**, e **Payload**. Na etapa de Implementação, o interior dos agentes foi codificado. Além disso, foi implementado o filtro de Kalman estendido para integrar informações de GPS, unidade de medição inercial e bússola. Na etapa de Testes, foram realizados testes de integração funcional e de desempenho computacional. Os resultados demonstraram que o sistema atendeu a todos os requisitos consumindo 38.3% de processamento.

Finalmente, os próximos passos desta pesquisa são discutidos.

ABSTRACT

Recently, an increasing number of research institutes around the world has been focusing their efforts in the study of unmanned aerial vehicles (UAV), which have proved to be very useful both in military and civil applications because of their major advantages: high reliability, reduced risk to life, reduced maintenance and implantation costs. The research presented in this work is part of the BR-UAV project, which is in development at *XMobots Sistemas Robóticos LTDA* and at the *Laboratório de Veículos Não Tripulados* of USP (Brazil). This project aims to contribute for the insertion of this technology in Brazil. Particularly, at the present stage, the project includes the development of the platform, avionics and autonomous control system for environment monitoring via visible and infrared spectrums. The main requirement of BR-UAV Project is the development of an unmanned aerial system that could flight in controlled airspace then this system has to be certificated.

This research is focused on the development of embedded software, and therefore this software should be developed according to a certification methodology. For this purpose, this work proposes a methodology that was based into five guidelines: development process, norms, operating system tools, application tools and mathematical tools.

The development process was started after the statement of objectives and the analysis of the state of art on UAV avionics. In the systems engineering phase, system requirements were captured and then the system architecture (hardware and software) was modeled and analyzed. From the system modeling, the functional and temporal software requirements could be captured in the analysis stage of the software engineering phase. In the Design stage, reactive-deliberative and safety-reliability software architectures were analyzed, and then it was presented the development of a hybrid architecture based on agents called GESAM (Generic and Extensible Software Architecture for Mobots). The GESAM was designed for achieving mainly generality and extensibility, as well as safety, reliability, deliberation and reactivity. At this stage, only the relationship structure and behavior of agents were modeled. The main components of the architecture are the agents: **Manager**, **Observer**, **Actuator**, **Communicator**, **Autopilot**, and **Payload**. In the Implementation stage, the agents were coded as well as the Extended Kalman Filter for integrating information from GPS, inertial measurement unit and Compass sensors. In the Tests stage, integration tests were performed. The results showed that the system could fulfill requirements using 38.3% of processing consumption.

Finally, the next steps of this research are discussed.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EQUAÇÕES

LISTA DE ABREVIATURAS

AD	Conversão analógical em digital
ADB	Air Data Boom
APSIG	Agent Platform Special Interest Group
ASIC	Application Specific Integrated Circuit
ATCS	Air Traffic Control System
AUML	Agent Unified Modeling Language
AUV	Autonomous Underwater Vehicle
BSP	Board Support Package
CBA	Circuit Board Assemblies
CDT	C Development Tool
CM	Configuration Management
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
CPLD	Complex Programmable Logic Device
CPS	Compass
CPU	Central Processing Unit
CUI	Control User Interface
DCD	Dual Channel Design
DD	Conversão Digital-Digital
DER	Designated Engineering Representatives
DFT	Design For Testability
DRP	Diverse Redundancy Pattern
ECU	Electronic Control Unit
ED	Proprioceptive Energy Data
EKF	Extended Kalman Filter
FAA	Federal Aviation Administration
FAR	Federal Aviation Regulation (EUA)
FIPA	Foundation for Intelligent Physical Agents
FPGA	Field Programmable Gate Array
GESAM	Generic and Extensible Software Architecture for Mobots
GPS	Global Positioning System
GUI	Graphical User Interface
HILS	Hardware-In-the-Loop Simulation
HRP	Homogeneous Redundancy Pattern
IDE	Integrated Development Environment
IFR	Instrumented Flight Rules
IMMC	Integrated Mission Management Computer
IMU	Inertial Measurement Unit

JAA	Joint Aviation Authorities
JAR	Joint Aviation Requirements
LRU	Line Replaceable Unit
MAP	Monitor Actuator Pattern
MCVP	Multi Channel Voting Pattern
MEMS	Micro-Electro-Mechanical Systems
MDA	Model Driven Architecture
MDD	Model Driven Development
MTBF	Mean Time Between Failure
NED	North East Down
OCP	Open Control Platform
OMDs	Object Model Diagrams
OMG	Object management Group
PLD	Program Logic Device
POSIX	Portable Operating System Interface in Unix
PSAC	Plan for Software Aspects of Certification
QA	Quality Assurance
RDD	Requirements Driven Development
ROOM	Rapid Oriented Object Modeling
RTCA	Radio Technical Commission for Aeronautics, Inc.
RTOS	Real Time Operating System
RTS	Real Time Systems
SANT	Sistema Aéreo Não Tripulado
SCMP	Software Configuration Management Plan
SDP	Software Development Plan
SOI	Stage Of Involvement
SVP	Software Verification Plan
SysML	System Modeling Language
SCD	Statechart Diagram
SCPD	Single Channel Protected Design
SD	Sequence Diagram
SEP	Safety Executive Pattern
SLAM	Simultaneous localization and mapping
SQAP	Software Quality Assurance Plan
TD	Proprioceptive Thrust Data
UAV	Unmanned Aerial Vehicle
UCD	Use Case Diagram
UGV	Unmanned Ground Vehicle
UML	Unified Modeling Language
UUV	Unmanned Underwater Vehicle

VANT	Veículo Aéreo Não Tripulado
VFR	Visual Flight Rules
WMT	Weighed Mean by Time

LISTA DE SÍMBOLOS

A	Matriz de transição de estados
B	Matriz de entradas de controle
C	Matriz de ruídos de processo
H	Matriz de leitura dos sensores
G	Matriz de ruídos de medição
x	Vetor de estados
z	Vetor de medições do processo
u	Vetor entrada de controle
w	Ruídos brancos de media nula do processo
v	Ruídos brancos de media nula dos sensores
Q	Covariância do ruído de dinâmica
R	Covariância do ruído de medição
P	Matriz covariância dos estados
K	Matriz de ganho de kalman
T	Intervalo de tempo no espaço discreto
λ	Latitude
ϕ	Longitude
h	Altitude
$\begin{bmatrix} V_N & V_E & V_D \end{bmatrix}$	Velocidade em NED
$\begin{bmatrix} a & b & c & d \end{bmatrix}$	Parâmetros do quaternion
g	Aceleração da gravidade
$\begin{bmatrix} b_{a_x} & b_{a_y} & b_{a_z} \end{bmatrix}$	Erros de *bias* do acelerômetro no sistema de coordenadas da plataforma
$\begin{bmatrix} b_{a_N} & b_{a_E} & b_{a_D} \end{bmatrix}$	Erros de *bias* do acelerômetro no sistema de coordenadas NED
$\begin{bmatrix} b_{g_x} & b_{g_y} & b_{g_z} \end{bmatrix}$	Erros de *bias* do giroscópio no sistema de coordenadas da plataforma
a , $\begin{bmatrix} a_x & a_y & a_z \end{bmatrix}$	Aceleração no sistema de coordenadas da plataforma
$\begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}$	Velocidades angulares no sistema de coordenadas da plataforma
R_λ	Raio da terra na direção da latitude
R_ϕ	Raio da terra na direção da longitude
Ω	Velocidade angular de rotação da Terra
C_p^n	Matriz mudança de base das coordenada da plataforma para a coordenada no sistema NED
δ_x	Erro randômico do sensor, onde índice “x“ indica de qual estado a quem o erro pertence.

SUMÁRIO

1 INTRODUÇÃO

Atualmente, fica evidente que o foco dos investimentos em projetos de pesquisa envolvendo VANTs está na área militar. Segundo OSD (2005) cerca de 3 Bilhões de dólares são investidos anualmente, sendo estes concentrados nos seguintes países: Estados Unidos, França, Alemanha, Reino Unido, Itália, Israel e Rússia. Todos, exceto Israel, são países membros da MTCR (*Missile Technology Control Regime*) que é um acordo informal e voluntário de 33 países, incluindo o Brasil, para controlar a proliferação de sistemas aerodinâmicos capazes de transportar armas de destruição em massa.

No entanto, a tendência é que estes sistemas tornem-se acessíveis a aplicações civis cujas principais vantagens são:

- Alta confiabilidade;
- Baixo risco à vida;
- Reduzido custo de implantação;
- Reduzido custo de manutenção;

Diversas aplicações civis de VANT estão começando a se difundir ao redor do mundo, tais como: vigilância policial de áreas urbanas, vigilância de áreas de fronteira, inspeção de oleodutos e gasodutos, controle de safras agrícolas, levantamento de recursos naturais, controle de queimadas, enlace de comunicações e cobertura de eventos para as redes de TV. Para citar dois casos de sucesso em projetos de VANTs para uso civil, podem ser citadas a empresa americana Aerosonde (www.aerosonde.com) e a empresa canadense Fugro Airborne (www.fugroairborne.com).

Neste contexto de aplicações civis, esta pesquisa se insere com contribuições nas seguintes áreas de aviônicos de VANT: metodologia de desenvolvimento de sistemas com requisitos de homologação; estabelecimento de requisitos; modelagem; implementação e testes de hardware e software.

1.1 Os VANTs no Mundo

Atualmente, 32 países estão desenvolvendo ou manufaturando mais de 250 modelos de VANT, e 41 países operam cerca de 80 modelos, principalmente para reconhecimento (OSD, 2005). Segundo OSD (2005), os gastos do Departamento de Defesa dos EUA (DoD) gastos com o desenvolvimento destes veículos, entre 1988 e 2001, eram em média US$250 milhões anuais. Entre 2002 e 2005 houve um salto para US$1,5 bilhões anuais e prevê-se que investimentos do DoD devem totalizar entre US$2 e US$3 bilhões anuais entre 2005 e 2010.

1.2 Os VANTs no Brasil

Assim como no cenário internacional, as primeiras utilizações de VANTs nasceram nas atividades militares. Houve no início da década de 80 o desenvolvimento do projeto Acauã, que visava à confecção de um alvo aéreo manobrável para auxílio no projeto do míssil Piranha. O projeto foi paralisado em 1988 por falta de recursos orçamentários.

Atualmente, existem no Brasil diversos projetos de VANT com finalidades distintas (VASCONCELLOS, 2003, 2006; OLIVEIRA, 2005), alguns dos quais em fase conceitual, enquanto outros estão prontos para comercialização. Contabiliza-se cerca de quatorze iniciativas direcionadas à utilização de VANTs, tanto na aviação civil como militar. Destas quatorze iniciativas, oito são comercias correspondendo às empresas Xmobots, Gyrofly, AGX, Embravant, Fitec, PrinceAirModel, Aeromot e Flight Solutions e seis são institucionais correspondendo à USP de São Paulo, USP de São Carlos, ITA, UnB, CTA e Cenpra.

No que se refere à estruturação de uma política nacional para o setor, o Ministério da Defesa do Brasil estabeleceu suas diretrizes através da Portaria Nº 606/MD de 11/06/2004, publicada no DOU Nº 112 em 14/06/2004 formando uma “Comissão de Coordenação Nacional do Programa VANT”. Quanto a regulamentações, foi redigida uma proposta de lei denominada RBHA-100 (2004) que estabeleceu algumas normas de operação para que os VANTs sejam integrados harmonicamente em nosso espaço aéreo, mas que não foi aprovada e encontra-se atualmente em discussão. Para operar um VANT no Brasil, atualmente, é preciso solicitar autorizações especiais da ANAC e, dependendo das altitudes e locais de operação, é preciso solicitar autorizações especiais do Controle de Tráfego Aéreo.

1.3 Motivação

A aplicação militar como visto, tem sido extensivamente explorada, principalmente por agências norte-americanas e israelitas. No entanto, aplicação civil desta tecnologia desperta o maior interesse por parte dos pesquisadores brasileiros, pois almejam que o mercado nacional abra diversas portas para a utilização de VANTs, mais especificamente, aplicações de monitoramento rural, preditivo, ambiental e urbano.

1.3.1 Monitoramento Ambiental

O monitoramento ambiental foca a prospecção de recursos naturais (identificação de extratos e tipologias florestais, petróleo, ferro, diamante, arqueologia e busca de cardumes), mapeamento meteorológico (camada limite, psicrometria, composição e qualidade atmosférica), monitoramento de fauna e flora (florestas, mangues, rios, lagos, mananciais de abastecimento, bem como detecção e suporte à extinção de incêndios).

Com a aplicação de VANTs no setor de aerofotogrametria, por exemplo, pode-se reduzir o custo operacional da captação de imagens em cerca de 30%, segundo a empresa

Xmobots Sistemas Robóticos LTDA., reduzindo-se a periculosidade da operação e mantendo-se a qualidade das informações obtidas, conforme observa-se na Figura 1-1, que retrata o resultado de um aerolevantamento não-tripulado de uma região de prospecção de ferrita.

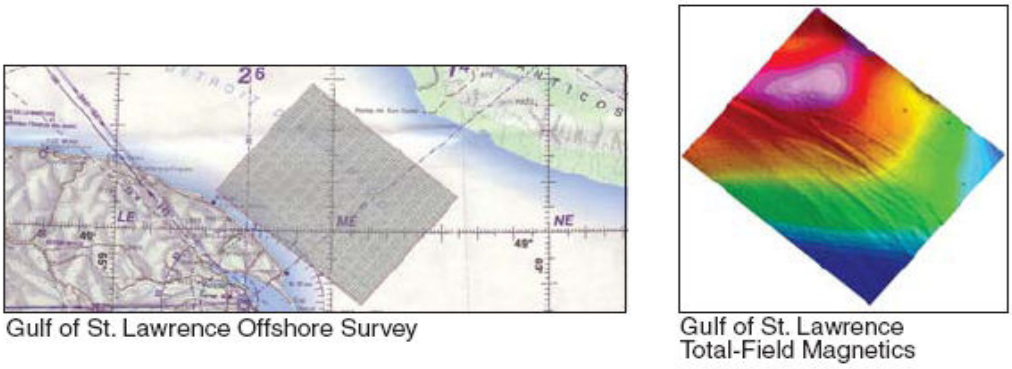

Figura 1-1: Prospecção de ferrita (Fonte: Fugro Airborne).

1.3.2 Monitoramento Preditivo

O monitoramento preditivo foca inspeções estruturais principalmente linhas da transmissão de energia elétrica, como também, represas, aquedutos, oleodutos, gasodutos, rios, ferrovias, rodovias, pontes e telecomunicação.

Entre os países industrializados, o Brasil é um dos mais dependentes da hidroeletricidade, com 96,8% da energia produzida por cerca de 600 barragens (ANEEL, 2006). O grande desenvolvimento da hidroeletricidade no Brasil foi entre 1975, quando a capacidade instalada era apenas de 18.500 GW, e 1985, quando passou para 54.000 GW. Atualmente estão projetados pelo governo 140 novos investimentos em hidrelétricas como parte da ampliação da capacidade nacional de geração. Todavia, a hidroeletricidade provavelmente contribuirá com menor porcentagem no quadro energético, pois, dados oficiais estimam que, por volta de 2008, as hidrelétricas proverão 81% da energia nacional devido à inserção da geração termoelétrica. Embora se observe uma tendência na redução da dependência da hidroeletricidade não se pode desprezar a extrema importância que a mesma mantém no setor energético brasileiro e como conseqüência disso, a interligação entre os centros geradores e os centros consumidores é viabilizada por um extenso sistema de transmissão, pois, as usinas hidrelétricas são caracterizadas por poucas instalações de alta capacidade geradora distantes dos centros de consumo. Se for considerado além das dimensões físicas, o ambiente hostil e a dificuldade de acesso para monitoramento e manutenção, fica evidente que as linhas de transmissão é o elemento do sistema elétrico de potência mais susceptível a falhas, como se pode verificar na Figura 1-2.

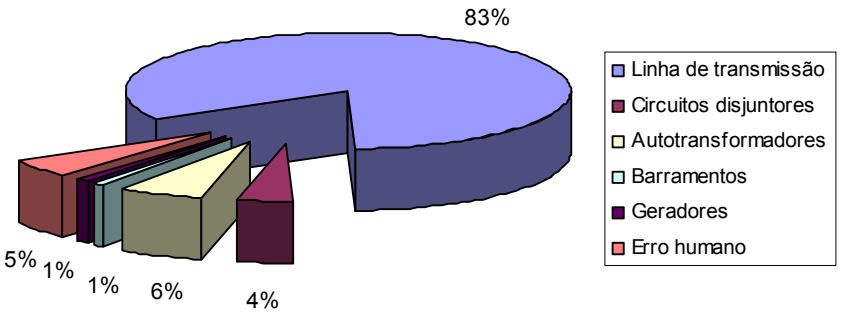

Figura 1-2: Distribuição de falhas em sistema de transmissão de 500kV do Brasil (ANEEL, 2006).

Como os sistemas elétricos de potência devem garantir um alto grau de confiabilidade na continuidade do fornecimento de energia elétrica, o monitoramento e a manutenção preditiva revelam-se imprescindíveis. Atualmente, o monitoramento é realizado por helicópteros com câmeras que provêm imagens tanto no espectro visível (Figura 1-3), para controle de vegetação e de invasão, quanto no espectro do infravermelho (Figura 1-4), para a detecção de rupturas eminentes a partir do efeito Joule evidenciado pelo gradiente de temperatura. No entanto, este serviço revela elevado custo de implantação, operação e manutenção, pois envolve a utilização de helicópteros tripulados com piloto e operadores, o que onera o seguro devido o alto risco à vida nas operações.

Figura 1-3: Monitoramento no espectro visual e infravermelho de linhas de transmissão (Fonte: FLIR Systems).

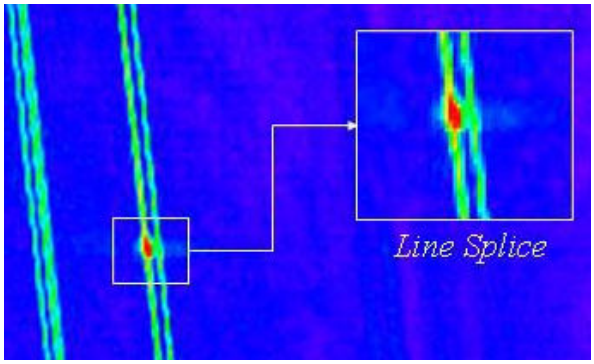

Figura 1-4: Detalhe de falha em linha de transmissão no monitoramento no espectro infravermelho (Fonte: FLIR Systems).

Dentro deste contexto, a utilização de um veículo aéreo autônomo surge como uma alternativa atraente, capaz de reduzir substancialmente os altos custos associados ao monitoramento.

1.3.3 Monitoramento Urbano

O monitoramento urbano engloba: o mapeamento urbano (planejamento, aerofotogrametria/cartografia/mapas), a segurança pública e apoio tático (polícia federal, militar e civil, corpo de bombeiros, companhias de tráfego), segurança privada (seguradoras, escolta de frota e valores), coberturas jornalísticas (eventos e trânsito) e turismo virtual.

Nas Figura 1-5 e Figura 1-6, podem-se observar aplicações de VANT na segurança pública e apoio tático à polícia nos EUA e à polícia na França, respectivamente.

Figura 1-5: Monitoramento infravermelho de portos (Fonte: FLIR Systems).

Figura 1-6: Monitoramento no espectro visível urbano (Fonte: FLIR Systems).

1.3.4 Monitoramento Rural

O monitoramento rural foca principalmente a agricultura de precisão (planejamento do manejo, controle do plantio, de invasoras, da fertilização do solo e da irrigação, bem como predição de rendimento) e em segundo plano a pecuária (observação e contagem de rebanhos), a topografia (curvas de nível, caracterização de solos) e divisas (controle de invasões);

A modernização do agro negócio brasileiro evidenciado nas duas últimas décadas revela que novas tecnologias representam um insumo cada vez mais importante para o setor, tanto do ponto de vista da produção como da gestão, exigindo assim o suporte de ponta para superação dos desafios existentes e, conseqüentemente, tornando a aplicação de VANTs no agro negócio um importante campo do conhecimento para elevar a agricultura de precisão a um novo patamar e assegurar a competitividade do setor no Brasil fronte ao mercado externo.

A utilização de VANT na agricultura viabiliza o mapeamento de plantações por meio de fotos tanto no espectro visível quanto no infravermelho. A partir do mosaico de fotos georeferenciadas é possível tratar as imagens (Figura 1-7) e detectar a partir das fotos geradas na câmera de espectro visível: solo nu devido às falhas na distribuição de sementes durante plantação, má formação devido a ataque de invasoras ou insetos, hipertrofia local devido à escassez de nutrientes ou fertilizantes; e a partir das fotos geradas pela câmera de espectro infravermelho: uniformidade na distribuição da irrigação.

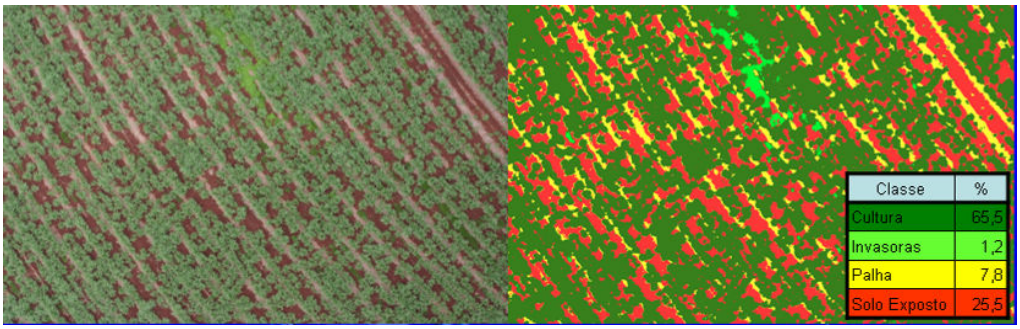

Figura 1-7: Tratamento de imagem no espectro visível (Fonte: Embrapa).

1.4 Objetivos

Como foi verificado, VANT é um tema novo, de expansão inevitável e de clara importância para o Brasil. Neste contexto, revela-se imprescindível **propor uma metodologia de projeto do sistema aviônico de VANTs** para orientar futuros engenheiros. Esta metodologia deve englobar as respectivas normas de homologação (para permitir a operação de VANTs em espaço aéreo controlado), o processo de desenvolvimento (incluindo definição de requisitos do sistema, definição dos requisitos de software, projeto da arquitetura, implementação e testes) bem como, a seleção de ferramentas de desenvolvimento adequadas.

Para exemplificar a metodologia de projeto de sistema aviônico proposta, deve-se **projetar, implementar e testar uma arquitetura de software aviônico de VANT** capaz de aquisitar os sensores, fundir as informações sensoriais e comunicar-se com a estação base enviando os dados de navegação da aeronave. Esta arquitetura de software deve oferecer uma estrutura para futuras implementações de observação (inserção de novos sensores e novos algoritmos de fusão sensorial), de comunicação (inserção de novos meios de comunicação com a estação base como SatCom e GPRS bem como inserção de novos meios de comunicação direcionados à coordenação entre veículos e de tráfego aéreo), de atuação

(inserção de novos atuadores) e de piloto automático (controle, guiamento, geração de trajetória e sistema de anti-colisão). Além disso, esta arquitetura deve gozar principalmente de **extensibilidade** (capacidade de adaptar-se facilmente à inclusão de novos módulos) e **generalidade** (os módulos devem ser suficientemente genéricos para que possam ser intercambiados entre diferentes veículos). Estas características visam possibilitar a adaptação desta arquitetura em outros veículos, porque se observou, a partir de interações com projetos de USV (*Unmanned Surface Vehicles*) (FERREIRA, 2003) e AUV (*Autonomous Underwater Vehicles*) realizados no LVNT (Laboratório de Veículos Não Tripulados), que os respectivos sistemas embarcados revelam-se muito semelhantes ao sistema aviônico de um VANT.

Neste contexto os objetivos podem ser assim resumidos:

- **Propor uma metodologia de projeto do sistema aviônico de VANTs:** deve-se definir um processo de desenvolvimento, as normas de homologação e as ferramentas de projeto.
- **Projetar uma arquitetura de software aviônico de VANT:** projeto de arquitetura genérica e extensível com estrutura para futuras implementações de gerenciamento de missão, planejamento de trajetória/anti-colisão, coordenação, guiamento, controle, navegação, sensoriamento, atuação, comunicação e controle de payload.
- **Implementar e testar parte da arquitetura proposta:** aquisitar sensores IMU, GPS e CPS, fundir informações sensoriais e enviar estados da aeronave para a estação base.

1.5 Organização do trabalho

Esta dissertação apresenta, no capítulo 2, uma visão geral sobre o estado da arte em sistemas aviônicos de VANTs. Uma análise estatística de centenas de trabalhos revela a carência de estudos na área. Destes trabalhos, os principais foram filtrados e sintetizados no escopo das principais características aviônicas, como: a arquitetura de hardware, sistema operacional, linguagem de programação, arquitetura de software, modelo do software e análise de requisitos de tempo real.

No capítulo 3, é apresentado o projeto BR-UAV, no qual se integra esta pesquisa. Detalha-se a missão do VANT Apoena I, as características da plataforma (parte mecânica do VANT) e da aviônica (hardware e software tanto embarcado quanto da estação base). Mais especificamente, na aviônica são apresentados a arquitetura de hardware e os sistemas que a compõem: sistema de energia, sistema de observação, sistema de comunicação, sistema de atuação e sistema de processamento.

No capítulo 4, é proposta uma metodologia de projeto de sistema aviônico de VANTs. Esta metodologia é balizada por normas de homologação, processo de desenvolvimento de sistemas críticos (incluindo definição de requisitos do sistema, definição dos requisitos de

software, projeto da arquitetura, implementação e testes) bem como por ferramentas de desenvolvimento sistema operacional de tempo real, *drivers*, pacotes de suporte de placas, aplicação e cálculo.

No capítulo 5, é realizada a análise dos requisitos do software aviônico embarcado. O procedimento de análise consiste em levantar os requisitos funcionais e temporais do sistema aviônico (hardware e software), modelar a arquitetura do sistema aviônico por meio da especificação SysML englobando modelos de blocos e comportamentos de hardware e software embarcado e da estação base, para finalmente, filtrar os requisitos funcionais e temporais do software aviônico.

No capítulo 6, é projetada a arquitetura de software aviônico embarcado. São apresentadas arquiteturas deliberativas e reativas bem como arquiteturas seguras e confiáveis que regem o projeto de uma proposta de arquitetura genérica e extensível batizada de GESAM. Em seguida é implementada a estrutura de relacionamento entre os diversos módulos de software estabelecidos pela GESAM correspondendo à realização do software aviônico embarcado.

No capítulo 7, são implementados todos os detalhes do software aviônico embarcado dando vida aos módulos definidos no projeto da arquitetura. Além disso, para que todos os módulos do sistema consigam se comunicar eficientemente, estabeleceu-se um padrão de estruturação de dados e para que o sistema não seja dependente de linguagem e sistema operacional, estabeleceram-se padrões de abstração de linguagem e de sistema operacional.

No capítulo 8, é apresentado o plano de testes bem como o resultado dos testes de integração na configuração de operação remota com observações da média ponderada pelo tempo e na configuração de operação remota com observações do filtro de Kalman estendido.

Finalmente, no capítulo 9, são apresentadas as conclusões englobando considerações técnicas, propostas de trabalhos futuros e considerações finais.

2 ESTADO DA ARTE EM SISTEMAS AVIÔNICOS DE VANTS

No contexto atual, conforme será observado neste capítulo, evidencia-se uma carência de estudos na área de sistemas aviônicos de VANTs com requisitos de homologação ou, de parte da homologação que é mais complexa, os sistemas aviônicos com requisitos de tempo real crítico.

É comum encontrar na literatura a expressão tempo real na qual o adjetivo “real” refere-se a uma oposição ao tempo simulado. Isto gera algumas confusões no momento de diferenciar sistemas de tempo real de outros sistemas computadorizados. Neste contexto, uma definição mais concisa e sem ambigüidades revela-se imprescindível. A característica que melhor diferencia sistemas de tempo real de outros tipos de sistemas é a capacidade de fornecer resultados não somente corretos (*correctness*), mas também dentro de prazos de tempo (*deadlines*) mais ou menos rígidos impostos pelo ambiente. Esta característica é conhecida como resposta temporal (*timeliness*) (LAPLANTE, 2004).

Um outro exemplo de definição encontra-se em Halang (1992) que cita a norma de padronização alemã DIN (DIN 44300 A2) para a caracterização destes sistemas. Segundo esta norma, uma operação em tempo real é: “o modo de operação de um sistema de computador em que os programas para processamento de dados que chegam do ambiente externos estão permanentemente prontos, de modo que seus resultados estejam disponíveis dentro de períodos de tempo predeterminados; o tempo da chegada dos dados podem ser aleatoriamente distribuídos ou determinados a priori em função das diferentes aplicações.”

Seguindo esta definição de tempo real filtraram-se cerca 600 trabalhos que declaram que suas implementações (aquisição de sensores, fusão sensorial, controle, guiamento, planejamento, comunicação, coordenação e SLAM) foram realizadas em tempo real, entretanto somente 5,5% (33 trabalhos como pode ser visto na Figura 2-1 denominados de *Proved Real Time* e *OCP/CORBA*) dão algum detalhe mínimo inerente a um sistema de tempo real, como: a arquitetura de hardware, sistema operacional, linguagem de programação, arquitetura de software, modelo do software e análise de requisitos de tempo real.

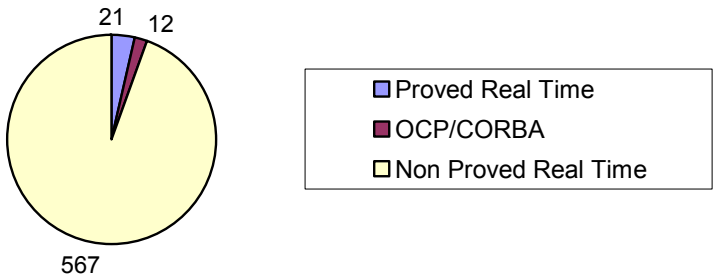

Figura 2-1: Distribuição de trabalhos com informações relativas a tempo real.

Destes 600 trabalhos, cerca de 2,0% (12 trabalhos) utilizam OCP/CORBA que representa a filosofia de computação distribuída não adotada neste trabalho e por este motivo foram desconsiderados.

Somente 3,5% (21 trabalhos - HONG, 2005; SUKKARIEH,2003; KIM,2004; KIM,2007; BRYSON,2007; ELFES, 2002; BUENO, 2003; JANG, 2006; TISDALE, 2006; HALL, 2001; HALL, 2002; EVANS, 2001; KIM, 2002; KIM, 2003; ATKINS, 1998; VIDOLOV, 2005; KUMON, 2006; HALLBERG, 1999; LOEGERING, 1999; CAMPBELL, 2004; FABIANI, 2006) dão detalhes da arquitetura de hardware utilizada (Figura 2-2) sendo centralizada, distribuída ou híbrida bem como da arquitetura de processamento utilizada (Figura 2-3) sendo PC104/x86, XScale, microcontrolado e outro.

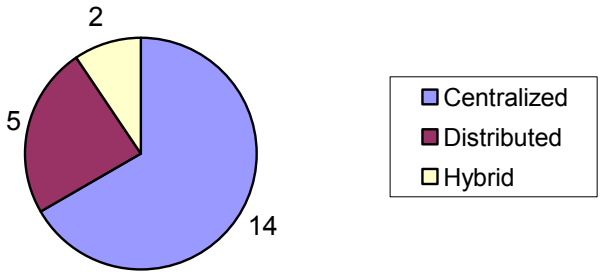

Figura 2-2: Distribuição das arquiteturas de hardware utilizadas.

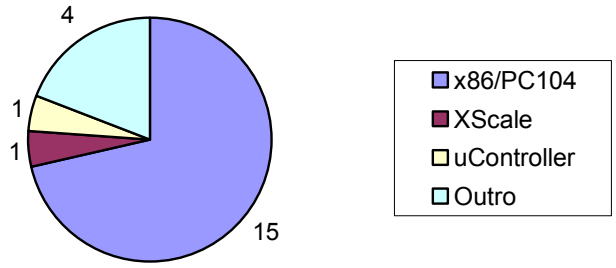

Figura 2-3: Distribuição das arquiteturas de processamento utilizadas.

Somente 3,0% (18 trabalhos - HONG, 2005; SUKKARIEH, 2003; KIM, 2004; KIM, 2007; BRYSON, 2007; ELFES, 2002; BUENO, 2003; JANG, 2006; TISDALE, 2006; HALL, 2001; HALL, 2002; EVANS, 2001; KIM, 2002; KIM, 2003; ATKINS, 1998; VIDOLOV, 2005; KUMON, 2006; FABIANI, 2006) dão detalhes do sistema operacional utilizado (Figura 2-4) sendo QNX, VXWorks, RTLinux, Linux e microC bem como da linguagem de programação utilizada (Figura 2-5) sendo C, C++ e Java.

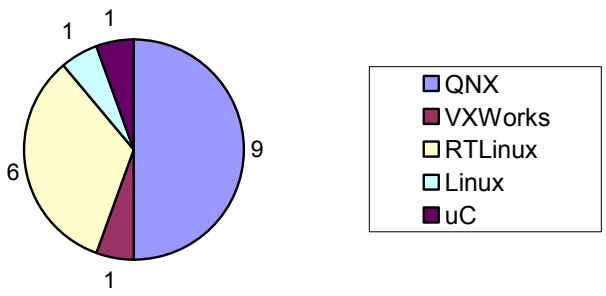

Figura 2-4: Distribuição dos sistemas operacionais utilizados.

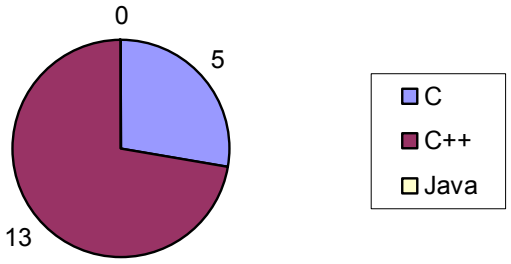

Figura 2-5: Distribuição das linguagens de programação utilizadas.

Somente 1,8% (11 trabalhos - HONG, 2005; SUKKARIEH, 2003; KIM, 2004; KIM, 2007; BRYSON, 2007; ELFES, 2002; BUENO, 2003; TISDALE, 2006; EVANS, 2001; KIM, 2002; KIM, 2003) projetam uma arquitetura de software.

Somente 1,5% (9 trabalhos - HONG, 2005; SUKKARIEH, 2003; KIM, 2004; KIM, 2007; BRYSON, 2007; JANG, 2006; TISDALE, 2006; HALL, 2001; HALL, 2002) analisam se os requisitos temporais estão sendo atendidos, a maior parte por meio de *deadline* não tomando a devida atenção com outros requisitos de tempo real como distribuição do consumo de processamento, cobertura de código, análise de erros de memória, análise dos eventos de *kernel*, *deadlocks*, *livelocks*, entre outros.

E finalmente, somente 1 trabalho (HONG, 2005) realiza o modelo do software, mas ainda demasiadamente superficial, representando o comportamento de todo o sistema por um diagrama de seqüência simplificado.

Nesta análise, ratificou-se a carência de estudos em desenvolvimento de sistemas aviônicos para VANTs com requisitos de homologação justificando a presente pesquisa. Ainda nesta análise, detectou-se que a seguinte configuração é a mais empregada: arquitetura de hardware centralizada com ECU PC104, sistema operacional de tempo real QNX, linguagem de programação C++ e verificação dos requisitos de tempo real por meio de *deadlines*.

Os trabalhos relatados na análise anterior foram resumidos em 15 projetos no capítulo 11 (ANEXO - Resumo dos Projetos Estado da Arte).

Nas próximas seções serão detalhados trabalhos que se destacam no desenvolvimento de hardware e desenvolvimento software.

2.1 Desenvolvimento de hardware

Um bom desenvolvimento de hardware deve seguir um processo de engenharia de sistemas que inclui as seguintes etapas: análise de requisitos de sistema para então filtrar os requisitos do hardware; projeto do processamento e da arquitetura de hardware; modelagem do sistema de hardware; simulação do comportamento do sistema de hardware. Com este processo é possível prever o comportamento do sistema de hardware incluindo modos de falhas críticas que determinam a confiabilidade do sistema.

O projeto Global Hawk (Figura 2-6) do grupo Northrop Grumman Ryan (LOEGERING, 1999) destaca-se no desenvolvimento do hardware. Este foi desenvolvido para a força aérea norte americana, para aplicação de monitoramento de fronteira, e possui homologação aviônica possibilitando seu vôo em regiões de espaço aéreo controlado.

Figura 2-6: Global Hawk.

Para o desenvolvimento do sistema aviônico, foram estabelecidos os seguintes requisitos:

- O VANT deve ser desenvolvido com projeção para futuro crescimento;
- O VANT deve ter não mais que 1 perda em 200 vôos;
- O VANT deve ser capaz de taxiar, decolar e pousar autonomamente, retornar ao hangar autonomamente e também navegar por *waypoints* e aceitar mudanças em qualquer momento na missão.
- A confiabilidade de vôo do VANT deve ser consistente com a operação segura sobre áreas habitadas, operar dentro do espaço aéreo sobre IFR.
- O VANT deve prover comunicação com o controle de espaço aéreo via voz;
- A exatidão do sistema de navegação deve ser inferior a 20m;
- O VANT deve ter um *transponder* capaz de operar no modo 4;

Para atender a todos estes requisitos foram feitas diversas análises principalmente no campo da confiabilidade para atender ao requisito de 1 falha em 200 vôos. Foi verificado que os modos de falhas mais críticas para hardware do sistema aviônico são:

1-Perda da IMU;

2-Perda dos atuadores de aileron, profundor e leme;

3-Perda do computador de controle de vôo;

4-Perda do ADB;

5-Perda do GPS por mais de 30 min.

Então foram feitas as análises de confiabilidade que indicariam a necessidade de redundância. Foi verificado que um sistema com dupla redundância para os componentes críticos (IMU, Atuadores de aileron, profundor e leme, Computador de vôo, ADB e GPS) garantiria a confiabilidade requisitada. Na Figura 2-7, pode-se observar a distribuição da aviônica no VANT e, na Figura 2-8, pode-se observar a arquitetura do sistema aviônico. Verificou-se também que um sistema em delta (redundância tripla) aumentaria a muito o custo por um pequeno aumento na confiabilidade.

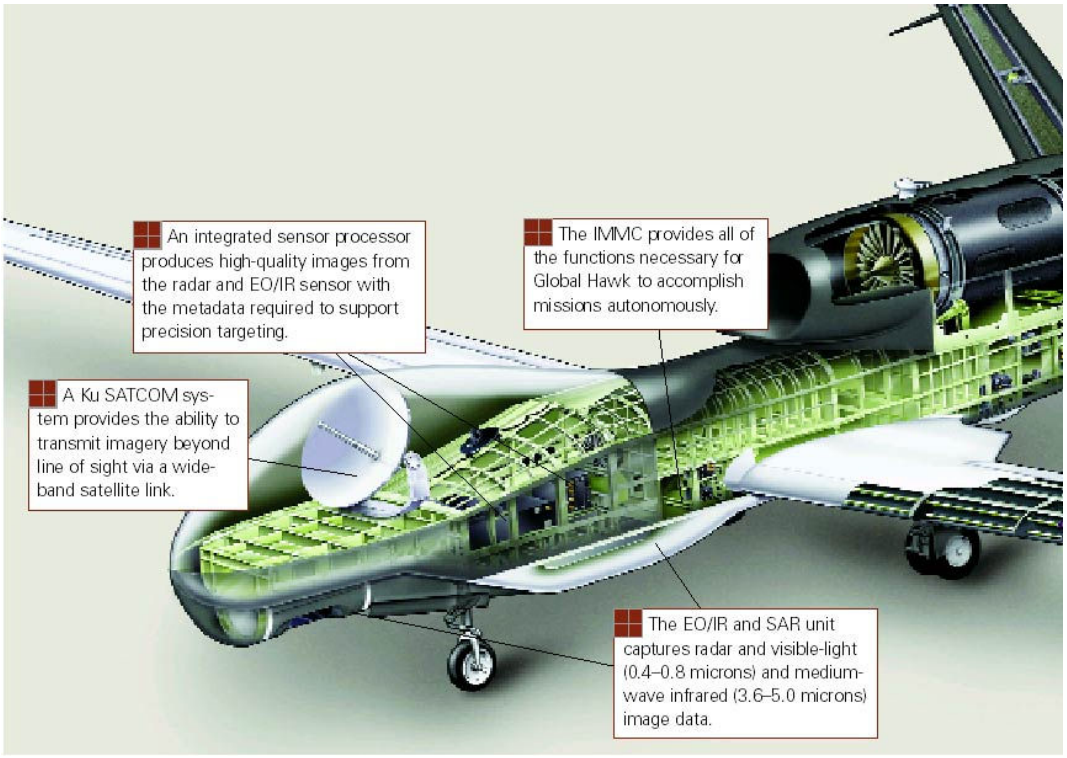

Figura 2-7: Sistema aviônico embarcado no Global Hawk (LOEGERING, 1999).

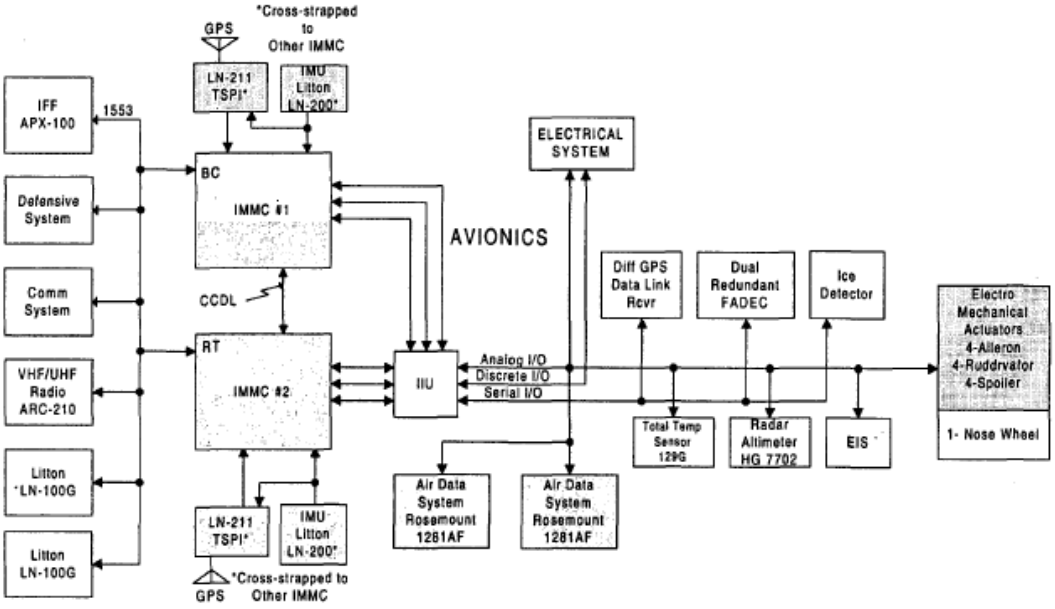

Figura 2-8: Arquitetura do sistema aviônico do Global Hawk (LOEGERING, 1999).

2.2 Desenvolvimento de software

Um bom desenvolvimento de software deve seguir um processo de engenharia de software que inclui as seguintes etapas: análise de requisitos de sistema para então filtrar os requisitos do software, projeto da arquitetura de software incluindo seleção do sistema operacional e linguagem de programação, modelagem e implementação do software, e testes. Pela etapa de testes é possível verificar se todos os requisitos funcionais e temporais especificados na etapa de análise de requisitos de software estão sendo atendidos.

O projeto Kyosho (Figura 2-9) da universidade nacional de Kyungpook (HONG, 2005) destaca-se no software, embora realize somente algumas etapas do processo de desenvolvimento de software. Este projeto concentra-se no estudo de arquiteturas de software para veículos aéreos autônomos.

Figura 2-9: VANT do projeto Kyosho (HONG, 2005).

Hong (2005) propõe uma arquitetura de software hierárquica cujo objetivo é suprir a deficiência de arquiteturas de software mais flexíveis capazes de suportar reconfiguração dinâmica de tarefas concorrentes sem a perda de qualquer *hard deadline* das tarefas de tempo real. Segundo ele, a maior parte dos pesquisadores considera somente problemas de controle e não como estes problemas serão resolvidos em tempo real, o que também foi evidenciado no início deste capítulo. Um esboço da arquitetura de software proposta por Hong (2005), para garantir a execução de tarefas em tempo real, pode ser observado na Figura 2-10.

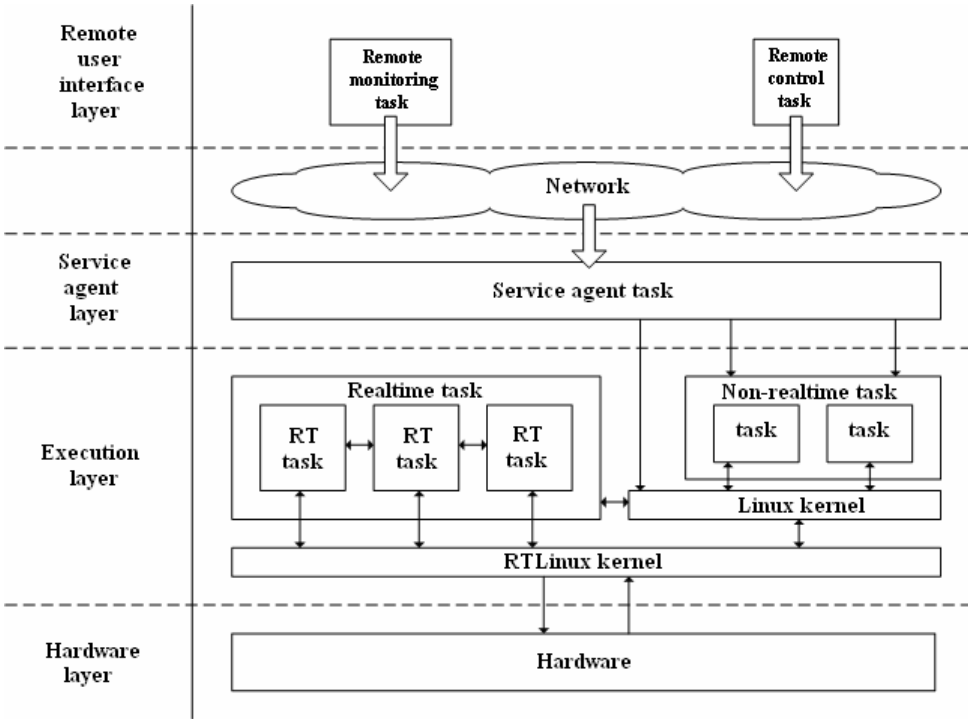

Figura 2-10: Arquitetura de software hierárquica do projeto Kyosho (HONG, 2005).

A arquitetura hierárquica possui quatro níveis: *Hardware*, *Execution*, *Service Agent* e *Remote User Interface*.

O nível de *Hardware* é composto por sensores, atuadores e computadores.

O nível de *Execution*, onde se alocam as tarefas de tempo real, é composto por tarefas de vôo autônomo sendo assim o nível de maior prioridade. Neste nível também podem ser alocadas tarefas que não são de tempo real, entretanto essas terão prioridade inferior às primeiras.

O nível de *Remote User Interface* é composto principalmente por tarefas de monitoramento e controle remoto, mas também, por comunicação do o nível *Service Agent* via rede.

O nível de *Service Agent* é composto pela tarefa servidor de monitoramento remoto. A tarefa *Service Agent* recebe requisições de informações da tarefa de monitoramento remoto do nível *Remote User*, através da rede, e decide pelo estado do VANT.

A seqüência de eventos do sistema (Figura 2-11) consiste em obter os dados dos sensores e do controle remoto via a porta serial do *Onboard Computer*. Em seguida, o sistema calcula a atitude usando o filtro de Kalman. Baseado na informação da IMU e CPS, a velocidade angular, velocidade linear e atitude são calculados em uma tarefa periódica. Estas informações são enviadas à base para estar monitorar o vôo. Os dados do GPS também são coletados e enviados para a estação base periodicamente. Na Figura 2-12 tem-se a seqüência de operações.

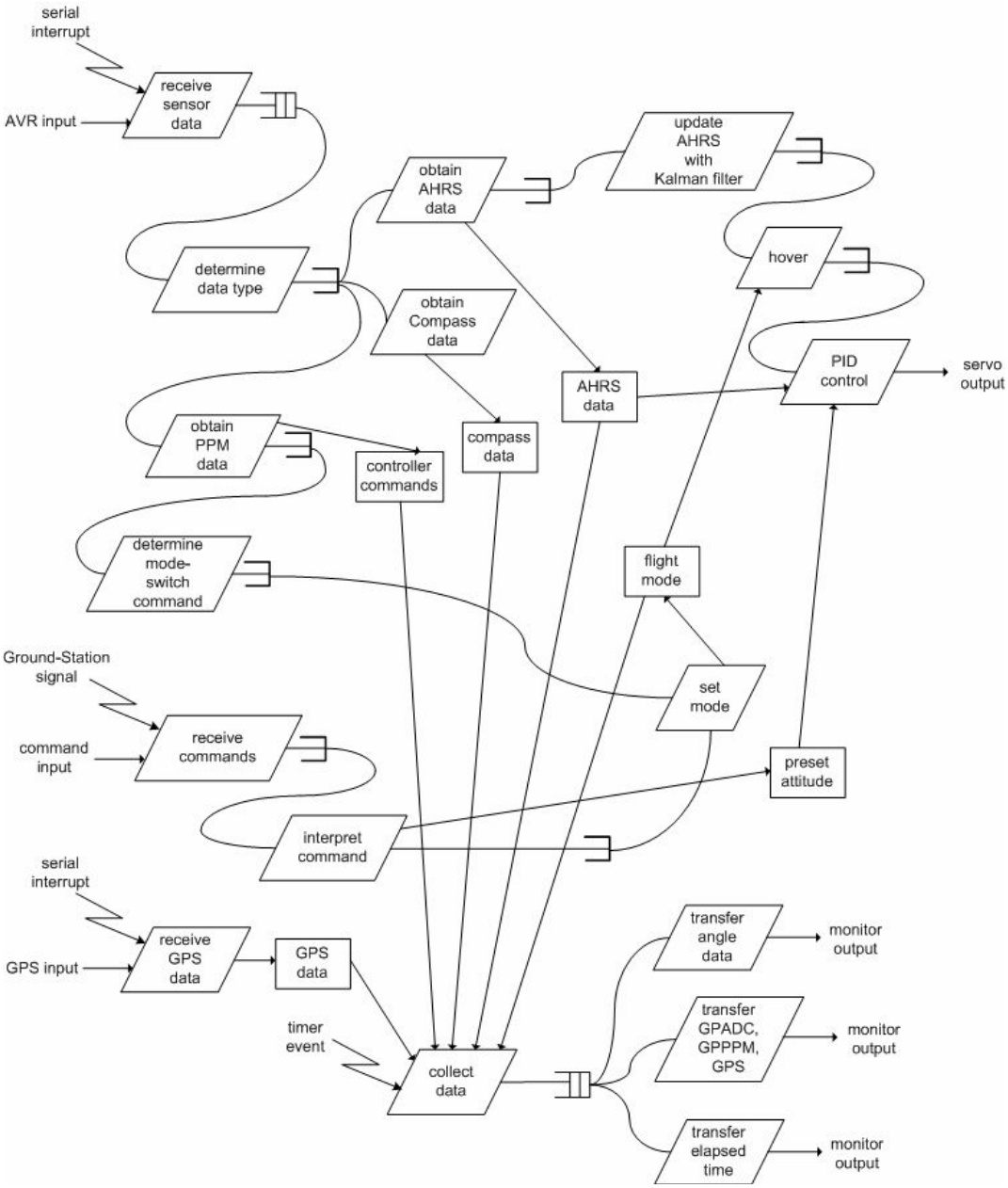

Figura 2-11: Diagrama de tarefas do projeto Kyosho (HONG, 2005).

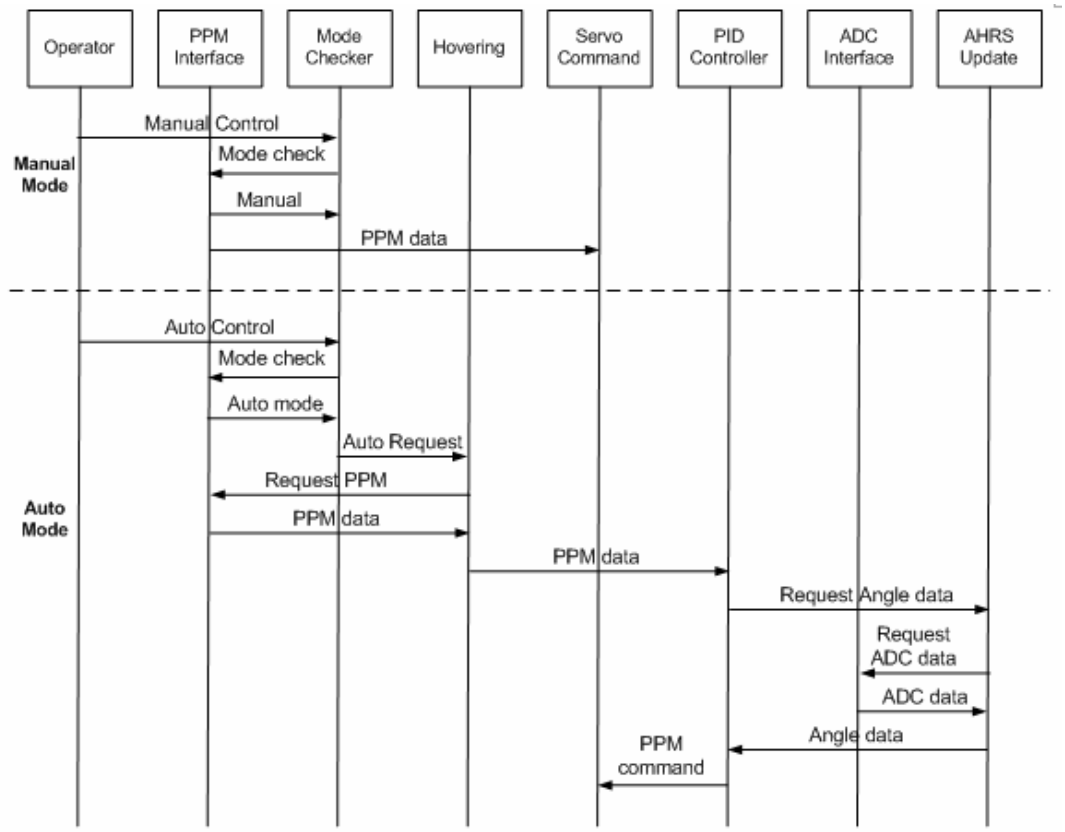

Figura 2-12: Diagrama de seqüência do projeto Kyosho.

O processador principal executa tarefas em múltiplos níveis de prioridade. Estas tarefas são divididas em alta, média e baixa prioridade. As tarefas de maior prioridade incluem aquisição de dados de sensores, atualização do AHRS (via filtro de Kalman), a realização do controle PID e o envio de dados para a estação base. Na Figura 2-13 pode-se observar que o requisito temporal para processar tais tarefas é de 320 microssegundos e na Figura 2-14 pode-se observar que tal requisito é cumprido uma vez que o tempo de processamento das tarefas de tempo real foi de 258 microssegundos.

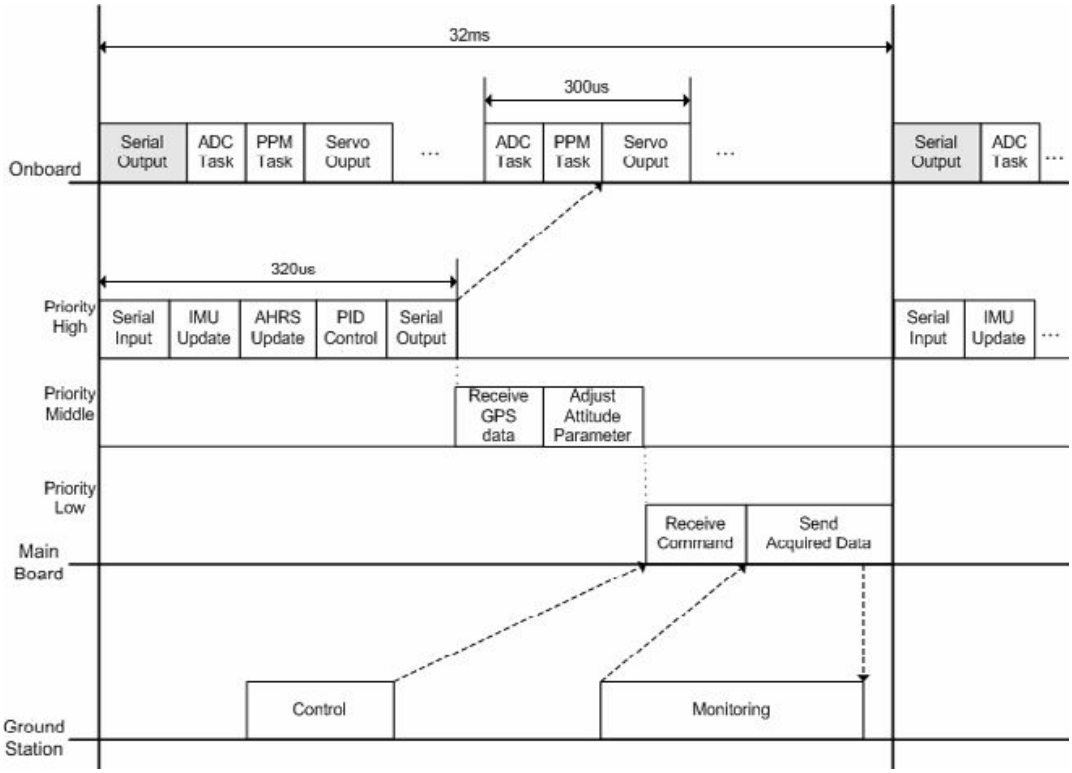

Figura 2-13: Diagrama de escalonamento do projeto Kyosho.

	Linux	RT-Linux
Average Value	32.242 ms	258 µs
Standard Deviation	1.669 ms	24.535 µs
Maximum Value	39.653ms	376 µs
Minimum Value	16.672ms	246 µs

Figura 2-14: Resultado do tempo de processamento das tarefas de tempo real.

2.3 Considerações finais deste capítulo

Neste capítulo, foi realizada uma análise de centenas de trabalhos que revelou a carência de estudos em desenvolvimento de sistemas aviônicos para VANTs com requisitos de homologação, o que justifica esta pesquisa. Também foi detectado que a seguinte configuração é a mais empregada em projetos aviônicos de VANTs: arquitetura de hardware centralizada com ECU PC104, sistema operacional de tempo real QNX, linguagem de programação C++ e verificação dos requisitos de tempo real por meio de *deadlines*.

Mais especificamente, no desenvolvimento de software com requisitos de homologação, que é o foco principal desta pesquisa, observou-se no projeto que se destaca pela qualidade do desenvolvimento de software, uma carência de diversas análises imprescindíveis para a homologação, que por sua vez serão detalhadas neste trabalho.

3 O PROJETO BR-UAV

Esta pesquisa integra-se ao projeto BR-UAV em desenvolvimento na empresa XMobots Sistemas Robóticos LTDA e no LVNT – Laboratório de Veículos Não Tripulados da USP.

O projeto visa a atender a necessidade de inserção da tecnologia VANT no Brasil. Para isso, desenvolve a **plataforma**, **aviônica** e **sistema controle autônomo** para aplicação de monitoramento **rural**, **preditivo**, **ambiental** e **urbano** a partir dos seguintes métodos de monitoramento: "espectro visível": (Fotografia e Filmagem) e "espectro infravermelho" (Fotografia e Filmagem).

O projeto foi dividido em fases, cada uma corresponde a um passo em direção ao objetivo final: aeronave cujo vôo seja completamente autônomo capaz de gerenciar a captação de imagens no espectro visível e do infravermelho.

Fase I – RCUAV – "*Radio Controlled Unmanned Aerial Vehicle*": A primeira fase incorpora o projeto da plataforma, ou seja, compreende os projetos: conceitual, aerodinâmico, estrutural, desempenho, estabilidade-controle e fabricação. O controle da aeronave é realizado por meio de equipamento de rádio controle utilizado em aeromodelos.

Fase II - RPUAV – "*Remotely Piloted Unmanned Aerial Vehicle*": A segunda fase tem como objetivo principal o vôo controlado por meio de instrumentação em uma Estação Base em Solo (EBS). Esta fase compreende o projeto aviônico e de interface com o piloto/usuário.

Fase III - AUAV – "*Autonomous Unmanned Aerial Vehicle*": A terceira e última fase do projeto consiste em projetar um sistema de controle autônomo.

3.1 Plataforma

Apoena, que em Tupi-Guarani significa "aquele que enxerga longe", é o nome da plataforma em desenvolvimento pela empresa XMobots. Esse nome advém do tipo de missão a ser realizada pelo VANT. As principais especificações físicas e de desempenho da plataforma podem ser observadas nas Tabela 3-1 e Tabela 3-2, respectivamente. Já o protótipo virtual pode ser observado nas Figura 3-1 e Figura 3-2.

Tabela 3-1: Especificações físicas da plataforma Apoena I.

Peso total	32 kg
Carga Paga (*Payload*)	6 kg
Combustível	Gasolina
Volume de combustível	10 litros
Envergadura	2,5 m
Comprimento	2,3 m
Diâmetro	0,4 m

Tabela 3-2: Especificações de desempenho da plataforma Apoena I.

Velocidade de estol (Flap 100%)	16 m/s
Velocidade de estol (Flap 0%)	19 m/s
Velocidade de cruzeiro	32 m/s
Velocidade máxima	45 m/s
Autonomia	8 h
Raio de alcance de link de dados	30 km
Distância de decolagem/pouso	80 m / 160 m
Largura da pista de decolagem	5 m
Potência do motor	5 Hp (2 tempos)

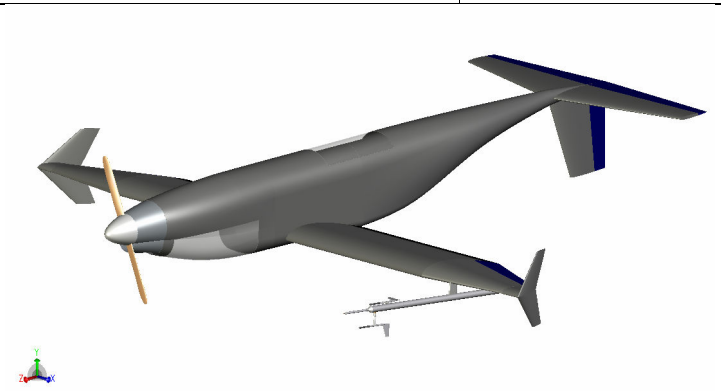

Figura 3-1: Vista dimétrica em vôo da Aeronave Apoena I _ PP7 (Fonte: XMobots).

Figura 3-2: Vista isométrica da Aeronave Apoena I _ PP7 (Fonte: XMobots).

3.2 Aviônica

Aviônica é uma tradução de "*Avionics*" do inglês, que por sua vez é devido do colapso de "*aviation*" e "*eletronics*" (RAYMER, 1992).

Na aviação civil e militar, a aviônica inclui: instrumentos de controle e navegação, atuadores (no caso, atuadores *fly-by-wire*), computadores, rádios, radares entre outros.

A aviônica neste projeto compreende o hardware e software embarcado e da estação base que, em conjunto, são capazes de controlar os vôos RC-UAV, RP-UAV e AUAV. Para tanto, em Amianti (2006) propõe-se uma arquitetura de hardware híbrida duplo redundante. Na Figura 3-3 pode-se observar um esquema da arquitetura de hardware aviônico.

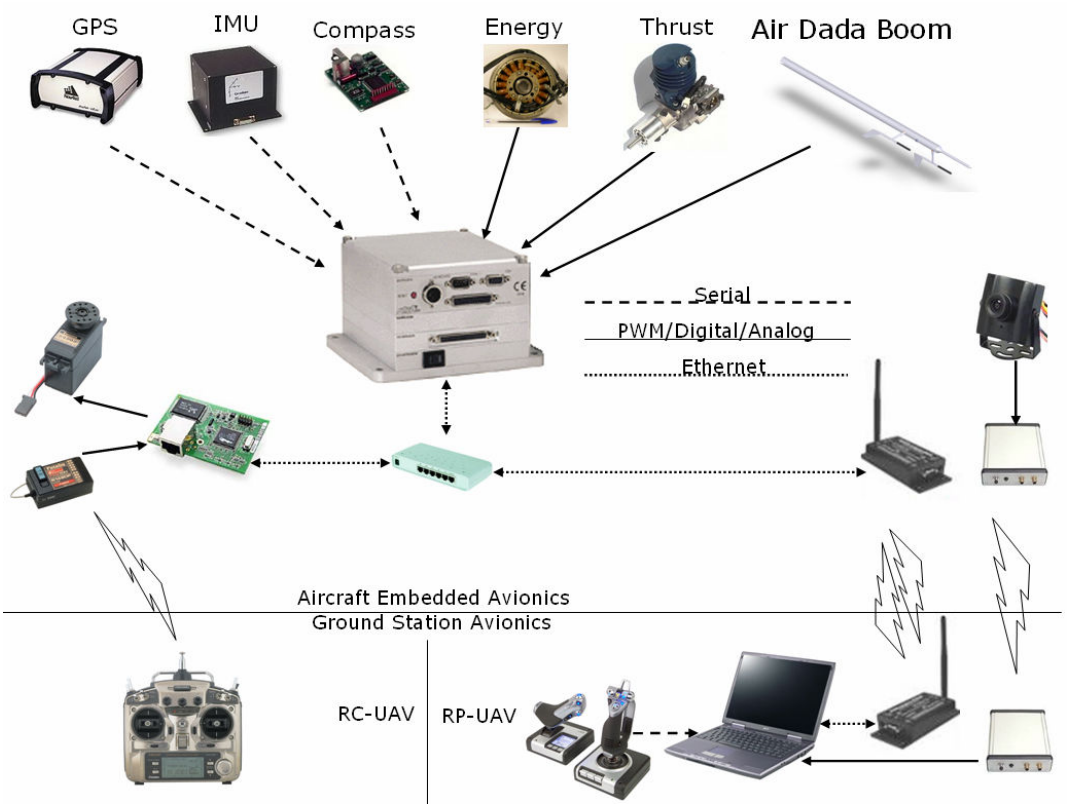

Figura 3-3: Arquitetura de hardware aviônico do VANT Apoena I.

O sistema de vôo rádio controlado ou RC-UAV, é composto por transmissor, receptor e servos, equipamentos estes equivalentes aos utilizados em aeromodelos. O controle do vôo é visual e externo à aeronave com visada direta, o que significa que o piloto observa do solo o vôo da aeronave e a partir desta informação a pilota pelo transmissor de rádio.

O sistema de vôo remotamente pilotado ou RP-UAV, possui dois métodos de controle. O método de controle VFR é composto por câmeras solidárias à aeronave, *joystick* e *notebook*. O controle de vôo é visual e interno à aeronave sem visada direta, o que significa que o piloto observa imagens transmitidas pela aeronave no *notebook* e a controla pelo *joystick*. Já o método de controle IFR é composto por instrumentação equivalente à utilizada na aviação civil e militar para vôo IFR bem como *joystick* e *notebook*. O controle do vôo é instrumentado o que significa que o piloto controla a aeronave pelo *joystick* somente considerando informações dos instrumentos mostradas no *notebook* sem o auxilio de qualquer imagem.

O sistema de vôo autônomo ou AUAV é composto por computação embarcada, sensores, atuadores e comunicação. A partir de uma missão carregada, o vôo deve ser autônomo sem qualquer intervenção humana.

3.2.1 Aviônica Embarcada

A aviônica embarcada neste projeto (AMIANTI, 2005; AMIANTI, 2006) consiste na integração de diversos sistemas mais precisamente: sistema de energia, sistema de observação, sistema de atuação, sistema de comunicação e sistema de processamento.

O **sistema de energia** é responsável pela geração, retificação/regulagem, distribuição/aterramento e armazenamento de energia elétrica (Figura 3-4) para a alimentação da plataforma, aviônica e *payload.* Para isso, é necessário gerar 250 W a 12 V e 140 W a 6 V a uma alta eficiência na conversão (>80%).

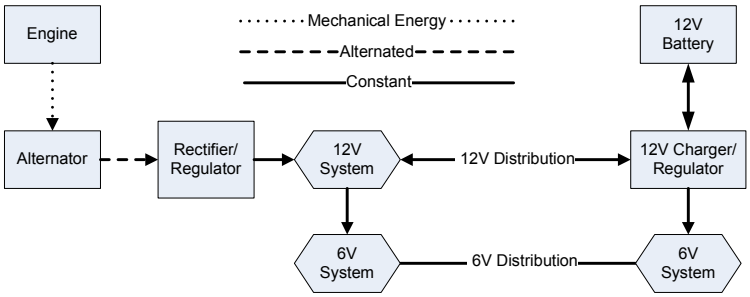

Figura 3-4: Esquema do sistema de energia.

O **sistema de observação** é responsável pela estimação dos estados da aeronave, para suprir tanto os vôos remotos VFR e IFR quanto os vôos autônomos.

Em Amianti (2005) definiram-se quais variáveis eram necessárias para o vôo RP-UAV, baseado na instrumentação regulamentada pela “FAR23 subpart F” (FAR, 2006) que gera as informações imprescindíveis para controle e navegação na aviação civil e militar nos vôos VFR e IFR que se assemelha consideravelmente ao RP-UAV. Definiram-se também quais variáveis eram necessárias para o vôo AUAV, com base em sistemas de controle e navegação autônomos bem como a dinâmica da aeronave Apoena I. Estabeleceu-se um conjunto de variáveis medidas que realimentam tanto o vôo RP-UAV e AUAV. As variáveis proprioceptivas são: temperatura dos gases de exaustão do motor, rotação do motor, nível de combustível (tanque principal), nível de combustível (tanque intermediário), consumo de combustível, tensão do alternador, corrente no alternador, tensão da bateria (nível de bateria), corrente na bateria. As variáveis heteroceptivas são: temperatura do ar (bulbo seco), umidade, pressão barométrica, densidade do ar, ângulo de ataque, ângulo de *sideslip*, velocidade do ar, latitude, longitude, altitude (mar), altitude (solo), velocidade longitudinal, velocidade lateral, velocidade vertical, ângulo de arfagem, ângulo de rolagem, ângulo de guinada, taxa de arfagem, taxa de rolagem e taxa de guinada.

A partir da definição das variáveis que devem ser medidas, desenvolveu-se uma tabela de seleção que minimiza a quantidade de sensores necessários, sendo os sensores necessários: GPS (Figura 3-5), IMU (Figura 3-6), CPS (Figura 3-7), ADB (Figura 3-8), VVF (Figura 3-9), TD (Figura 3-10) e ED (Figura 3-11).

Figura 3-5: GPS (posição, velocidade e aceleração).

Figura 3-6: IMU (atitude (Pitch e Roll), taxa de rotação e aceleração).

Figura 3-7: CPS (Roll, Pitch, Yaw e Campo Magnético).

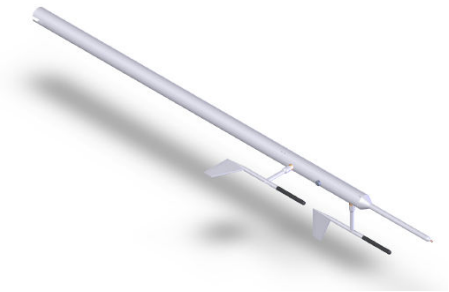

Figura 3-8: ADB (ângulos de ataque e side-slip, temperatura, umidade e pressão atmosférica e velocidade do ar).

Figura 3-9: VVF (responsável por simular por meio de câmeras solidárias à aeronave a visão do piloto se este estivesse a bordo, com o objetivo de permitir vôo VFR).

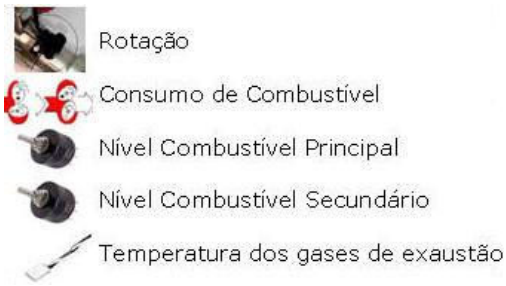

Figura 3-10: TD (temperatura dos gases de exaustão do motor, rotação do motor, nível de combustível do tanque principal-intermediário e consumo de combustível).

Figura 3-11: ED (tensão no alternador, corrente no alternador, tensão na bateria, corrente na bateria).

O **sistema de comunicação** é responsável pela comunicação entre os diversos nós embarcados (incluindo nós do *payload* e comunicação) e os diversos nós da base (incluindo nós de *payload* e comunicação), via rede sem fio, entre a rede da base e a rede da aeronave e vice versa. Na Figura 3-12 pode-se observar um diagrama do sistema de comunicação.

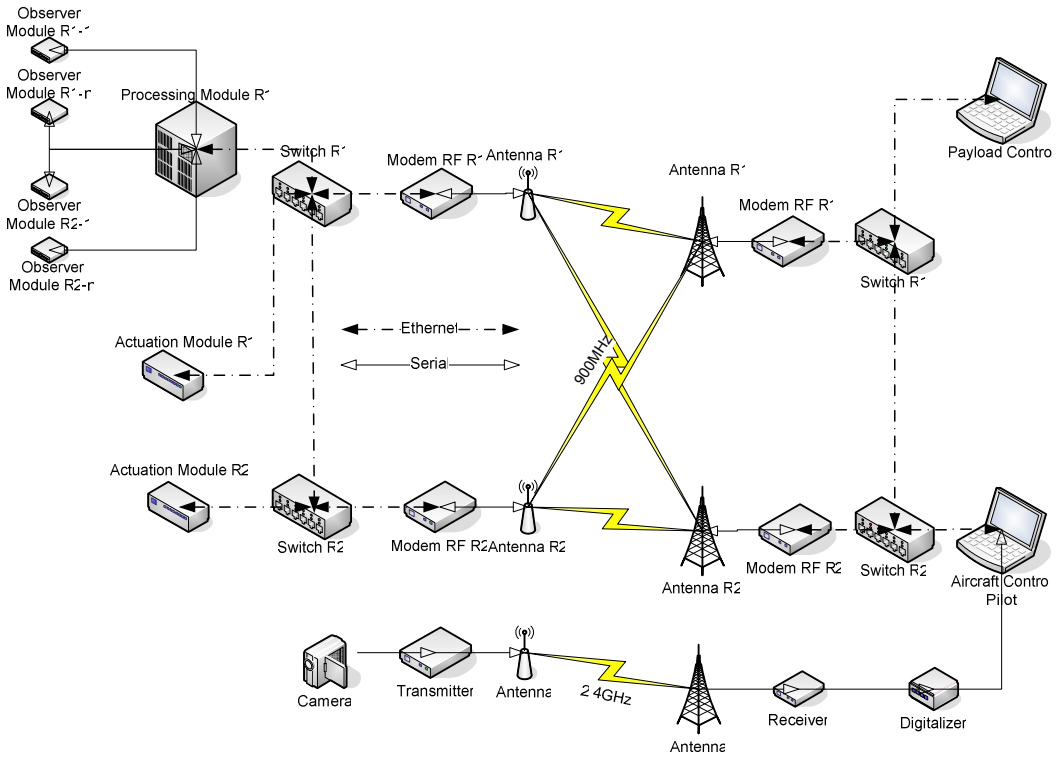

Figura 3-12: Conceito de comunicação na arquitetura híbrida duplo redundante.

O **sistema de atuação** é responsável pela atuação do: motor, freio, ailerons, flapes, profundor, leme, bequilha e pára-quedas. Este sistema é composto pelos receptores, módulo de atuação e servos. Na Figura 3-13 pode-se observar o sistema RC-UAV na parte superior composto por um transmissor e dois receptores RC e o sistema RP-UAV na parte central representado pelo *joystick*, *notebook*, *data-link* e PC-104. O módulo de atuação funciona como chave seletora do modo de pilotagem, sendo ainda sua função monitorar a comunicação dos receptores RC e gerar os sinais PWM a partir da referência gerada pelo PC-104.

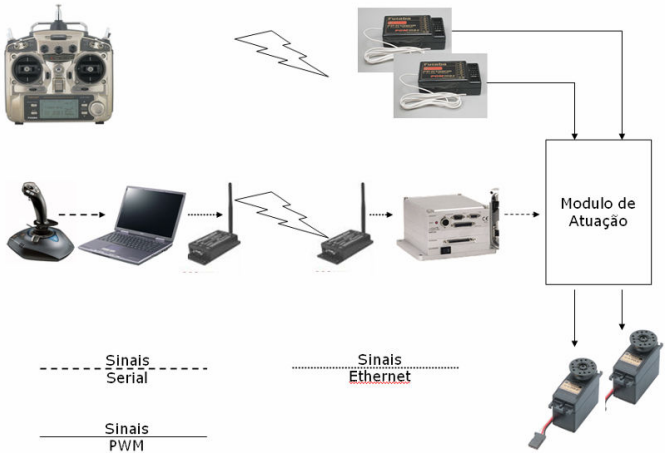

Figura 3-13: Esquema do sistema de atuação.

O **sistema de processamento** é composto pelo hardware e software embarcado.

O **hardware** é um PC104 da empresa Real Time Devices USA, Inc., contendo os seguintes módulos:

- CPU de 32 bits, *clock* de 1000 MHz, duas portas seriais RS232/422/485, porta paralela bidirecional ECP, interface para teclado e mouse e flash BIOS;

- Disco rígido de 3.2 Gbytes;

-Entradas e saídas analógicas e digitais, conversor analógico-digital, cronômetros/contadores;

- Controlador VGA com interface para monitor;

- Controlador NE2000 Ethernet;

- Expansor com quatro interfaces seriais.

O **software** é o principal foco de estudo deste trabalho e será tratado nos próximos capítulos.

3.2.2 Estação base

A estação base gerencia o vôo RP-UAV a partir da interface com o piloto (Figura 3-14). Nesta é utilizado o sistema operacional Linux e interface gráfica desenvolvida em Java.

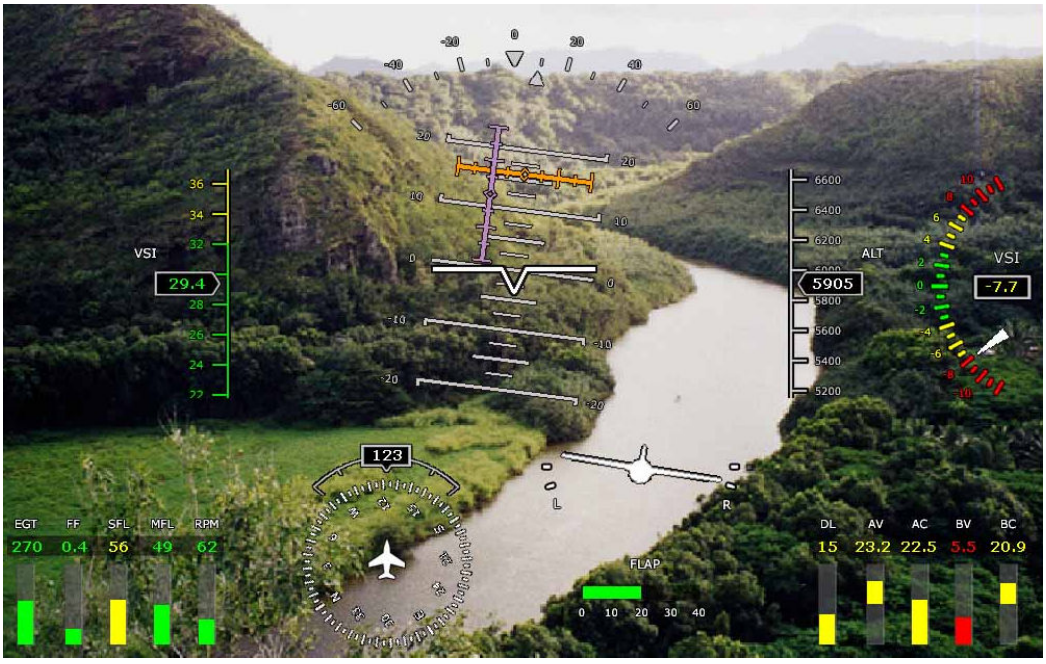

Figura 3-14: Interface com o Piloto do sistema RP-UAV (Fonte: XMobots).

3.3 Sistema Controle Autônomo

A função do sistema de controle autônomo é gerenciar o vôo autônomo a partir da missão, planejamento de trajetória, guiagem e controle e assim cumprir a fase III do projeto BR-UAV.

3.4 Considerações finais deste capítulo

Este capítulo discorreu um resumo do projeto BR-UAV fornecendo a base conceitual para a seqüência do projeto do software aviônico embarcado, objeto desta pesquisa.

Na plataforma Apoena I, foram detalhadas as principais especificações físicas e de desempenho. Estas serão essenciais para determinar a dinâmica da plataforma que por sua vez rege a definição requisitos temporais do sistema aviônico como a taxa de amostragem para observar a dinâmica da plataforma.

Na aviônica, foram detalhadas as principais características da arquitetura hardware bem como os sistemas de hardware que foram projetados em trabalhos anteriores. Estas características são essenciais para a modelagem do hardware que é utilizada na definição dos requisitos do software.

Já, no sistema controle autônomo, somente foi enumerado as funções que deverão ser implementadas em trabalhos futuros.

4 METODOLOGIA DE DESENVOLVIMENTO

Nesta pesquisa a metodologia é balizada por três elementos básicos de desenvolvimento: processo, normas e ferramentas. Nas próximas seções cada um destes elementos será detalhado para o contexto de software aviônico embarcado com requisitos de homologação.

4.1 Processo de desenvolvimento

Um processo de desenvolvimento pode ser definido como um conjunto de atividades que organizam e deixam claro o que e quanto precisa ser feito. A razão básica de se utilizar um processo de desenvolvimento do sistema bem como do software é a melhoria da eficiência de desenvolvimento, ou seja, produzir sistemas com maior confiabilidade e menor custo. Especificamente, um bom processo deve:

- Prover um guia desde a definição de requisitos até a entrega do sistema final, passando por projeto, implementação e testes;
- Melhorar a qualidade do sistema em termos de: diminuição do número de defeitos, redução da severidade dos defeitos, potencialização da reutilização, melhoria da estabilidade e manutenção;
- Melhorar a capacidade de predição do projeto em termos de: custo e tempo total;
- Melhorar a comunicação em equipes multidisciplinares.

Segundo Douglass (2007), o processo de desenvolvimento em V é tradicionalmente utilizado em indústrias de sistemas tempo real crítico como a aeronáutica. No entanto, o mesmo autor ressalta que esta abordagem é incompleta, pois não prevê iterações de projeto e não especifica diversas fases do desenvolvimento. Ainda ressalta que, devido ao grande tempo necessário para o desenvolvimento de componentes mecânicos e elétricos, é importante que todos os requisitos e arquitetura estejam totalmente definidos e entendidos antes de qualquer projeto significante ocorra. Por esta razão, Douglass (2007) propõe o processo Harmony que é um hibrido do processo em “V” e do processo em espiral. Na Figura 4-1 pode-se observar uma visão geral do processo de desenvolvimento Harmony. Este pode ser dividido em duas grandes fases, a fase de engenharia de sistemas (Figura 4-2) e a fase de engenharia de software representada pela espiral (Figura 4-3).

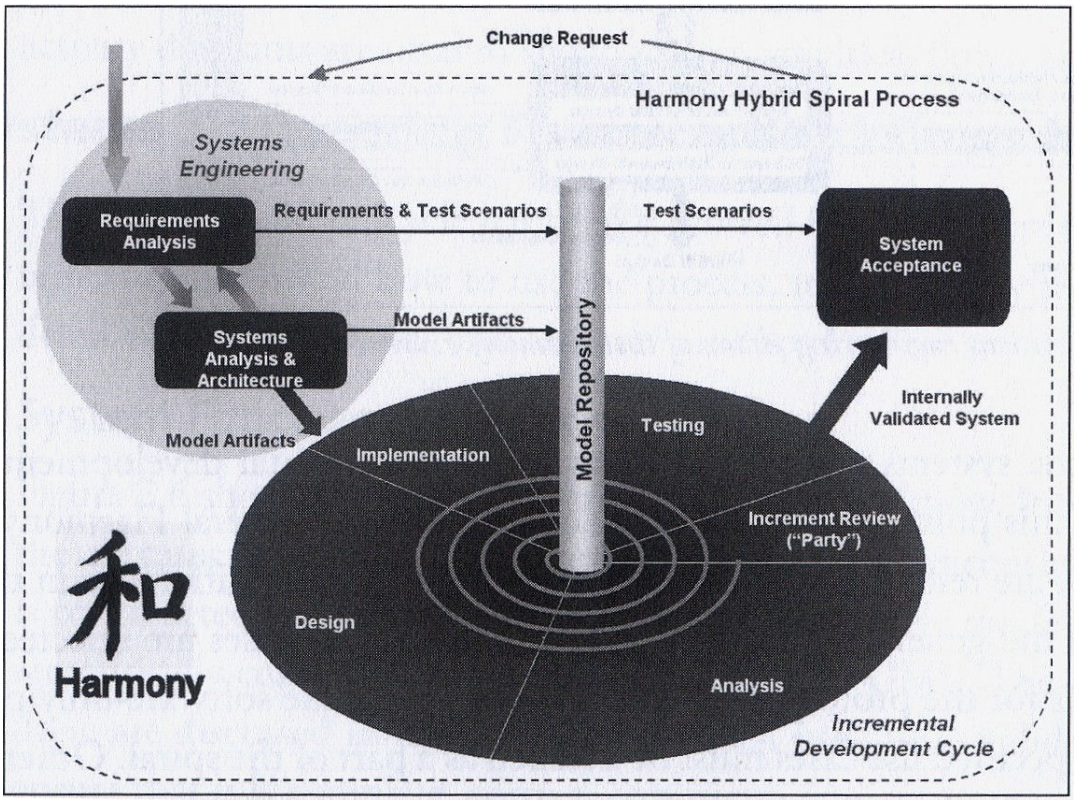

Figura 4-1: Visão geral do processo de desenvolvimento Harmony (DOUGLASS, 2007).

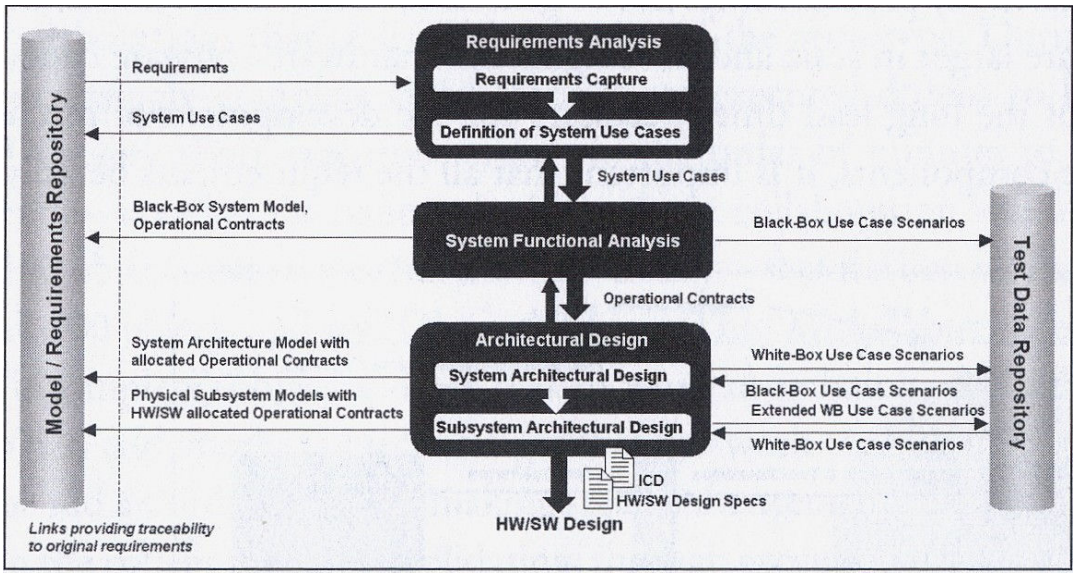

Figura 4-2: Fase de engenharia de sistemas do processo de desenvolvimento Harmony (DOUGLASS, 2007).

Entende-se por sistema tudo aquilo que está em desenvolvimento e, no caso do projeto BR-UAV compreende: a plataforma, hardware embarcado, hardware da estação base, software embarcado e software da estação base.

Na fase de engenharia de sistemas, inicialmente são modelados os requisitos do sistema (seção 5.1) que compreende a captura das necessidades tanto funcionais como temporais da missão do sistema. Em seqüência, é projetada e modelada a arquitetura do sistema. (seção 5.2). Nesta etapa, entende-se por arquitetura de sistema a estrutura de relacionamento entre os principais módulos de hardware e software. Conforme foi observado no capítulo 3, o projeto da arquitetura de hardware já foi realizado em trabalhos anteriores cabendo nesta etapa a modelagem da arquitetura de hardware e o projeto e modelagem conceitual da arquitetura de software. De posse do modelo do sistema, e por meio de simulações, são filtrados os requisitos de software que serão utilizados na etapa de análise da fase de engenharia de software.

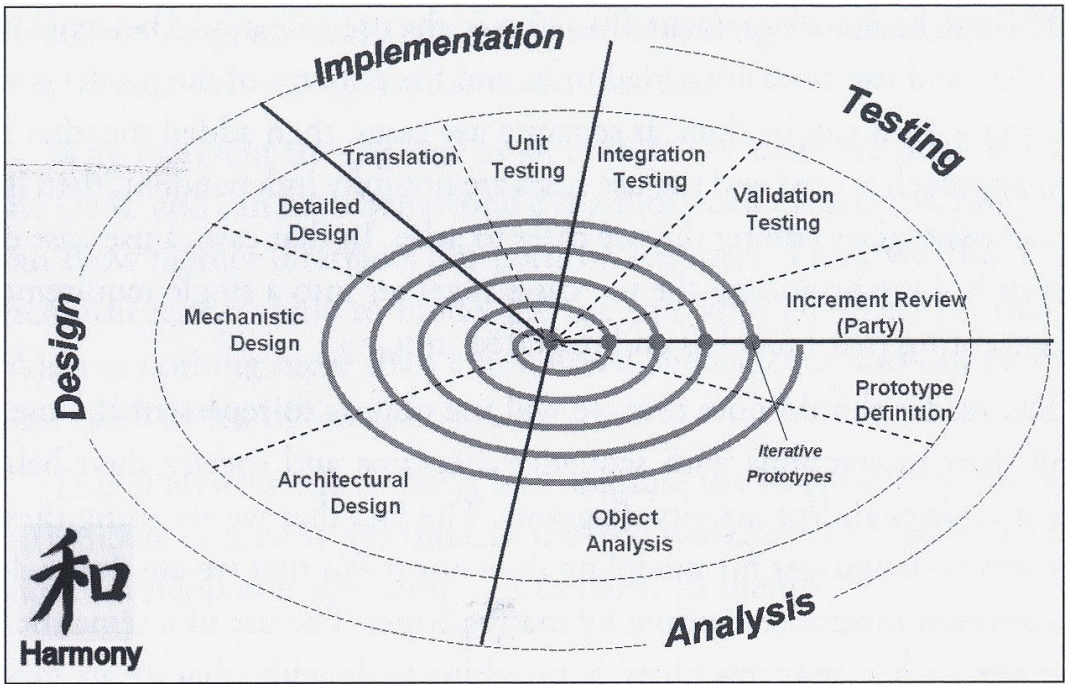

Figura 4-3: Espiral do processo de desenvolvimento Harmony (DOUGLASS, 2007).

A fase de engenharia de software compreende a análise, projeto, implementação e testes de módulos de software que são executados nos respectivos módulos de hardware modelados na fase de engenharia de sistemas. Neste trabalho, será desenvolvido o módulo de software que é executado na ECU central (PC-104) da arquitetura de hardware proposta na seção 3.2.

A etapa inicial da fase de engenharia de software, a análise (*Analysis*) (seção 5.3), consiste na captura dos requisitos tanto funcionais como temporais da missão do software por meio da definição do protótipo. Em seguida, na análise de objetos, esses requisitos são refinados até o ponto de se obter a estrutura e o comportamento requeridos do software em desenvolvimento. A análise estrutural de objetos identifica as unidades estruturais do software, sua organização e as respectivas relações entre esses elementos, refinando principalmente os requisitos funcionais estabelecidos no protótipo. Já a análise de comportamento de objetos

define os modelos de comportamento essenciais, refinando principalmente os requisitos temporais estabelecidos no protótipo. No final da etapa de análise, os modelos estruturais e comportamentais são avaliados por toda a equipe de sistemas (plataforma, hardware e software) e pelo cliente de forma a garantir que todos os requisitos foram bem entendidos e traduzidos. Para garantir que o software final atenderá a todos estes requisitos, os modelos estruturais e comportamentais são cruzados com a resposta do produto final (software).

Em seguida, a etapa de projeto (*Design*) pode ser subdividida nos passos projetos de arquitetura, mecanicista e detalhado. O passo de projeto da arquitetura (seção 6.1) identifica os principais módulos de software e sua estrutura de relacionamento. O passo de projeto mecanicista (seção 6.2) consiste na realização da arquitetura projetada no passo anterior. Basicamente compreende a aplicação da arquitetura (como se fosse um padrão de projeto) em todo o software. O passo de projeto detalhado (seções 7.1, 7.2, e 7.3) consiste basicamente em “dar vida” aos módulos de software. São definidas as estruturas internas dos módulos, incluindo frequentemente padrões, visibilidade e tipo de dados dos atributos, as realizações das associações, agregações, e composições até que os módulos sejam implementáveis.

A etapa de implementação (*Implementation*) (também nas seções 7.1, 7.2, e 7.3) pode ser subdividida nos passos de tradução e testes unitários. A tradução consiste em codificar todo o modelo desenvolvido na etapa de projeto. A tradução utiliza um conjunto de regras para converter os modelos em código. E para cada realização de cada módulo de software são realizados os testes unitários que validam seu atendimento aos requisitos.

Finalmente, na etapa de testes (*Testing*), são realizados os testes de integração dos diversos módulos de software. A validação do sistema é realizada a partir do cruzamento dos resultados dos testes com os requisitos estabelecidos na fase de análise.

4.2 Normas de desenvolvimento

A homologação é um requisito estabelecido para as aeronaves tripuladas, sendo o certificado de Aeronavegabilidade (RBHA, 2007) o mais importante da homologação. Este certificado é expedido pela autoridade regulamentadora que no caso do Brasil é a ANAC e, que por sua vez, procura estar compatível com as autoridades regulamentadoras dos países desenvolvidos como a FAR (EUA) e JAR (Europa). Este certificado atesta a uma aeronave, que se corretamente operada, é capaz de decolar, manter vôo e pousar de forma segura, representado que o risco de acidente é baixo o suficiente para que seja aceitável pela sociedade.

Diversos países investem significativamente em pesquisas para possibilitar a aplicação de VANTs em espaço aéreo controlado, o que possibilitaria vôo sobre cidades e áreas povoadas. Os esforços se concentram na determinação de normas de homologação e operação como ocorre na Austrália (CASA, 2006) e EUA (DO-304, 2007) bem como em propostas conjuntas entre EUA e Europa (AVIONICS, 2003). No Brasil já existem esforços para

determinar um regulamento brasileiro de homologação aeronáutica direcionado para VANTs como a RBHA-100 (2004) proposta em 2004 e que até então se encontra em discussão.

Embora as normas para o desenvolvimento de VANTs ainda estejam em discussão, segundo HighRely (2006) as normas DO-178B (1992), no que se refere ao software, e DO-254 (2000), no que se refere ao hardware, já são utilizadas com sucesso como normas para homologação de aviônicos de VANTs nos EUA. Pelo fato desse país ter o maior mercado de aviação, estas normas acabam sendo aplicadas ou adaptadas no mundo inteiro como é o caso da ED-12B equivalente européia da DO-178B. Sendo assim, o atendimento às exigências das normas DO-178B e DO-254 são passos muito importantes para obtenção de certificado de aeronavegabilidade, no Brasil e no mundo.

As normas DO-178B e DO-254, produzidas pela Radio Technical Commission for Aeronautics, Inc. (RTCA), tornaram-se um padrão para aviônicos em todo mundo de forma que o *Advisory Circular* AC20-115B (FAA-AC, 2007) da FAA classificou a DO-178B como meio de certificar qualquer software aviônico. O sucesso da DO-178B e DO-254 na aviação tornaram-se base de muitos processos de certificação em outras indústrias.

A norma DO-254 aplica-se a todas as estratificações do desenvolvimento de hardware, incluindo: *Line Replaceable Units* (LRU), *Circuit Card/Board Assemblies* (CCAs), componentes customizados de micro-código (ASIC, PLD, FPGA, CPLD), componentes integrados híbridos (CIH), componentes multi-chip e dispositivos COTS; como pode ser observado na Figura 4-4.

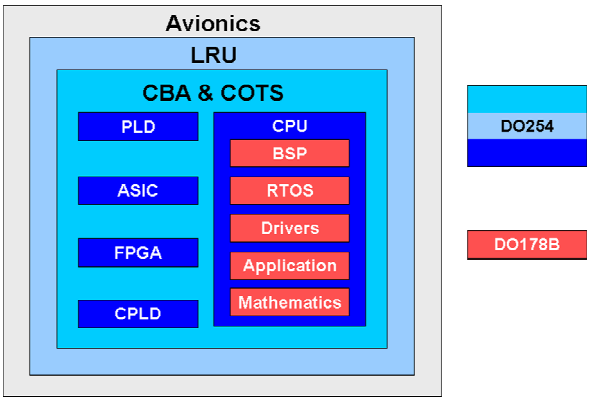

Figura 4-4: Organização da DO254 e DO178B.

Neste ponto ja é evidente que o foco desta pesquisa concentra-se no software aviônico. Assim, é importante garantir que o hardware com o qual este interage seja certificado segundo a norma DO-254 para não se ter interferências indesejáveis do hardware no software. No projeto do VANT Apoena I todo o hardware que interage com o software é COTS e possui certificação da FAA permitindo o desenvolvimento do software com reduzido risco de influência por erros de hardware.

A norma DO-178B aplica-se a qualquer software que roda em uma CPU, incluindo pacotes de suporte de hardware, sistemas operacionais, *drivers*, aplicações e cálculos

matemáticos, ou em outras palavras, qualquer executável que é carregado dentro da memória durante a execução; como pode ser observado na Figura 4-4.

A norma DO-178B define uma série de orientações para o processo de desenvolvimento de software aviônico bem como requere muita documentação e arquivos para a comprovação do atendimento aos requisitos da norma. A quantidade de componentes necessários para a certificação e o nível de certificação determina a quantidade de informação que a documentação deve conter. Os níveis de certificação A, B, C, D e E correspondem às conseqüências de uma falha potencial do software, sendo: catastrófica, severa, maior, menor e sem efeitos, respectivamente.

Segundo HighRely (2005) a carga de trabalho no desenvolvimento de software aviônicos para atender a DO-178B é distribuída conforme a Figura 4-5. Neste contexto fica evidente que a automação tarefas repetitivas como codificação e testes que atendam a norma seria uma forma de reduzir significativamente o custo de desenvolvimento de software. Ainda segundo HighRely (2005) o desenvolvimento de software aviônicos para atender a DO-178B sem automação aumenta o custo de desenvolvimento entre 60% e 100% com programadores experientes, já com automação este custo reduziria para a faixa entre 20% e 40%.

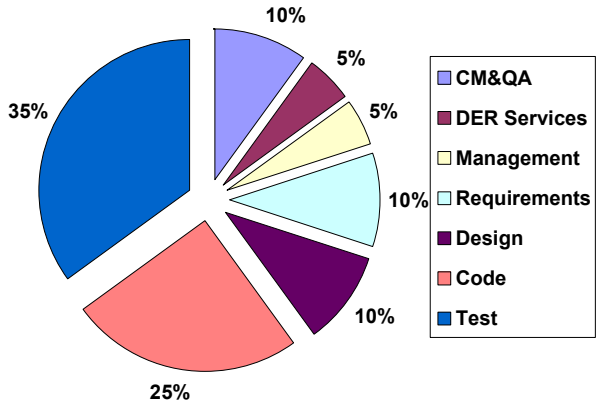

Figura 4-5: Carga de desenvolvimento segundo a DO178 (HIGHRELY, 2005).

Poucas ferramentas de desenvolvimento e análise de software podem ser utilizadas na difícil tarefa de desenvolver sistemas e software para aplicações com requisitos como a DO-178. Entretanto na Figura 4-6 pode-se observar uma proposta de ferramentas que atendem a DO-178B bem como viabilizam automação de código e teste. Nas seções seguintes será realizada a análise de cada uma delas.

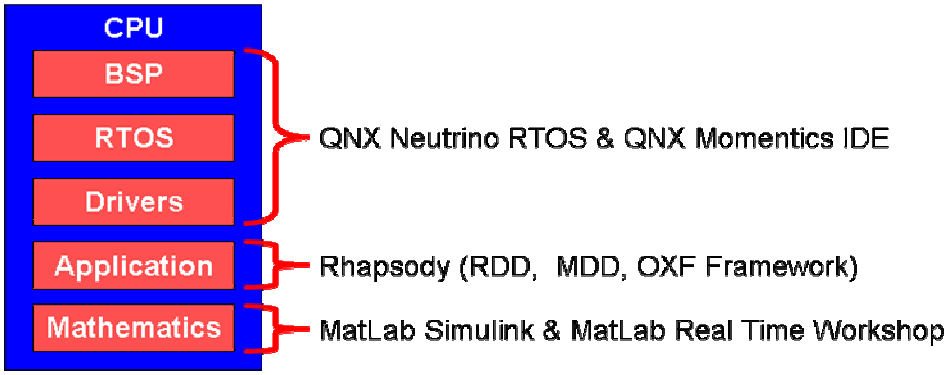

Figura 4-6: Proposta de ferramenta de automação para o desenvolvimento aviônico.

4.3 Ferramentas de sistema operacional (BSP, RTOS e Driver)

4.3.1 Características de sistemas operacionais de tempo real

Um sistema operacional (SILBERSCHATZ, 2001; STALLINGS, 2001; TANENBAUM, 2002) age como um intermediário entre a aplicação e o hardware e para tanto este inclui pacotes de suporte de placas (BSP) e *drivers*. Assim seus objetivos principais são desde executar e gerenciar aplicações até gerenciar a utilização dos recursos de hardware do computador numa maneira eficiente.

As características do ambiente com o qual o sistema operacional interage determina o sistema operacional adequado. Como o sistema em questão possui as características de um sistema de tempo real, principalmente restrições temporais bem determinadas, verifica-se a necessidade de utilizar um sistema operacional de tempo real.

O sistema operacional de tempo real (RTOS) (FARINES, 2000; LI, 2003; LAPLANTE, 2004) é definido como um sistema de software gerenciador dos recursos de hardware de uma aplicação de tempo real (essencialmente, um sistema em que cumprimento de prazos é crítico). Quando se diz que o cumprimento de prazos é crítico, isso não implica necessariamente em alta velocidade de processamento, mas sim, principalmente, determinismo nas respostas do sistema.

Enquanto sistemas operacionais convencionais apresentam um bom comportamento médio, distribuindo os recursos de forma justa entre as tarefas e os usuários, sistemas operacionais de tempo real definem o sistema não somente em termos funcionais, mas também temporais. Assim, por meio de uma estrita ligação entre processos computacionais e processos físicos, os sistemas operacionais de tempo real destacam-se pelas seguintes características: resposta temporal, simultaneidade, previsibilidade e robustez. Essas características podem ser definidas da seguinte forma:

- **Resposta Temporal**: capacidade de fornecer resultados dentro de prazos de tempo impostos (sistemas de tempo real possuem uma forte correlação entre resultado correto no tempo correto).

- **Simultaneidade**: capacidade de responder a entradas de duas ou mais fontes independentes dentro de determinados limites de tempo por meio de processamento paralelo em processadores distintos ou multiprogramação sobre um único processador.

- **Previsibilidade**: capacidade de se afirmar algo a respeito do comportamento futuro de um determinado sistema de forma determinística ou probabilística.

- **Robustez**: capacidade de permanecer dentro de um estado previsível mesmo que as entradas fornecidas pelo ambiente não estejam dentro de faixas especificadas ou quando uma parte do sistema falha.

Geralmente esses sistemas são utilizados como um dispositivo de controle em uma aplicação dedicada controlando experimentos científicos, sistemas de imagem médicos, sistemas de controle industriais e alguns sistemas de exibição. Em sistemas embarcados, a aquisição de dados sensoriais e utilização dos mesmos no controle e na manutenção da trajetória do robô móvel, muitas vezes evidenciam a necessidade de execução de tarefas em tempo real.

Existem três categorias de sistemas de tempo real referentes ao não cumprimento dos requisitos de tempo:

- **Sistemas de tempo real crítico** (*Hard Real Time Systems*): Nos casos em que um atraso na execução de uma tarefa é potencialmente prejudicial ao processo como um todo, classifica-se o sistema como de tempo real crítico. Geralmente, nesses casos, os erros precisam ser rastreados de forma a permitir uma ação imediata caso algo fora do planejado ocorra. Exemplos de sistemas de tempo real crítico são sistemas de guiagem de mísseis e sistemas de controle de vôo.

- **Sistemas de Tempo Real Não-Crítico** (*Soft Real Time Systems*): Quando um sistema de tempo real executa uma tarefa em um tempo superior ao determinado, mas os resultados ainda podem ser utilizados sem grandes conseqüências para o cumprimento da missão, diz-se que o sistema é de tempo real não-crítico. Esses possuem uma utilização limitada no controle industrial robótico ou em sistemas embarcados. Muitas vezes, são representados por sistemas de aquisição de dados.

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

Uma aplicação de tempo real é tipicamente um programa concorrente, formado por um conjunto de tarefas conhecidas como processos e *threads* que se comunicam. Um requisito

básico para os sistemas operacionais em tempo real, então, é oferecer suporte para processos e *threads*.

Um **processo** é um programa em execução. Geralmente num processo estão inclusos: um contador de programa, uma pilha com dados temporários (como parâmetros de métodos, endereços de retorno e variáveis locais) e uma seção de dados com as variáveis globais.

Uma **thread** é um "processo leve" que inclui um contador de programa, um conjunto de registrador e uma pilha. A principal característica de uma *thread* comparada a um processo é que ela compartilha com outras *threads* pertencentes ao mesmo processo sua seção de código, seção de dados, arquivos abertos etc. Dessa maneira, a utilização de *threads* acelera a troca de contexto, economiza recursos e facilita a multiprogramação.

Outro fator essencial de um sistema de tempo real é o escalonamento. O **escalonamento** serve para organizar qual processo será tratado pela CPU a cada momento. Para tal, os processos são divididos em três filas principais, essas são:

- **Fila de trabalho**: conjunto de todos os processos no sistema;
- **Fila de espera**: conjunto de todos os processos residentes na memória principal, prontos e aguardando para serem executados;
- **Fila de dispositivos**: conjunto de processos aguardando por um dispositivo de I/O;

Há escalonadores de longo prazo que selecionam quais processos devem ser levados à fila de espera e escalonadores de curto prazo que selecionam quais processos nessa fila devem ser executados em seguida, alocando a CPU. Um escalonador de curto prazo costuma ser chamado freqüentemente (ordem de milisegundos). Um escalonador de longo prazo, que também controla o grau de multiprogramação, pode ser evocado com uma freqüência menor, da ordem de segundos ou minutos.

A decisão de escalonamento da CPU pode ser necessária quando um processo:

- Troca seu estado de executando para aguardando, por exemplo, devido a um requerimento de I/O;
- Troca seu estado de executando para pronto, voltando à fila de espera, por exemplo, quando uma interrupção ocorre;
- Troca seu estado de aguardando para pronto, voltando à fila de espera, por exemplo, devido à conclusão de uma E/S;
- Termina.

Em sistemas de tempo real, é muito comum que os processos recebam uma priorização, ou seja, um número inteiro associado a cada processo que determina sua importância quando comparado aos outros processos. Dessa forma a CPU pode ser alocada

ao processo com maior prioridade. Nem sempre isso é necessário, no entanto. Os quatro tipos de escalonamento utilizados pelos sistemas operacionais são:

- **FIFO** (*First In, First Out*): O processo atual continua a usar o tempo da CPU até que fica em estado bloqueado ou finaliza.
- **SJF** (*Shortest Job First*): Associa a cada processo o tempo que o mesmo consome da CPU (através de um cálculo estimado) e usa esse período para priorizar o processo com menor tempo para ser finalizado.
- **RR** (*Round Robin*): Cada processo recebe uma parcela do tempo da CPU, geralmente entre 10 e 100 milisegundos. Depois que esse tempo se esgota, o processo cede lugar ao próximo e fica na fila de espera.
- **Adaptativo**: Semelhante ao RR porém, quando um processo consome sua parcela do tempo, tem sua prioridade reduzida ao retornar à fila de espera.

Escalonamentos do tipo SJF, RR e Adaptativo podem ser preemptivos ou não-preemptivos. Quando se utiliza um escalonamento **não-preemptivo**, o processo em execução não cede a CPU para outros processos a menos que seja concluído. Já com a utilização de um escalonamento **preemptivo**, um processo em execução pára de ser executado quando um processo de prioridade superior chega à fila de espera. Esse processo aguarda que o novo processo seja tratado e volta a ser executado num momento oportuno.

Quando um sistema de tempo real é avaliado, essas características costumam ser apenas alguns dos parâmetros considerados. A próxima seção descreve como realizar uma comparação razoável para a tomada de decisão de que RTOS utilizar.

4.3.2 Seleção do Sistema Operacional de Tempo Real

Segundo Dedicated Systems Experts (2006) os sistemas operacionais mais aplicados a sistemas em tempo real são QNX, Red Hat ELDS, Vx Works e Windows CE .NET. e as Tabela 4-1 a Tabela 4-4 comparam esses sistemas operacionais segundo sete parâmetros:

- **Instalação e configuração** (*Installation and Configuration*): considera a rapidez e facilidade de instalação, a eficiência da interface gráfica com o usuário (GUI), detecção automática de configuração, se verifica a validade de uma combinação de escolhas de configuração customizada, se módulos podem ser acrescidos, removidos e configurados sem grandes transtornos e se o sistema operacional pode ser instalado em diferentes plataformas de hardware.
- **Arquitetura de sistema operacional em tempo real** (*RTOS Architecture*): considera os métodos de manipulação de tarefas e interrupções, de gerenciamento de memória, se o sistema suporta *threads* e processos, se pode ser distribuído entre múltiplos nós, se o *kernel* é uma estrutura flexível ou monolítica, a possibilidade de travamento do processador, a presença de mecanismo preventivo de inversão de prioridade e mecanismo protetor de memória virtual, além de domínios de proteção.

- **Recursos de API** (*API Richness*): Riqueza de rotinas, protocolos e ferramentas para construir software de aplicação. Um bom API (*Application Program Interface*) facilita o desenvolvimento de um programa de tempo real providenciando, por exemplo, funções que auxiliam no gerenciamento de memória, temporizadores, semáforos, manipulação de interrupções e gerenciamento de tarefas.

- **Suporte para Internet** (*Internet support*): avalia as ferramentas e produtos disponíveis para dar suporte a páginas HTML dinâmicas, *javascript*, *cookies*, etc. Também verifica a possibilidade de construir aplicações de Internet em sistemas embarcados e a presença de suporte para comunicação através da internet ou intranet.

- **Ferramentas** (*Tools*): considera se as ferramentas presentes satisfazem todas as necessidades do programador e se são intuitivas no uso, avaliando também o nível de exigência do processador e a memória RAM necessária para executar as ferramentas numa velocidade razoável. Por fim, detecta a presença de *bugs*.

- **Documentação e suporte** (*Documentation and Support*): avalia se a documentação oferece uma boa visão geral do sistema e de sua arquitetura, se os parâmetros de API são bem explicados, como os códigos fontes são comentados e a facilidade em se encontrar exatamente o que se procura.

- **Resultados dos testes** (*Test results*): se o comportamento do sistema é rápido, por exemplo, capaz de manter-se dentro dos limites de tempo, e previsível, por exemplo, se todas as chamadas de funções funcionam como esperado (o que não ocorre em diversos casos). Analisa a desempenho diante de testes de *stress*, detectando vazamentos de memória, degradação do desempenho quando o sistema está carregado e latência nas interrupções.

Tabela 4-1: QNX Neutrino RTOS 6.2 – Média 8,0 (DEDICATED SYSTEMS EXPERTS, 2006).

Installation and Configuration	0	8	10
RTOS Architecture	0	9	10
API Richness	0	7	10
Internet support	0	8	10
Tools	0	8	10
Documentation and Support	0	7	10
Test Results	0	9	10

Tabela 4-2: Red Hat ELDS v1.1 – Média 4,4 (DEDICATED SYSTEMS EXPERTS, 2006).

Installation and Configuration	0	3	10
RTOS Architecture	0	3	10
API Richness	0	5	10
Internet support	0	8	10
Tools	0	8	10
Documentation and Support	0	2	10
Test Results	0	2	10

Tabela 4-3: Vx Works – Média 6,4 (DEDICATED SYSTEMS EXPERTS, 2006).

Installation and Configuration	0	4	10
RTOS Architecture	0	7	10
API Richness	0	8	10
Internet support	0	9	10
Tools	0	8	10
Documentation and Support	0	4	10
Test Results	0	5	10

Tabela 4-4: Windows CE .NET – Média 6,9 (DEDICATED SYSTEMS EXPERTS, 2006).

Installation and Configuration	0	5	10
RTOS Architecture	0	7	10
API Richness	0	7	10
Internet support	0	9	10
Tools	0	8	10
Documentation and Support	0	5	10
Test Results	0	7	10

As médias dos sistemas operacionais (Figura 4-7) fornecem um direcionamento para a escolha do sistema operacional a ser aplicado no projeto.

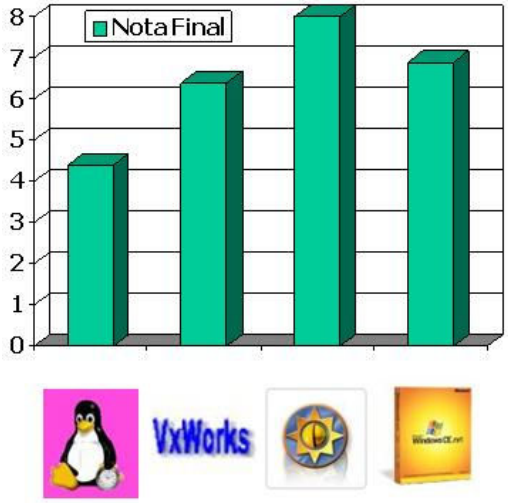

Figura 4-7: Classificação dos sistemas operacionais segundo Dedicated Systems Experts (2006).

Uma outra análise comparativa que engloba mais sistemas operacionais, mas que analisa menos características, foi realizada por Melanson (2003). Na Figura 4-8 pode-se observar os sistemas operacionais analisados, na Figura 4-9 pode-se verificar as características analisadas bem como as respectivas ponderações e na Figura 4-10 pode-se observar a classificação dos RTOS segundo Melanson (2003).

RTOS Name	Vendor
AMX™	Kadak
C Executive™	JMI Software Systems
CMX™	CMX Systems
Cortex™	Artesys
Delta OS™	CoreTek Systems
E COS™	Cygnus Solutions (Red Hat)
Emb OS™	Segger Microcontroller Systeme GmbH
Integrity™	Green Hills Software
Lynx OS™	Lynx Real time systems
Nucleus Plus™	Accelerated Technology Incorporated
OS-9™	Microware system corp
OSE™	OSE (Enea)
Precise/MQX™	Precise Software
QNX/Neutrino™	QNX
RTEMS™	On-Line Applications Research Corporation
SuperTask™	US software
TTPos™	TTTech
UC/OS-II™	Micrium, Inc
VRTX™	Mentor Graphics
Vx Works™	Wind River systems

Figura 4-8: Sistemas operacionais analisados comparativamente (MELANSON, 2003).

Category Name	W_i	Criterion	W_j	$W_i \times W_j$
Kernel	13 %	Architecture	35 %	5 %
		Multi-process support	15 %	2 %
		Multi-processor support	25 %	3 %
		Fault tolerance	25 %	3 %
Scheduling	20 %	Algorithm	40 %	8 %
		Priority assignment mechanism	20 %	4 %
		Time to release task, independent from list length*	40 %	8 %
Process/Thread/Task model	12 %	Number of priority levels	26 %	3.12 %
		Priority inversion protection	18 %	2.16 %
		Task States	10 %	1.2 %
		Max number of tasks	18 %	2.16 %
		Task switching latency	18 %	2.16 %
		Dynamic priority changing	10 %	1.2 %
Memory	10 %	Min and max RAM space per task	20 %	2 %
		Min and max ROM space	20 %	2 %
		Max addressable memory space per task	20 %	2 %
		Memory protection support	20 %	2 %
		Dynamic allocation support	10 %	1 %
		Virtual memory support	8 %	0.08 %
		Memory compaction	2 %	0.2 %
Interrupt and Exception Handling	8 %	Preemptable ISRs	30 %	2.4 %
		Worst case interrupt handling time	30 %	2.4 %
		ISR model or levels	20 %	1.6 %
		Modifiability of interrupt vector table	20 %	1.6 %
Application Programming Interface	7 %	Library compliance	4 %	0.28 %
		Precise absolute clock	10 %	0.7 %
		External clock support	10 %	0.7 %
		Synchronization and exclusion primitives	18 %	1.26 %
		Communication and message passing	18 %	1.26 %
		Network protocols	10 %	0.7 %
		Certifications	10 %	0.7 %
		I/O support	10 %	0.7 %
		File systems	10 %	0.7 %
Development Information	15 %	Development methodology	20 %	3 %
		RTOS supplied as source or object code	15 %	2.25 %
		Supported compiler	20 %	3 %
		Supported processors	30 %	4.5 %
		Supported development languages	15 %	2.25 %
Commercial Information	15 %	Cost	30 %	4.5 %
		Royalty fees	10 %	1.5 %
		Years on market (i.e. product maturity)	20 %	3 %
		Used in time critical applications	20 %	3 %
		Support type and cost	20 %	3 %

Figura 4-9: Características dos sistemas operacionais analisadas (MELANSON, 2003).

QNX/Neutrino™	1st
OS-9™	
Precise/MQX™	2nd
OSE™	
Delta OS™	3rd
RTEMS™	
Lynx OS™	
Integrity™	
VxWorks™	
Nucleus Plus™	4th
VRTX™	
TTPos™	
C Executive™	
Emb OS™	
CMX™	
ECOS™	5th
uC/OS-II™	
SuperTask™	
AMX™	
Cortex™	6th

Figura 4-10: Classificação dos sistemas operacionais segundo Melanson (2003).

Considerando-se as características detalhadas anteriormente, as comparações feitas, pensando-se na arquitetura de hardware escolhida e a disponibilidade de licença do software, decidiu-se pelo sistema operacional QNX Neutrino que teve ótimo desempenho em todas as análises, idealizado, desde seu início, para aplicações de tempo real.

4.3.3 QNX Neutrino

O QNX foi criado em 1980 e desde então tem sido utilizado como RTOS de diversas aplicações de missão crítica, desde instrumentos médicos e roteadores internet a dispositivos de telemática, 911 *call centers*, controle de processo, sistemas de controle de tráfego aéreo (QNX, 1998; KRTEN, 1998; QNX, 2004; QNX NEUTRINO, 2007).

O RTOS QNX Neutrino além de gozar de confiabilidade, tolerância à falhas e escalabilidade, possui as seguintes características:

- Confiabilidade *Military-grade* (arquitetura Microkernel, proteção total da memória);
- Projeto baseado no padrão POSIX;
- Escalabilidade massiva (processamento distribuído de forma transparente e multiprocessamento simétrico);
- Plataforma com ferramentas abertas (IDE baseada no Eclipse, cobertura de código, ferramentas de analise de sistema e aplicações);
- Gerenciamento de potência (aceita toda a gama de processadores de baixa potência);

E, a principal característica para este projeto:

- Certificação de padrões militares (**DO-178B**, MIL-STD 1553, QNX Neutrino RTOS: #0033857, *cage code*: 3AD83) .

O QNX provê recursos de multiprocessamento, escalonamento baseado em prioridades e rápido chaveamento entre processos. Em resumo, pontos considerados como positivos em relação ao sistema operacional QNX são rápido desempenho, excelente arquitetura para um sistema robusto e distribuído e bom suporte de plataforma.

Outra característica útil do QNX é a flexibilidade, possibilitando ao usuário montar um sistema operacional personalizado, disponibilizando recursos de forma racional. A modularidade do sistema operacional QNX permite que o desenvolvedor elimine processos desnecessários do sistema ou escreva novos programas para prover novos serviços.

O QNX consiste de um pequeno núcleo (Microkernel) ligado a um grupo de processos atuando em cooperação (Figura 4-11). O Microkernel é dedicado apenas às funções essenciais: *threads*, sinais, troca de mensagens, sincronização, escalonamento, *timers*, canais, conexões e interrupções.

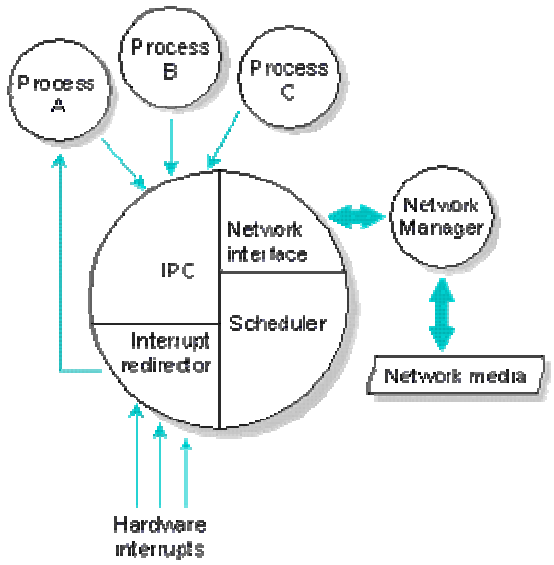

Figura 4-11: Arquitetura do Microkernel do QNX.

A área vermelha na Figura 4-12 representa pelo Microkernel e o *Process Manager* revela os únicos componentes do sistema que são confiáveis.

O *Process Manager* provê funcionalidades mais longas e complexas, como: *pathname*, processos, memória virtual, debug, recursos, inicializador e semáforos nomeados.

A área azul representada pelo sistema de arquivos, rede, janelas, multimídia e aplicação, revela os módulos não confiáveis que se conectam ao barramento de mensagens, residem em uma área de memória protegida, tem interfaces de mensagens bem definidas, não podem corromper outros componentes e podem ser inicializados, parados e atualizados sem necessidade de parar o restante do sistema.

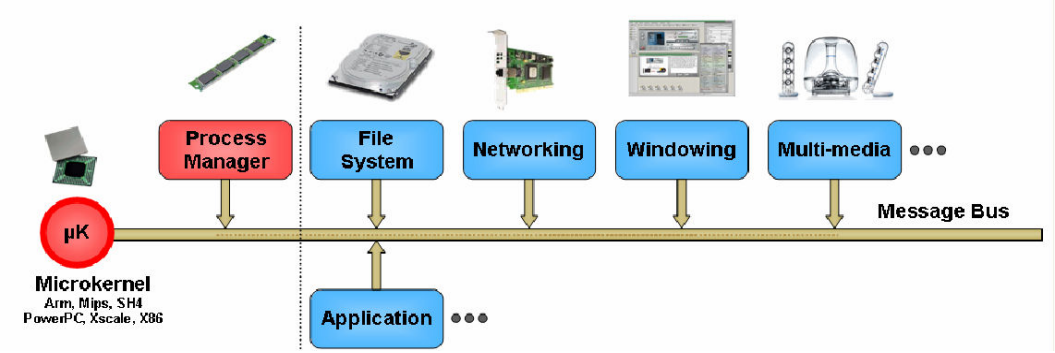

Figura 4-12: Arquitetura do Microkernel e a interação entre os demais módulos do sistema.

4.3.4 QNX Momentics

Conforme já mencionado, o QNX Neutrino é um sistema operacional que roda em um hardware embarcado, geralmente de memória de processamento relativamente reduzidos e muitas vezes desprovido meios de acesso como vídeo, teclado ou mouse, impossibilitando o desenvolvimento da aplicação no computador embarcado. Para solucionar tal problema, utiliza-se uma IDE como o QNX Momentics (QNX MOMENTICS, 2007) que se conecta remotamente ao computador embarcado permitindo a realização das seguintes tarefas essenciais para o desenvolvimento de um RTS:

- IDE baseada no eclipse CDT, permitindo a edição de código com muitas ferramentas auxiliares de programação, comuns em linguagens de programação orientada a objetos;
- *Debug* multiprocessado, multiprocesso e *multithread*;
- Perfil do sistema: informações como execução dos eventos e trocas de contexto na linha do tempo, interrupções, consumo de processamento, etc.;
- Análise de memória: permite identificar erros em regiões de memória, otimizar uso de memória, bem como ilustrar graficamente alocações requisitadas e atendidas;
- Perfil da aplicação: gráfico de chamadas, consumo de processamento por linha de código para identificação de gargalos de desempenho;
- Cobertura de código: mapeamento de linhas de código executadas durante testes;
- Identificação de *deadlocks*;
- Gerenciamento de instalação e configuração para diversos hardwares-alvo; e,
- Kit de desenvolvimento de *drivers* de periféricos.

4.4 Ferramentas de Aplicação (Application)

No capítulo 6 é relatada a arquitetura do software aviônico, e esta arquitetura influencia diretamente nos requisitos das ferramentas de desenvolvimento de aplicação. Neste observa-se que a arquitetura de software híbrida proposta será desenvolvida segundo um paradigma de agentes (WOOLDRIDGE, 1995) que requisita orientação a objetos comportamentais.

4.4.1 Linguagem de Programação

Segundo HighRely (2007) das linguagens de programação de tempo real (Burns, 2001) as mais eficientes para certificação segundo a DO-178B são C, C++ e ADA. Para atender o requisito de orientação a objeto pode-se utilizar C++ ou ADA. Ainda, HighRely (2007) recomenda a utilização de C++ devido à alta difusão desta linguagem no meio de programação, permitindo encontrar maior quantidade tanto de programadores (principalmente para suporte) quanto de bibliotecas e *frameworks*.

4.4.2 Especificação da aplicação

O desenvolvimento de sistemas *Hard Real Time* ou sistemas críticos exige sua especificação para garantir consistência (ausência de ambigüidades), completude (atende aos requisitos), corretude (ausência de *deadlocks* e *livelocks*) e previsibilidade (determinação de limites superior e inferior para tempos de execução).

Métodos de especificação formal de software utilizam formalismos matemáticos bem fundamentados como teoria de conjuntos, lógica de predicados, autômatos infinitos, etc., permitindo dedução e prova do formalismo e, em conseqüência, dedução e prova do funcionamento do sistema especificado formalmente. Neste contexto, a grande vantagem da especificação formal seria a redução de testes para a validação do sistema.

Em sistemas críticos destaca-se a especificação formal utilizando a linguagem Z e álgebra de processos CSP. A linguagem Z descreve de maneira precisa e correta as propriedades que um sistema deve ter sem evidenciar como deve ser feito. Já a álgebra de processos CSP possibilita a visualização da dinâmica e concorrência de um sistema computacional bem como da comunicação entre processos através de canais.

Embora as vantagens da especificação formal em sistemas críticos, sua utilização em sistemas aviônicos revela alguns inconvenientes. O complexo formalismo matemático e de difícil aprendizado cria barreiras à comunicação em equipes multidisciplinares que necessitam de métodos de especificação intuitivos. Este fator é um grande responsável pela expansão do UML em sistemas críticos. Por outro lado observam-se propostas promissoras como a de Polido (2007), que propõe métodos de mapeamento do UML-RT para o CSP-OZ de forma a validar os modelos UML-RT (de especificação não formal) utilizando-se o CSP-OZ (de especificação formal). Nesta proposta o CSP-OZ complementaria as carências de formalismo do UML reduzindo inconsistências e número de testes de validação necessários.

A UML é uma linguagem de modelagem orientada a objetos de finalidade geral que provê uma extensa base conceitual para um amplo espectro de aplicações, espectro este que abrange inclusive os sistemas de tempo real. Em sua versão 1.1, versão que foi adotada como padrão pela OMG, a UML apresentava extensões destinadas a aplicações em áreas específicas de conhecimento, dentre elas sistemas de tempo real. Tais extensões surgiram em grande parte de contribuições advindas de métodos previamente existentes destinados à modelagem orientada a objetos em campos específicos. No caso de sistemas de tempo real, por exemplo, cita-se o ROOM (SELIC, 1994) e o Octopus (AWARD, 1996). Entretanto Selic (1994), criador do ROOM, comenta que, embora existam várias extensões chamadas de tempo real que acrescentaram novos conceitos à UML, a experiência tem provado que isto não é necessário. A modelagem de tempo real pode ser inteiramente realizada através da especialização apropriada da base conceitual já existente na UML (Selic, 1999). Não existem diagramas especiais dentro da UML para expressar características de tempo real em um sistema, ao invés disso, informações de modelagem de tempo-real podem ser inseridas a UML, especialmente em diagramas comportamentais.

Já no que concerne a orientação a agentes, no passado, as pesquisas dentro do campo da engenharia de software orientado a agentes não obtiveram grande aceitação pela indústria de desenvolvimento de software. O grande motivo, segundo Bauer (2001), foi a falta de tato com que foi tratada a questão por parte dos pesquisadores dentro do meio acadêmico. Buscando mudar tal situação, formalizou-se uma cooperação entre a FIPA e a OMG, que permitiu o estabelecimento de relações entre a tecnologia de agentes e padrões do desenvolvimento de software orientado a objetos. Como primeiro resultado desta cooperação foi apresentado uma proposta de extensão UML: a AUML (Odell, 2000), que atualmente encontra-se em fase de discussão junto com um dos grupos de trabalho da OMG, o APSIG. Defensores da AUML alegam que a UML ainda não provê base conceitual suficiente para modelagem de agentes e sistemas baseados em agentes. Esta afirmação, segundo Odell (2000), é basicamente por duas razões: primeiramente porque, comparados a objetos, agentes possuem mais fortemente o conceito de autonomia, pois podem tomar a iniciativa e ter controle sobre como são processadas as requisições externas, em segundo porque, agentes não agem apenas isoladamente, mas em cooperação ou coordenação com outros agentes. Em virtude de não haver de fato uma extensão UML para a modelagem de sistemas multiagentes e sim propostas em discussão junto aos organismos responsáveis pela padronização, optou-se pela utilização das idéias básicas contidas nos artigos que propõem a AUML. De forma geral, tais artigos (ODELL, 2000; BAUER, 2001) apresentam o conceito de agente como uma extensão de objetos ativos exibindo tanto autonomia dinâmica (a habilidade de iniciar uma ação sem requisições externas) quanto determinística (a habilidade de recusar ou modificar uma requisição externa). E é neste sentido que a noção de agente será utilizada na modelagem da arquitetura de software proposta no capítulo 6.

Finalmente, para ter-se a especificação completa do sistema deve-se especificar o hardware. Assim deve-se estabelecer uma metodologia de modelagem que integre o hardware à AUML-RT.

A OMG SysML é uma linguagem de modelagem gráfica para especificar, analisar e projetar sistemas complexos que incluem hardware, software, informação, interface homem máquina e processos. Em particular, a linguagem provê representações gráficas com fundamentação semântica para a modelagem de requisitos, comportamentos, estruturas e integração. SysML representa um subconjunto da UML 2.0 com extensões necessárias para satisfazer os requisitos da UML para sistemas de engenharia como pode ser observado na Figura 4-13. O SysML usa XMI para trocar dados de modelagem entre as ferramentas e é também compatível com a ISO 10303-233 (ISSO, 2008) que é a normalização ISO para o intercâmbio de dados em sistemas de engenharia.

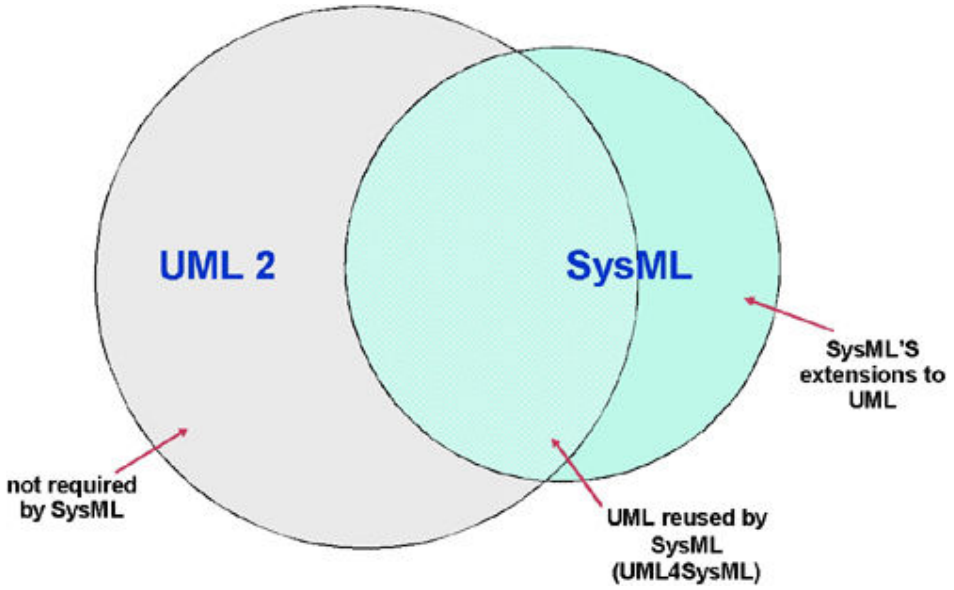

Figura 4-13: Relação entre SysML e UML.

4.4.3 Seleção da ferramenta de desenvolvimento da aplicação

Considerando os requisitos estabelecidos nas seções anteriores, nesta seção será definida a ferramenta de modelagem tanto de hardware e software de aplicação.

Entre as ferramentas que implementam SysML, AUML-RT e SysML DO-178B, destacam-se o Telelogic Rhapsody e o Rational Rose/RT.

Segundo Bichler (2002) ambas as ferramentas implementam UML e algumas ferramentas adicionais com o objetivo de melhorar a modelagem de sistemas de tempo real. Enquanto Rhapsody implementa uma variante dos *Statecharts* do UML, Rose/RT mescla UML com ROOM. Na Tabela 4-5 pode-se observar uma comparação entre o Rhapsody e Rose/RT (BICHLER, 2002). Podem-se constatar as principais características do Rhapsody e Rode/RT:

- A linguagem de modelagem do Rhapsody é mais fácil para trabalhar principalmente porque não contém nenhum diagrama adicional ao UML. Enquanto que Rose/RT adiciona os diagramas de estrutura orientada a componentes;
- Rose/RT se adequa à modelagem orientada a componentes, porque herda características do ROOM. Já o Rhapsody se adequa à modelagem orientada a objetos porque classes referenciam diretamente outras classes;
- O Rhapsody apresenta vantagens em aspectos de cálculos dirigidos por tempo e dados, porque contém os elementos correspondentes dos statecharts do UML, enquanto que o Rose/RT não implementa tais elementos;
- Ambos têm deficiência de ferramentas de teste e verificação.

Tabela 4-5: Comparação entre Rhapsody e Rose/RT (BICHLER, 2002).

Property	Rhapsody®	Rose/RT®
ease of use	+	0
standard conform	+	0
object-oriented	+	+
component-oriented	-	+
asynchronous signals	+	+
synchronous operations	+	0
event-driven	+	+
time-driven	0/+	0
data-driven	-	-
simulation	+	+
test definition	-	-
verification	-	-

Conforme dito anteriormente, Selic (1994), criador do ROOM, comenta que, embora existam várias extensões chamadas de tempo real que acrescentaram novos conceitos à UML, a experiência tem provado que isto não é necessário. Assim as novas ferramentas do Rose/RT podem facilitar a modularização, mas não são efetivamente necessárias.

Outro ponto a considerar é a conformidade com padrões. O Rhapsody atende completamente ao padrão UML, UML-RT, AUML e SysML, enquanto que Rose/RT não atende a todos esses padrões. Assim, o Rhapsody garante melhor adaptação de uma futura ferramenta de especificação formal como CSP-OZ.

No caso do Rhapsody, a Telelogic fornece uma ferramenta adicional, o Test Conductor, para suprir a deficiência de definição e verificação de testes segundo a DO-178B.

4.4.4 Telelogic Rhapsody

O Rhapsody (DOUGLASS, 1999; DOUGLASS, 2007) é uma ferramenta de desenvolvimento dirigido por requisitos e modelos (RDD & MDD) cujo foco está na automação do projeto de software, pois cria sistemas executáveis a partir de modelos UML. O princípio desta automação baseia-se na geração de código a partir de modelos UML através de uma *framework* de tempo real. Esta *framework* também pode opcionalmente instrumentar o código, permitindo a realização de *debug* de modelo remotamente. Todo este processo de desenvolvimento e *debug* via modelos reduz o tempo dos ciclos de desenvolvimento e melhora a qualidade do sistema. Também permite que o sistema seja continuamente testado durante o processo de desenvolvimento, em vez de testá-lo no fim, o que garante sua validação desde os estágios iniciais até o fim do projeto, o que permite identificar os erros a tempo de modificação, sem grandes impactos.

As principais características do Rhapsody são:

- Tratamento de requisitos: modelagem, mapeamento e análise;
- Desenvolvimento de Sistemas e Software completamente portáveis que suportam SysML 1.0, UML 2.1 e XMI.
- Suporta vários RTOS: QNX Neutrino, VxWorks, Integrity, RT-Linux, entre outros;
- Geração automática de código baseado em regras em C, C++, Java e Ada, onde alterações no modelo UML são convertidas automaticamente em alterações no código bem como alterações no código são convertidos automaticamente em modelos UML, garantido que o código esteja sempre sincronizado com a especificação e por sua vez com a documentação reduzindo erros potenciais;
- Integração com o QNX Momentics;
- *Framework* de tempo real, que já implementa diversos recursos fundamentais na programação de sistemas concorrentes, como troca de mensagens entre processos e *threads*, reatividade de processos, *timeouts*, etc.; além de oferecer a portabilidade do código gerado para diversos sistemas operacionais, como QNX Neutrino, VxWorks, Linux, entre outros;
- Estende do tradicional MDD e inclui a geração de testes dirigida por modelos. Cenários criados na fase de análise, descrevendo o comportamento do sistema, são reusados durante o teste de integração para garantir que o sistema atende a estes cenários. Isto é conhecido como teste baseado em requisitos e é implementado pelo Test Conductor. Por outro lado o Automatic Test Generator gera vetores de testes para todo o sistema, para garantir a completa cobertura em testes unitários, de

integração e sistema. Segundo a empresa Telelogic, esta combinação cria um novo paradigma no desenvolvimento de sistemas embarcados chamado *Design For Testability*, DFT: observação remota da aplicação com *debug* de modelo, testes baseado em requisitos e geração automática de testes.

No que concerne à normalização, o Rhapsody produz código homologável via DO178-B e ED-12B. A habilidade de capturar e gerar documentação de requisitos, projeto, implementação e testes tudo dentro do Rhapsody, permite ao usuário seguir um processo de desenvolvimento bem definido e facilmente repetido. Estes últimos são conceitos chave para a homologação via DO-178B.

Um conceito importante da DO-178B é que todos os requisitos são cobertos pelo código e todo o código que existe é devido diretamente a um requisito. Ou seja, não existe requisito bem atendido com código morto.

Um resumo do princípio de funcionamento do Rhapsody pode ser observado na Figura 4-14. No computador de desenvolvimento, o engenheiro insere modelos UML através dos diagramas de edição AUML-RT/SysML. Todas as informações sobre a aplicação (inclui modelos, códigos e outras propriedades) em desenvolvimento são armazenadas no repositório de modelo de aplicação. O engenheiro também define algumas configurações como: linguagem de programação, sistema operacional no qual será embarcada a aplicação, instrumentação ou não de aplicação ou código. Com o modelo desenhado, o gerador de código gera o código referente ao modelo utilizando a OXF, bem como mantém a sincronização entre o código e o modelo de forma de mudança no código reflitam no modelo e vice-versa. Em seguida o código é compilado, lincado, carregado e executado no computador embarcado.

Caso a animação esteja habilitada, o código gerado está instrumentado e cada evento que ocorre no computador embarcado é enviado via TCP-IP ao computador de desenvolvimento. O Rhapsody então mostra de forma gráfica todos os eventos que ocorrem no computador embarcado. A ferramenta de animação do Rhapsody também permite o envio de estímulos à aplicação embarcada.

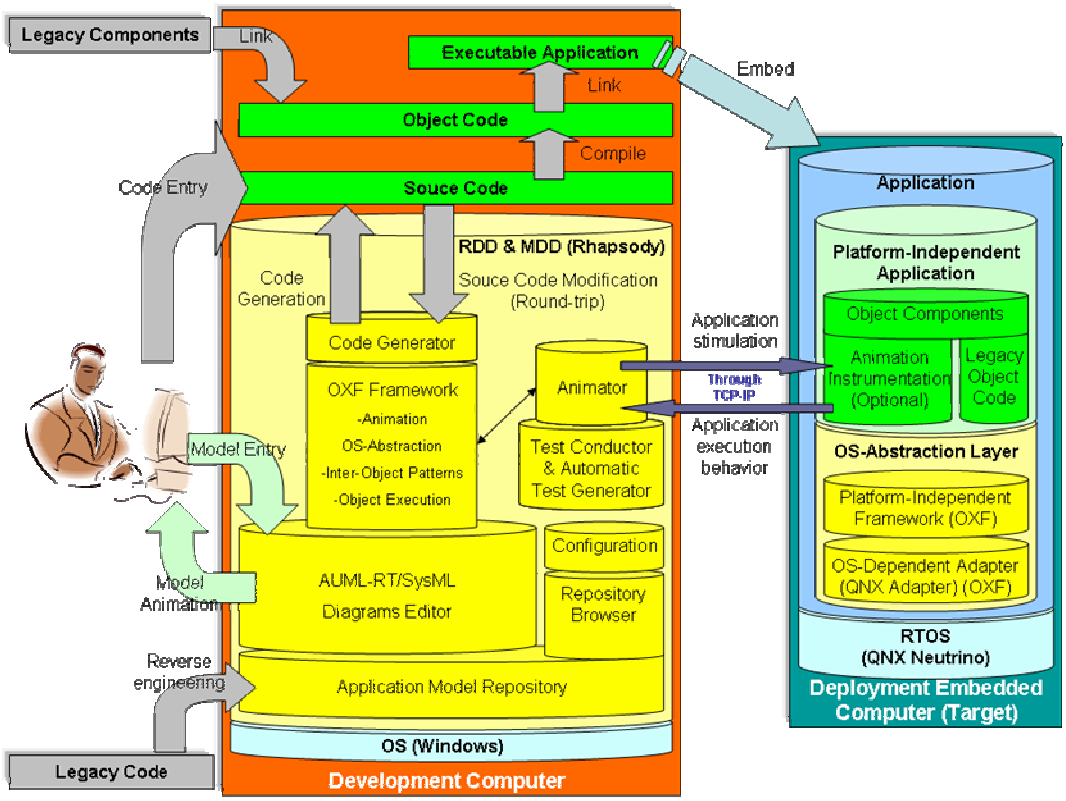

Figura 4-14: Esquema funcional do Rhapsody.

4.4.5 OXF Real-Time Framework

Parte considerável do código de infra-estrutura de aplicações é o mesmo, mais especificamente: os meios de iniciar e finalizar tarefas, escalonamento de tarefas e recursos, métodos para implementar e executar máquina de estados, bem como caminhos de implementar associações. Todos estes serviços não são específicos a uma aplicação. Segundo Douglass (1999), entre 60% e 90% de uma aplicação é código comum que pode ser reusado se estruturado apropriadamente.

O método tradicional de desenvolver sistemas consiste em escrever diversas vezes trechos de códigos semelhantes ou iguais. Uma proposta seria capturar serviços comuns que todas as aplicações de um domínio necessitam de forma que estas possam ser reusadas, especializadas se apropriado e substituídas se necessário. Estes serviços são geralmente um grupo de padrões de projeto que realizam o suporte da aplicação, e neste contexto, esta infra-estrutura é chamada de *framework*.

O uso de *frameworks* apropriadas pode facilitar o desenvolvimento de uma aplicação porque serviços comuns não precisam ser reimplementados. A *framework* deve prover um grupo de serviços apropriados para um dado domínio de aplicações e deve também trabalhar dentro do ambiente de execução do domínio, incluindo protocolos de comunicações e sistemas operacionais. *Frameworks* de domínio específico que atendem a estes critérios resultam em aplicações que são mais robustas e de rápido ciclo de desenvolvimento.

Uma *framework* do domínio de tempo real é uma *framework* vertical, otimizada para aplicações de tempo real. Usualmente provêem padrões de: suporte a arquitetura, colaboração e distribuição, confiabilidade e segurança, e, comportamento com objetivos de previsibilidade, velocidade e tamanho.

A ferramenta de automação de projeto do Rhapsody gera código a partir da *Object Execution Framework* (OXF), uma *framework* do domínio de tempo real. A OXF consiste de quatro partes: *Execution Framework; Inter-Object Association Paterns, Abstract Operationg System* e *Animation Framewrok*.

- *Execution Framework:* Provê a infra-estrutura para a execução do modelo UML a partir da metaclasses expostas na Tabela 4-6.

Tabela 4-6: Metaclasses da Execution Framework.

Metaclasses	Descrição
OMReactive	Executa máquina de estados, gerencia eventos e atividades relacionadas.
OMThread	Cada *thread* tem uma fila de mensagens que guarda eventos para objetos executando dentro da *thread*.
OMEvent	Esta é a classe base de todos os eventos no Rhapsody. Classes reativas comunicam-se enviando eventos deste tipo.
OMTimeout	Realiza todos os *timeouts* nos statecharts.
OMTimerManager	Gerencia todos os *timeouts* pendentes.

- *Inter-Object Association Paterns:* O Rhapsody resolve todas as associações via ponteiros. Quando a multiplicidade é 0 ou 1, um ponteiro simples é utilizado no objeto associado à classe. Entretanto quando a multiplicidade é maior que 1 o Rhapsody cria uma classe *container* para manipular associações múltiplas. Na Tabela 4-7 pode-se observar a metaclasse de associação OMContainer.

Tabela 4-7: Metaclasses da Inter-Object Association.

Metaclasses	Descrição			
OMContainer	Multiplicidade da associação	*Ordered Property*	*Container Type*	Implementação
	0..1 ou 1	Ignorado	*Scalar*	Ponteiro
	m..n, n, or *	Falso	*Unordered*	OMCollection
	m..n, n, or *	Verdadeiro	*Ordered*	OMList
	Qualified	Ignorado	*Qualified*	OMMap
	* onde m e n são escalares fixos.			

- *Abstract Operationg System:* o objetivo de abstrair o sistema operacional das outras partes da *framework* é garantir a portabilidade da aplicação em diversos sistemas operacionais. Para mover do QNX Neutrino para o VxWorks, por exemplo, nada mais será necessário do que reimplementar do nível de abstração de sistema operacional, mais especificamente as metaclasses listadas na Tabela 4-8.

Tabela 4-8: Metacleasses da Abstract Operating System.

Metaclasses	Operações	Descrição
OMOSFactory	Classe *singleton* que é uma instância padrão de fábrica.	
	createOMOSMessageQueue	Cria uma fila de mensagens
	createOMOSEventFlag	Cria um *flag* de evento.
	createOMOSThread	Cria uma *thread*.
	createOMOSWrapperThread	Cria uma *thread* abstrata.
	createOMOSMutex	Cria uma exclusão mútua.
	createOMOSTickTimer	Cria um *clock* de *timer*.
	createOMOSIdleTimer	Cria um *timer* ocioso.
OMOSThread	*Thread* concreta alocada pela classe OMOSFactory.	
	suspend	Suspende a execução da *Thread*.
	resume	Resume a execução da *Thread*.
	Start	Começa a execução da *Thread*.
	getOSHandle	Chama operações específicas do sistema operacional.
	setPriority	Define a prioridade da *Thread*.
OMOSMutex	Exclusão mútua concreta alocada pela classe OMOSFactory.	
	lock	Trava a *mutex*.
	free	Livra a *mutex*.
OMOSMessageQueue	Fila de mensagens concreta alocada pela classe OMOSFactory.	
	put	Insere uma mensagem na fila.
	pend	Espera por mensagem.
	isEmpty	Verifica se a fila está vazia.
	get	Obtem uma mensagem da fila.
OMOSEventFlag	Classes de eventos concretas para sinalizar *threads* da disponibilidade de eventos.	
	signal	Sinaliza um evento.
	reset	Limpa um evento.
	wait	Espera um evento.

- *Animation Framework:* A *framework* de animação gera o suporte para animação do Rhapsody. O Rhapsody utiliza esta *framework* para a inserção da instrumentação nas linhas de código da aplicação, que se comunica com o computador de desenvolvimento via um *socket* TCP-IP. Assim o sistema operacional do computador embarcado deve prover comunicações TCP-IP e interface *socket*.

O lado bom de usar instrumentação para retornar o estado de execução para a ferramenta de animação é que o Rhapsody pode animar o modelo graficamente conforme a aplicação executa. Por outro lado, o código da instrumentação é grande e lento, em relação ao código da aplicação. Isto torna a aplicação com instrumentação inapropriada para análise de desempenho, entretanto adequada para a análise funcional. Na Tabela 4-9 pode-se observar um teste realizado na configuração de aplicação do item 8.2. Nela observa-se que o código instrumentado neste caso consumiu mais que quatro vezes o processamento.

Tabela 4-9: Custo de processamento em diversas condições de instrumentação.

Code	*Model*	*Instrumentation*	*Processing Cost*
Release	*Release*	*None*	17,5%
Debug	*Release*	*Code*	22,3%
Debug	*Debug*	*Code & Model*	94,0%

4.5 Ferramenta de Cálculo Matemático (Mathematics)

Uma das características mais singulares de sistemas embarcados é sua intensa interação com o ambiente físico. Particularmente no sistema aviônico em questão, o sistema embarcado deve “sentir” o ambiente, “pensar” e “agir” no ambiente gerando a autonomia da aeronave. Cada um destes três processos envolve cálculos matemáticos que computacionalmente são processados por algoritmos matemáticos.

O primeiro passo do projeto do sistema controle autônomo consiste em modelar matematicamente a dinâmica do ambiente, que no caso engloba a dinâmica da Terra (atmosfera, gravidade, campo magnético, solo, etc.) e do veículo (plataforma, sensores e atuadores, etc.). Então se desenvolvem os modelos matemáticos dos processos que dão autonomia ao VANT, mais especificamente modelos de: filtragem, navegação, controle, guiagem, planejamento de trajetória, anti-colisão, entre outros. Durante o projeto do sistema controle autônomo, diversas simulações são realizadas para garantir que os requisitos estão sendo atendidos. Para tanto existem disponíveis diversas ferramentas que desempenham tal função, a destacar: Matlab, Octave, Scilab, LabView, RLab, O-Matrix, Sysquake e FreeMat.

Depois de finalizado o projeto do sistema controle autônomo, deve-se implementar o algoritmo matemático correspondente ao modelo matemático desenvolvido. Neste passo reside um sério problema no projeto de sistemas embarcados que é a corretude da implementação destes modelos matemáticos. Muitos desses modelos são complexos, envolvendo centenas, ou até milhares de variáveis e, caso uma seja implementada com valor errado, o resultado por ser catastrófico, mesmo que na compilação não seja detectado nenhum erro.

Um método para suprir esta deficiência no projeto de VANTs é o desenvolvimento de simuladores de hardware (HILS) como em Shixianjun (2006). Entretanto, este método revela desvantagens, principalmente a necessidade de desenvolvimento de um outro sistema de tempo real que simule o ambiente, ou seja, dinâmica da Terra (atmosfera, gravidade, campo magnético, solo, etc.) e do veículo (plataforma, sensores e atuadores, etc.) e como a simulação do ambiente é também um modelo matemático, a corretude da implementação do respectivo algoritmo matemático também passa a ser um problema, assim como na implementação do algoritmo matemático do sistema controle autônomo.

Uma proposta para resolver isso é a utilização de geração automática do algoritmo matemático a partir do seu modelo. Neste contexto garante-se que o modelo foi corretamente implementado no código.

4.5.1 Seleção da ferramenta de análise numérica

Como dito anteriormente, existem disponíveis diversas ferramentas de análise numérica, a destacar: Matlab, Octave, Scilab, LabView, RLab, O-Matrix, Sysquake e FreeMat. Entretanto somente o Matlab atende aos seguintes requisitos:

- Geração de código no padrão ANSI C99;
- Conexão com o Rhapsody;
- Atendimento à DO-178B;

4.5.2 Matlab Simulink e Matlab Real Time WorkShop

O Matlab é uma linguagem de alto-nível e desempenho para computação técnica, o Simulink é uma ferramenta para modelar, simular e analisar sistemas dinâmicos, e o Real Time Workshop é uma ferramenta de geração automática de código. A geração automática de algoritmos matemáticos no Matlab é realizada pelo Real Time Workshop utilizando modelos matemáticos do Simulink.

O Matlab já é extensivamente utilizado no desenvolvimento de sistema de controle de vôo automático, sistema de aviso de emergência, sistema de controle *Fly-By-Wire* segundo a norma DO-178B nível A, B, C e D (POTTER, 2007) e foi verificado que a taxa de defeitos na integração dos sistemas reduziu em uma ordem de grandeza. Estes sistemas foram empregados nas seguintes aeronaves: Embraer 170, Bombardier Global Express, Gulfstream IV & V, Dassault Falcon 900 & 2000, Cessna Sovereign, Augusta Bell 139 Helicopter.

Segundo Potter (2005) o Real Time Workshop atingiu nível de qualidade Seis Sigmas. Para atingir Seis Sigmas, a taxa de defeito deve ser inferior a 3,4 defeitos por 1 milhão de oportunidades. Durante dois anos, 1,6 milhões de linhas de código foram gerados e compilados utilizando o compilador da Borland. Um defeito foi encontrado durante a revisão do código e nenhum defeito foi encontrado durante testes unitários. O erro foi encontrado no código de um chaveador multi-porta que estava em um laço de 2ms ao invés de 5ms, entretanto este código passou pela certificação da DO178B nível A.

Para verificar a eficiência do código autogerado, Hodge (2004) analisou o tamanho do espaço em memória do modelo de controle de uma *powertrain*. A Tabela 4-10 mostra o espaço em memória na ROM e RAM de um típico software de controle de *powertrain* implementado manualmente e automaticamente. Verifica-se que o software gerado automaticamente é menor que o implementado manualmente, validando a viabilidade de utilização de geração automática de código.

Tabela 4-10: Comparação entre Hand Code e Auto Code na ROM e RAM (HODGE, 2004).

	Hand Code	Auto Code
ROM	6408	6192
RAM	132	112

4.6 Considerações finais deste capítulo

Neste capítulo, foi proposta uma metodologia de desenvolvimento de sistemas aviônicos de VANTs com requisitos de homologação que será seguida em todo o desenvolvimento relatado nos próximos capítulos. Esta metodologia fundamenta-se no processo, nas normas e nas ferramentas de desenvolvimento.

O processo de desenvolvimento estabelece quais tarefas e quando estas tarefas devem ser executadas. Verificou-se que é importante que todos os requisitos e arquitetura do sistema estejam totalmente definidos e entendidos antes que qualquer projeto significante ocorra, inclusive o de software. Assim, o processo de desenvolvimento deve contemplar não somente o software (processo de engenharia de software), mas também o sistema (processo de engenharia de sistemas) e para tanto foi selecionado o processo de desenvolvimento Harmony. O Harmony prevê a captura dos requisitos e modelagem do sistema antes do início do desenvolvimento do software.

As normas de desenvolvimento foram restringidas às normas de homologação. As normas de homologação atestam que uma aeronave, desde que corretamente operada, é capaz de decolar, manter vôo e pousar de forma segura, o que significa que o risco de acidente é baixo o suficiente para que seja aceitável pela sociedade. Verificou-se que as normas de homologação de VANTs ainda encontram-se em discussão e que atualmente são aplicadas as normas aeronáuticas convencionais com as pertinentes adaptações em VANTs. Mais especificamente no desenvolvimento de hardware e software, as normas DO-254 e DO 178B, respectivamente, já são aplicadas em VANTs com sucesso e por isso foram adotadas. Verificou-se também, que a automação de tarefas repetitivas como codificação e testes é uma forma de reduzir significativamente o custo de desenvolvimento de software reduzindo-se o acréscimo de custo para atender as normas de homologação de aproximadamente 80% para 30%. Neste contexto, foi estabelecido um conjunto de ferramentas de desenvolvimento para garantir esta automação, sendo elas: - ferramentas de sistema operacional, - ferramentas de aplicação e - ferramentas de cálculo.

Verificou-se que o sistema aviônico em questão possui todas as características de um sistema de tempo real, principalmente restrições temporais bem determinadas, revelando a necessidade de utilizar um sistema operacional de tempo real. Assim o conjunto de ferramentas de sistema operacional deve incluir o pacote de suporte de placas (BSP), o RTOS e os *Drivers*. Para selecionar este conjunto de ferramentas, foram viabilizadas diversas análises comparativas entre os principais RTOS, nas quais o QNX Neutrino obteve melhor

desempenho. Assim, O QNX Neutrino será embarcado na ECU Central (PC104) e sobre ele rodará a aplicação aviônica. Em um computador de desenvolvimento do tipo Desktop, rodará a IDE QNX Momentics que se conectará remotamente à ECU Central permitindo: debug, perfil de sistema e aplicação, cobertura de código, análise de memória entre outras tarefas essenciais para o desenvolvimento de aplicações de tempo real.

Para a aplicação, foi selecionada a linguagem de programação C++ e especificação AUML-RT. Neste contexto, pôde-se selecionar a ferramenta de aplicação a partir da análise comparativa entre o Rhapsody e Rose/RT, na qual o Rhapsody apresentou melhor desempenho. No Rhapsody, será realizada, por meio da especificação SysML, a captura de requisitos, modelagem e simulação do sistema aviônico, englobando requisitos, modelos e simulações de hardware e software. Em seguida, a partir da simulação do sistema, serão capturados os requisitos de software. Por meio da especificação AUML-RT, será modelado todo o software aviônico embarcado e o Rhapsody gerará o respectivo código. O Rhapsody será executado no mesmo computador de desenvolvimento do tipo Desktop, citado anteriormente, e manterá uma conexão bidirecional com o QNX Momentics para sincronização entre código (gerado na IDE QNXMomentics ou no Rhapsody) e modelo (gerado no Rhapsody).

Dentro da aplicação, ficou evidente existência de uma categoria particular de código, os algoritmos matemáticos que no caso do projeto BR-UAV implementam a autonomia da aeronave. Verificou-se a dificuldade em garantir a corretude da implementação destes algoritmos matemáticos a partir dos modelos matemáticos e, para resolver isso, valeu-se de ferramentas de cálculo que geram automaticamente do algoritmo matemático a partir do modelo matemático. Dentre as diversas ferramentas disponíveis, somente o MatLab Simulink e MatLab RealTimeWorkshop atende a todos os requisitos. O MatLab Simulink é uma linguagem de modelagem matemática capaz de simular e analisar sistemas dinâmicos e será utilizado no passo de projeto do modelo matemático de autonomia. Já o RealTimeWorkshop é um gerador de código e será utilizado na geração automática de código a partir de modelos do Simulink. O MatLab rodará no mesmo computador de desenvolvimento do tipo Desktop citado anteriormente e manterá uma conexão unidirecional com o Rhapsody para envio do código/modelo matemático (gerado no MatLab) para o código/modelo de software (gerado no Rhapsody).

O conjunto de ferramentas de desenvolvimento operará em conjunto para auxiliar o desenvolvedor conforme a Figura 4-15.

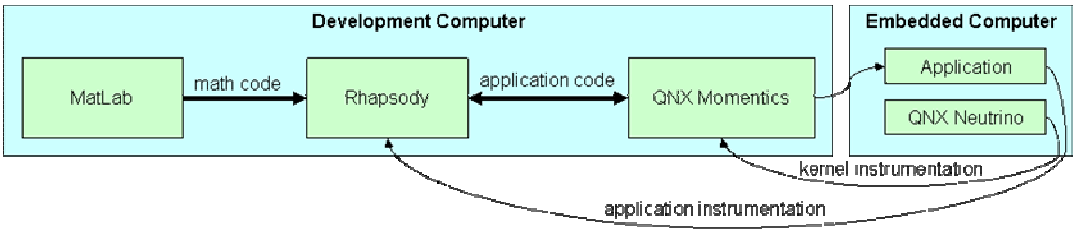

Figura 4-15: Resumo da interconexão entre ferramentas.

Neste contexto, pode-se concluir que este capítulo propôs uma metodologia que potencializa o desenvolvimento de sistemas aviônicos homologáveis e isto ficará mais evidente nos próximos capítulos.

5 ANÁLISE DO SOFTWARE AVIÔNICO EMBARCADO

A análise do software aviônico embarcado é composta por três etapas. Primeiro define-se na seção 5.1 os requisitos do sistema aviônico, ou seja, requisitos que o SANT (composto por operador, plataforma, hardware e software) impõe ao sistema aviônico (composto por hardware e software). Em seguida modela-se o sistema aviônico na seção 5.2 para então, na terceira e última etapa apresentada na seção 5.3, filtrar e refinar os requisitos do software embarcado. Como o objetivo deste trabalho é estudar o software embarcado, não foram filtrados e refinados os requisitos do software da estação base.

5.1 Análise dos Requisitos do Sistema Aviônico

Nesta seção, determinam-se os requisitos do sistema aviônico, que é composto pelo hardware e software tanto embarcado quanto da estação base. Assim devem-se considerar, para a determinação de tais requisitos, os atores que interagem com a aviônica, que no caso são: piloto, operador de *payload*, sistema de GPS e plataforma.

5.1.1 Requisitos Funcionais

Na Figura 5-1 pode-se observar o caso de uso geral da aviônica da aeronave Apoena 1. Nesta, podem-se observar os atores do sistema: o RPUAVPilot ou piloto RP-UAV, o PayloadOperator ou operador de *payload* bem como o GPS.

O RPUAVPilot tem a função de controlar e navegar a aeronave com o objetivo de percorrer uma trajetória pré-estabelecida pela missão. O ator RPUAVPilot controla e navega a aeronave por meio de sua interação com o *joystick* de controle e com a interface de controle. O *joystick* de controle interage com o RPUAVPilot permitindo a este ator o controle de *thrust* (aceleração), *elevator* (profundor), *flap* (flapes), *aileron* (ailerons), *rudder* (leme), VVFCamera (seleção de câmeras de vôo visual) e *lights* (luzes de navegação, táxi, pouso e anti-colisão). Já a interface de controle interage com o RPUAVPilot gerando interface gráfica (Figura 3-14) composta por VVFsys (imagens das câmeras de vôo visual) e IFsys (informações dos estados da aeronave).

O PayloadOperator tem a função de controlar a carga paga com o objetivo de monitorar, a partir de foto e vídeo, uma dada região estabelecida pela missão. O ator PayloadOperator controla o *payload* por meio de sua interação com uma interface de controle de *payload* e com a interface gráfica de *payload*. A interface de controle de *payload* interage com o PayloadOperator permitindo a este ator o controle de *pan-tilt*, ou seja, seus movimentos de compensação de *pitch* (arfagem) e *roll* (rolagem) bem como o controle de câmera, ou seja, suas configurações de *zoom* e *shooting* (disparo de foto). Já a interface de *payload* interage com o PayloadOperator gerando interface gráfica de auxílio ao controle de *payload* que é específica para cada aplicação.

Neste trabalho, restringe-se o projeto ao caso de uso AircraftControlandNavigation uma vez que o caso de uso PayloadControl deve ser realizado de forma personalizada para cada tipo de monitoramento, ou seja, para cada tipo de *payload*.

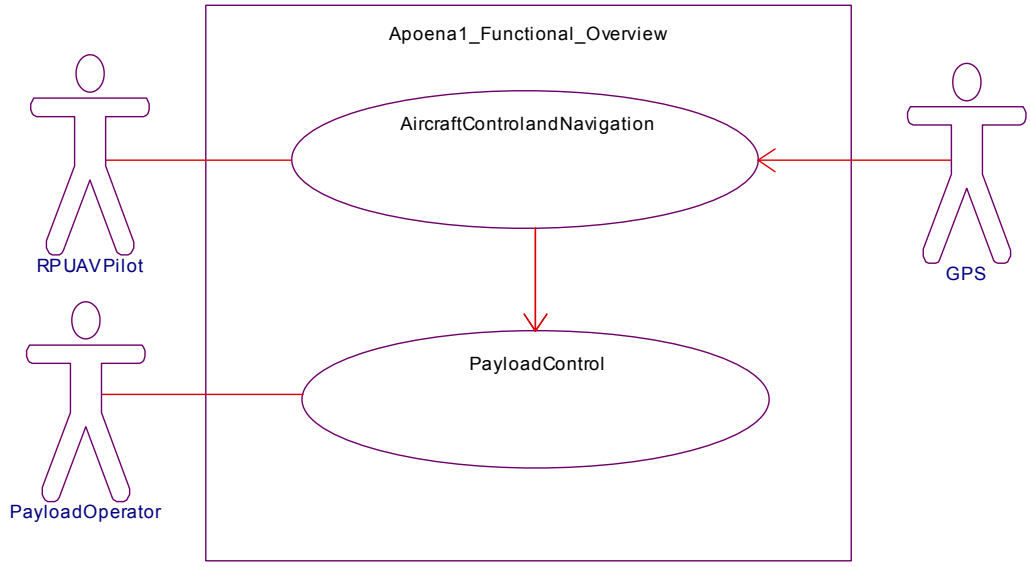

Figura 5-1: SysML Use Case - Visão geral das funcionalidades da aviônica da aeronave Apoena I SysML Use Case.

5.1.2 Requisitos Temporais

Os requisitos temporais são definidos pelos atores externos que interagem com o sistema, ou seja, dinâmicas da plataforma e piloto.

- Dinâmica da plataforma

A plataforma do VANT Apoena1 possui dinâmica com função de transferência de quarta ordem: quatro pólos longitudinais e quatro pólos laterais (AMIANTI, 2006; NELSON, 1989; ROSKAN, 1997).

Na Figura 5-2 tem-se a dinâmica longitudinal onde se observa os dois pólos complexo-conjugados chamados de pólos de fugóide (**0,394rad/sec=0,0627Hz**) caracterizados pela resposta lenta e por ser os pólos dominantes da dinâmica longitudinal de aeronaves. Observam-se também os dois pólos reais chamados de pólos de período curto (**15,8rad/sec=: 2,51Hz e 48,4rad/sec=70,0Hz**) caracterizados pela resposta rápida do sistema.

Na Figura 5-3 tem-se a dinâmica lateral onde se observa o pólo real chamado de pólo de espiral (**0,0893rad/sec=0,0142Hz**) caracterizado pela resposta lenta e por ser o pólo dominante da dinâmica lateral de aeronaves. Ainda observam-se os dois pólos complexos conjugados chamados de pólos de *dutch-roll* (**11,3rad/sec=1,80Hz**) e o pólo real chamado de pólo de rolagem (**40,8rad/sec=6,49Hz**) caracterizados pela resposta rápida do sistema.

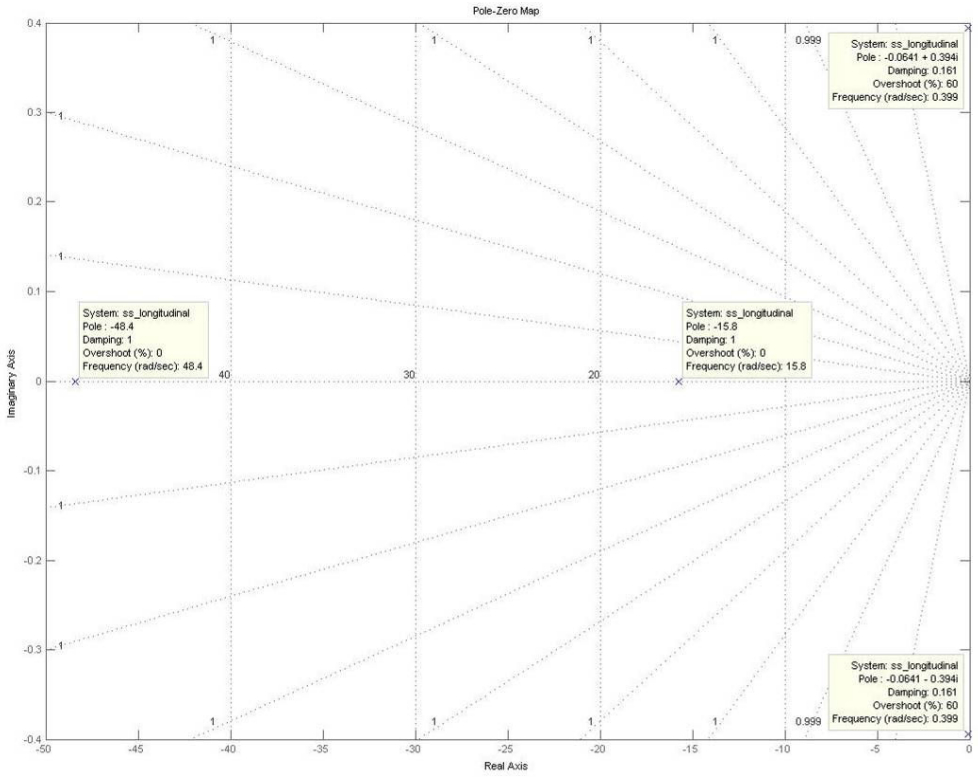

Figura 5-2: Pólos da dinâmica longitudinal da aeronave Apoena1.

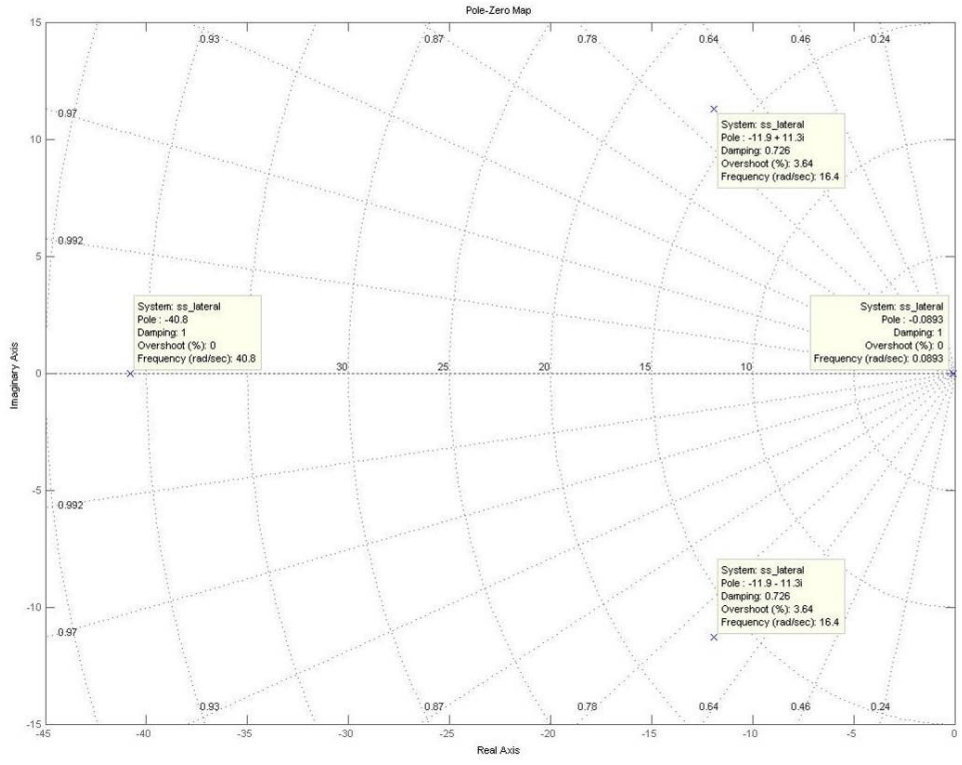

Figura 5-3: Pólos da dinâmica lateral da aeronave Apoena1.

Conforme observado, tem-se uma planta absolutamente estável, pois todos os pólos possuem parte real negativa. Já a estabilidade relativa à dinâmica desejada revela desempenho pobre uma vez que tem-se três pólos (dois de fugóide e um de espiral) com dinâmica excessivamente lenta, se considerado o caso deste UAV, cuja função objetivo de desempenho pode ser resumida em "obter uma dinâmica com o sobre-sinal inferior a 10% e tempo de assentamento da ordem de 1 a 5 segundos, dada as restrições de comando e de envelope de vôo".

De OGATA (1995) tem-se:

$$T_s = \frac{4}{\zeta.\omega_n} \leftrightarrow [2\%] \Rightarrow \zeta.\omega_n = [4{,}00 \quad 0{,}80] \overset{\zeta \cong 0{,}7}{\Rightarrow} \omega_n = [5{,}71 \quad 1{,}14][rad/s] = [0{,}91 \quad 0{,}18][Hz]$$

Equação 5-1: Cálculo da freqüência do sistema controlado.

Assim, para atender a estes requisitos de desempenho o sistema de controle em malha fechada (no caso do sistema RP-UAV, o piloto ou algum sistema de aumento de estabilidade tem a função de fechar a malha de controle e melhorar o desempenho da aeronave) precisa possuir pólos dominantes com freqüência entre 0,91 e 0,18Hz.

Da teoria da amostragem e reconstrução de sinais digitais bem como da implementação digital de controladores (OGATA 1995), recomenda-se uma freqüência de amostragem de 6 a 12 vezes do sistema que se deseja observar, no caso o sistema em malha fechada. Assim fica claro que o sistema deve operar com freqüência da ordem de 5 a 15 Hz.

- Dinâmica da reação humana

Para interagir com o sistema, o piloto precisa ser capaz de acessar e controlar os estados do VANT. O termo "*user interface*" ou "interface homem-máquina" é frequentemente usado no contexto de sistemas computacionais significando o conjunto de meios pelos quais os usuários interagem com uma máquina, dispositivo, computador ou outra ferramenta complexa. A *user interface* prove meios de:

- **Entrada**: melhor conhecido como CUI. Permite aos usuários manipularem o sistema, que neste projeto é caracterizado pelo *joystick*;
- **Saída**: melhor conhecida como GUI. Permite ao sistema produzir os efeitos da manipulação do usuário, que neste projeto é caracterizado pela interface com o piloto RP-UAV.

Para tanto, a CUI e a GUI devem ser compatíveis com a dinâmica humana, pois o operador fecha a malha de controle.

O tempo de reação a reflexo é caracterizado pelo tempo entre um estímulo e a reação do organismo. Existem quatro tipos de tempo de reação (PURVES, 2007):

- **Tempo de reação simples**: é o tempo para o organismo reagir a um estímulo simples ou pequena mudança no ambiente;

- **Tempo de reação complexo**: é a latência entre um estímulo variável e uma resposta respectivamente variável;

- **Tempo de reação de reconhecimento**: é o tempo que envolve a decisão entre a resposta ou não a um dado evento;

- **Tempo de reação a escolha**: é o tempo que envolve a decisão de qual resposta para um dado evento, ou seja, respostas definidas para cada evento e diferentes entre eventos.

Fica claro que, para a CUI e a GUI, o tempo de reação à escolha deve ser atendido. No entanto a quantificação desta variável tem sido tema de estudo como os estudos realizados pela escola de medicina de Stanford da qual extraiu-se o teste de reação simples (Figura 5-4) que foi aplicado ao piloto RP-UAV do projeto BR-UAV. A principal razão pela qual a quantização do tempo de reação a escolha é complexa fundamenta-se que esta depende de muitos fatores entre eles, os principais são: reconhecimento, escolha, número de estímulos, tipo de estímulo, intensidade do estímulo, doenças, distração. Outros fatores que também podem influenciar: prática, erros, fadiga, sexo, idade, raça, tremor dos dedos, mão direita ou esquerda, visão, sobriedade, batimentos cardíacos, drogas estimulantes (cafeína).

Embora o tempo de reação à escolha seja imprescindível ao projeto da CUI e da GUI, é trivial observar que este tempo de reação sempre será superior ao tempo de reação simples uma vez que o primeiro envolve processamento de informações e o segundo não. Assim adotar o tempo de reação simples revela-se conservador e suficiente. Na Figura 5-4 pode-se observar o teste de reação simples aplicado ao piloto RP-UAV do projeto BR-UAV Cleber Miniello Roma (piloto VFR e IFR de classes bimotor e instrutor de vôo). Embora se observe que o tempo de reação simples do mesmo seja 213,30±32,91 milisegundos, o tempo de reação simples mínimo do corpo humano é de 100ms. Outra referência de tempo de reação pode ser observada em HB (2007). Assim será utilizado 100ms como tempo de reação do piloto.

Figura 5-4: Teste de tempo de reação simples (Fonte: COGNITIVELABS, 2007).

Outra questão que deve ser considerada é o atraso inerente a processos computacionais. Estes atrasos devem ser imperceptíveis para o piloto (usuário) ou até mesmo perceptíveis, mas que não causem mal estar ou periculosidade no controle do sistema. Revela-se mais conservador definir que o atraso máximo seja imperceptível para o piloto. No entanto, quantificar a percepção dos pilotos aos atrasos do sistema revela-se ainda mais complicado que o tempo de reação. No entanto, valer-se-á de experiências empíricas para estabelecer este parâmetro. Em sistemas telefônicos e de vídeo conferência o tempo máximo de atraso até o qual não há percepção do usuário é de 150ms.

Até o presente foram discutidos os principais requisitos ligados ao tempo, podendo-se estabelecer um resumo dos requisitos temporais:

- o sistema deve operar com freqüência da ordem de 5 a 15 Hz;
- o tempo de reação simples mínimo do corpo humano é de 100ms;
- o tempo máximo de atraso até o qual não há percepção do usuário é de 150ms.

A intersecção destes requisitos estabelece dois requisitos temporais:

- **RT1-Freqüência do sistema: 10 Hz**; A intersecção do requisito 1 com o 2 estabelece que a freqüência do sistema deva estar entre 10 e 15 Hz. Sabendo que se utilizarmos uma freqüência maior que 10 Hz o sistema capturaria mais de uma vez o mesmo comando do piloto, 10 Hz, revela-se suficiente.
- **RT2-Atraso máximo do sistema: 150ms**.

5.2 Modelo e Simulação do Sistema Aviônico

5.2.1 Modelo da arquitetura do sistema

O modelo da arquitetura do sistema é composto pelos diagramas de blocos externos que representam a estrutura do sistema (Figura 5-5) e pelos diagramas de blocos internos que representam os relacionamentos entre os módulos que compõem a estrutura do sistema (Figura 5-6 e Figura 5-7) conforme a linguagem de modelagem SysML.

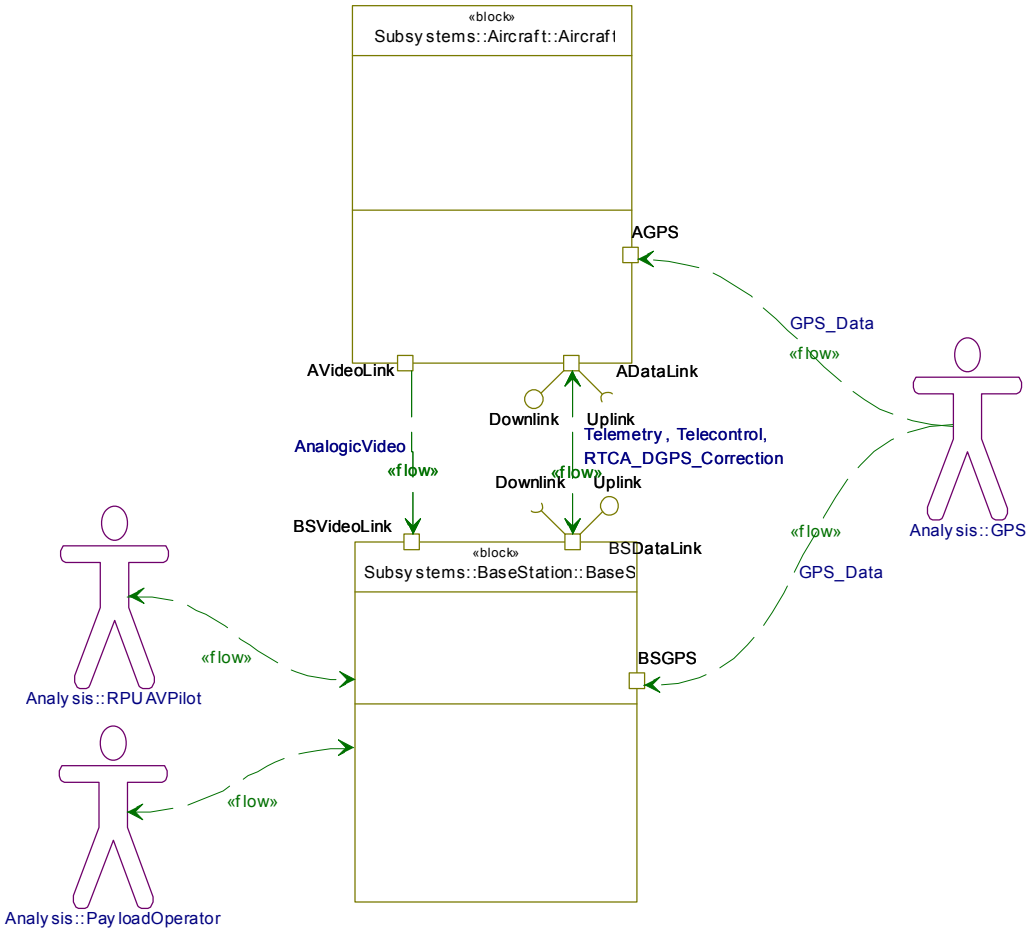

Figura 5-5: SysML External Block - Estrutura da aviônica.

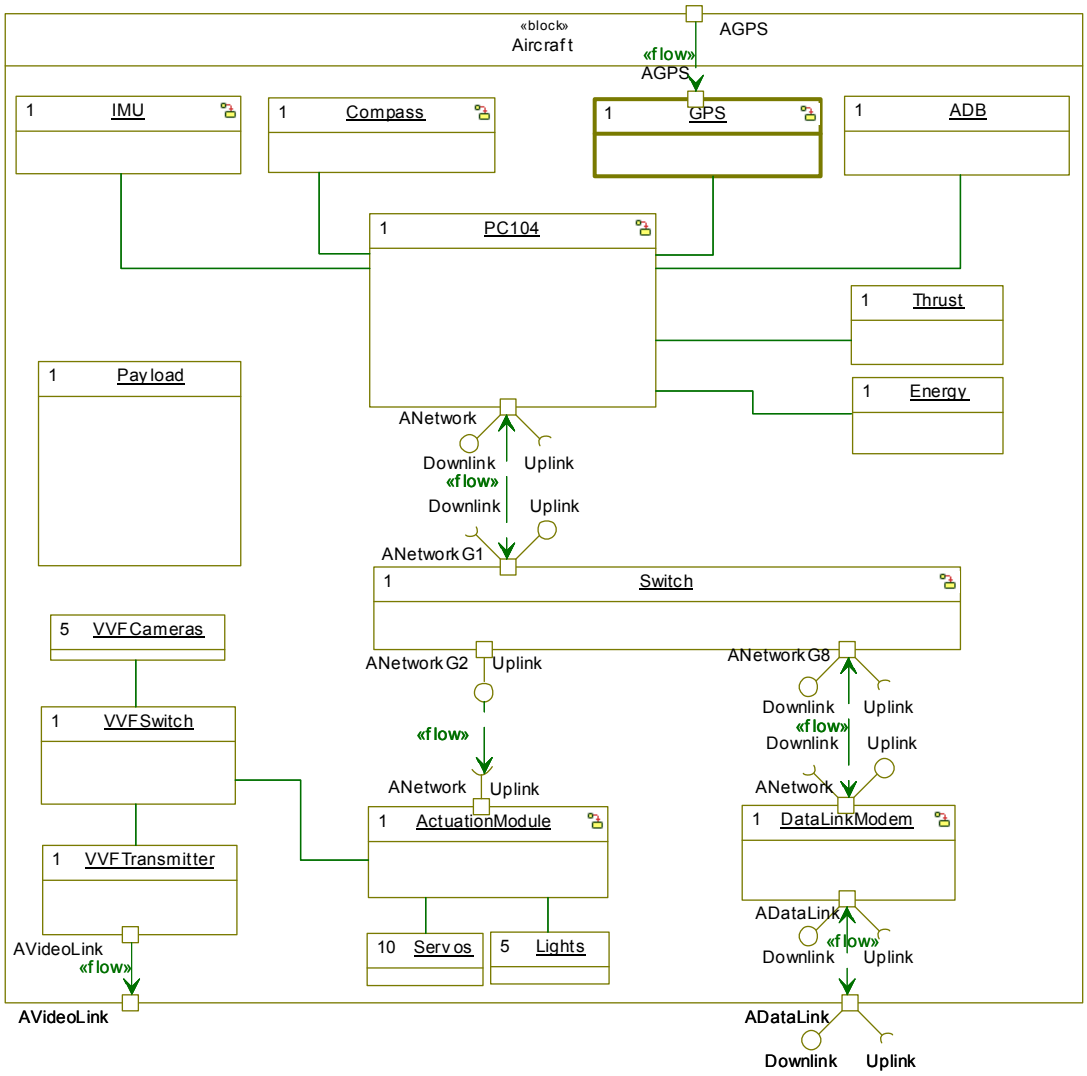

Figura 5-6: SysML Internal Block - Aircraft.

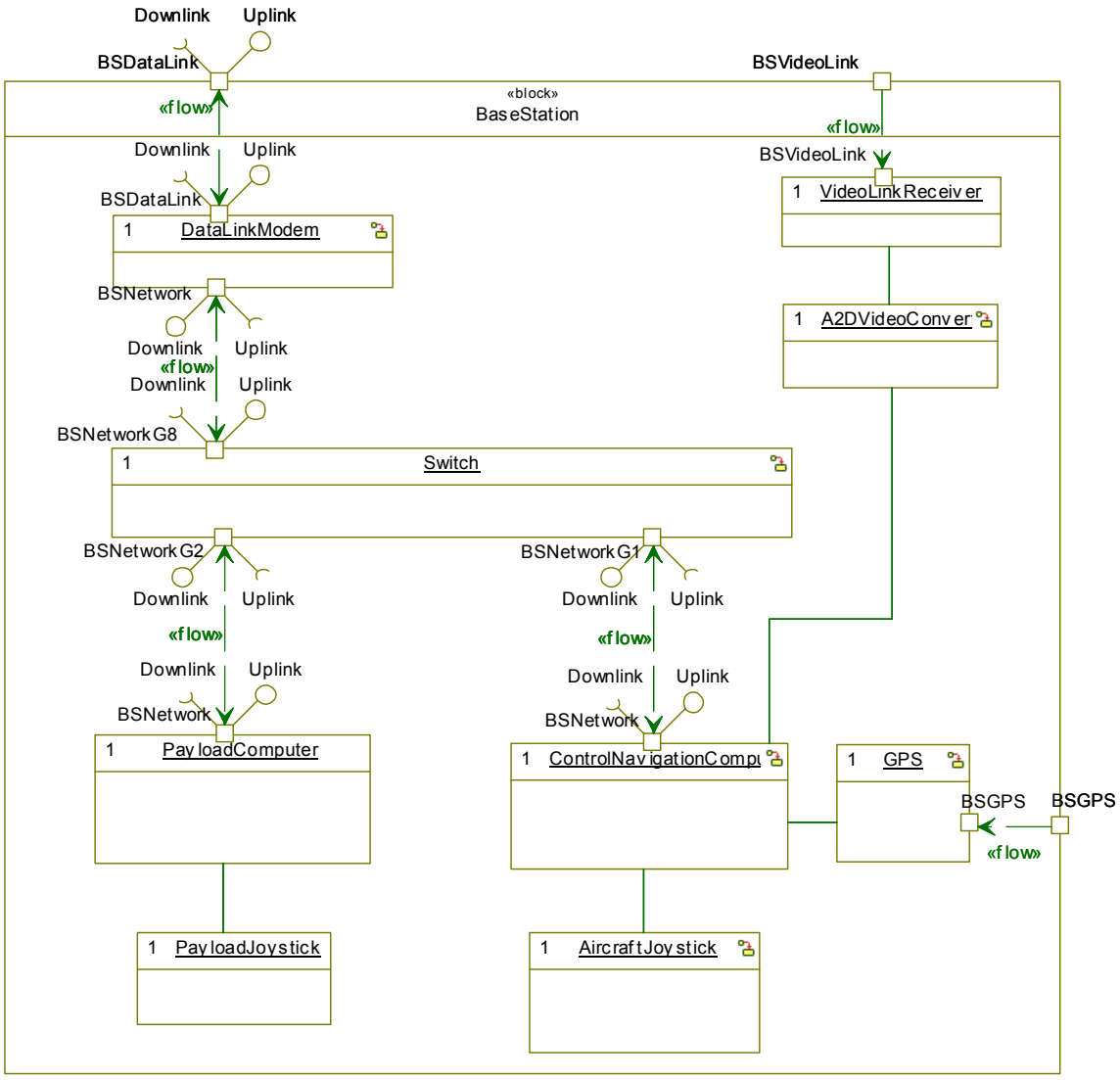

Figura 5-7: SysML Internal Block – BaseStation

5.2.2 Modelo do comportamento do sistema

O modelo do comportamento do sistema é implementado via diagramas de seqüência e *statecharts*. O diagrama de seqüência mostra as mensagens passadas entre módulos ao executar uma instância particular de um caso de uso, que neste caso será um ciclo de controle RP-UAV conforme pode ser observado na Figura 5-8, que é caracterizado por:

- aquisição se sensores;
- tratamento de sensores;
- filtro estimador de estados;
- *dowlink*;
- geração de marcadores;
- captura de imagem;
- fusão do frame de interface;
- aquisição do *joystick*;
- aquisição do GPS da base;
- *uplink*;
- geração de referências para os atuadores e correção do GPS da aeronave.

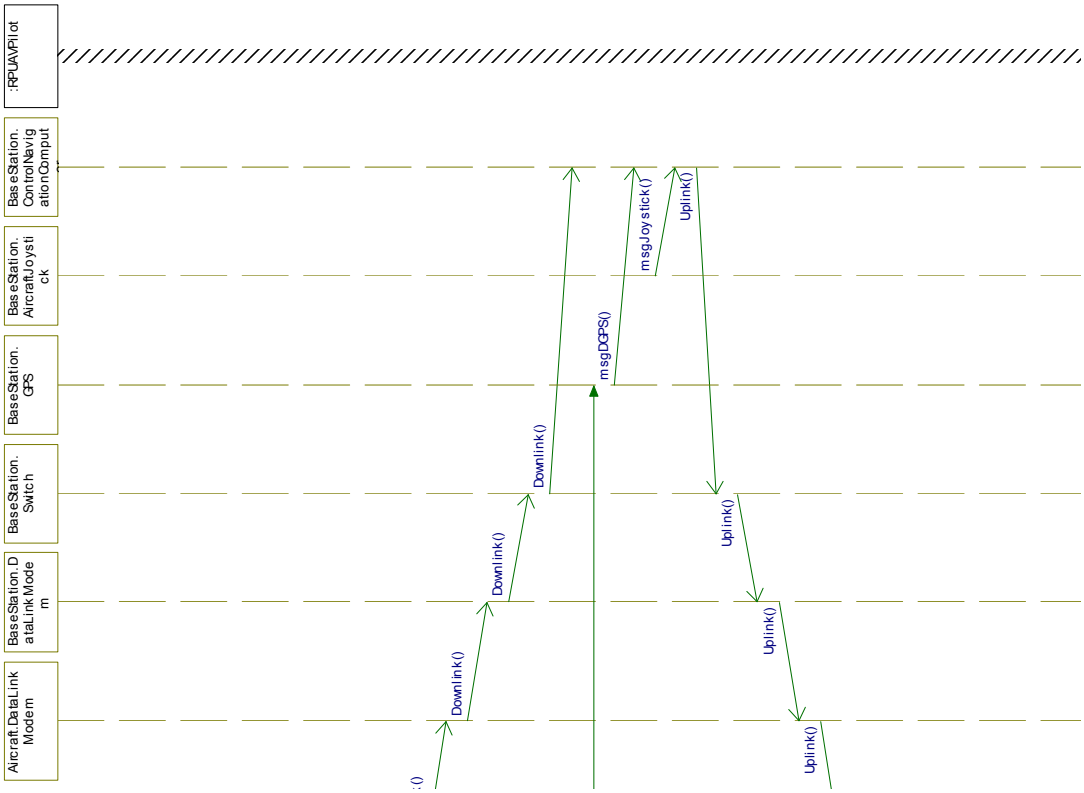

Figura 5-8: SysML Sequence - Diagrama de seqüência do RP-UAV.

Os *statecharts* modelam o comportamento de cada módulo bem como sua iteração com os demais. A modelagem dos sensores Aircraft_IMU (Figura 5-9), Aircraft_CPS (Figura 5-10), Aircraft_GPS (Figura 5-11) e BaseStation_GPS (Figura 5-12); dos *switchs* Aircraft_Switch (Figura 5-13) e BaseStation_Switch (Figura 5-14); dos *data-links* Aircraft_DataLink (Figura 5-15) e BaseStation_DataLink (Figura 5-16); do conversor de vídeo analógico-digital BaseStation_A2DvideoConverter (Figura 5-17); e da interface de controle com

o piloto BaseStation_Joystick (Figura 5-18), seguem as especificações de concorrência, tempos, e comunicação definidos pelos respectivos fabricantes.

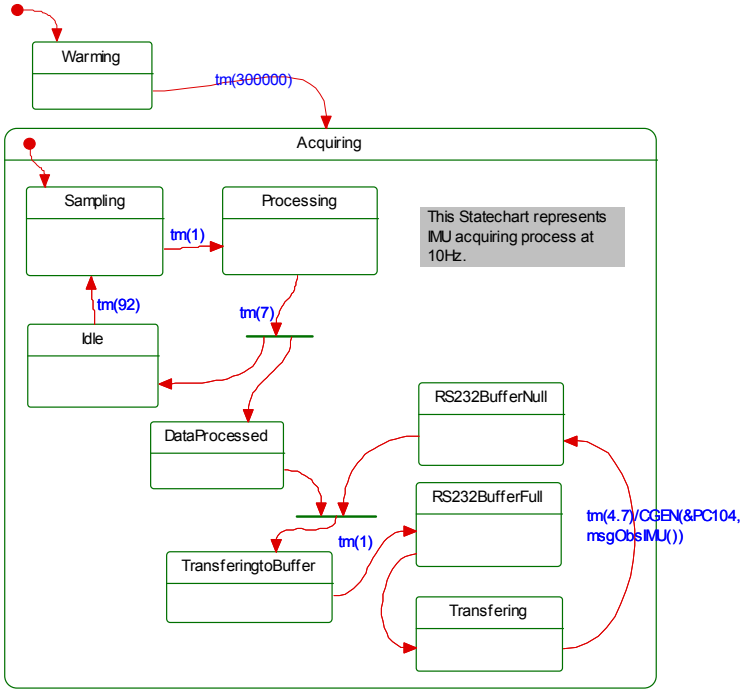

Figura 5-9: SysML Statechart - Aircraft_IMU.

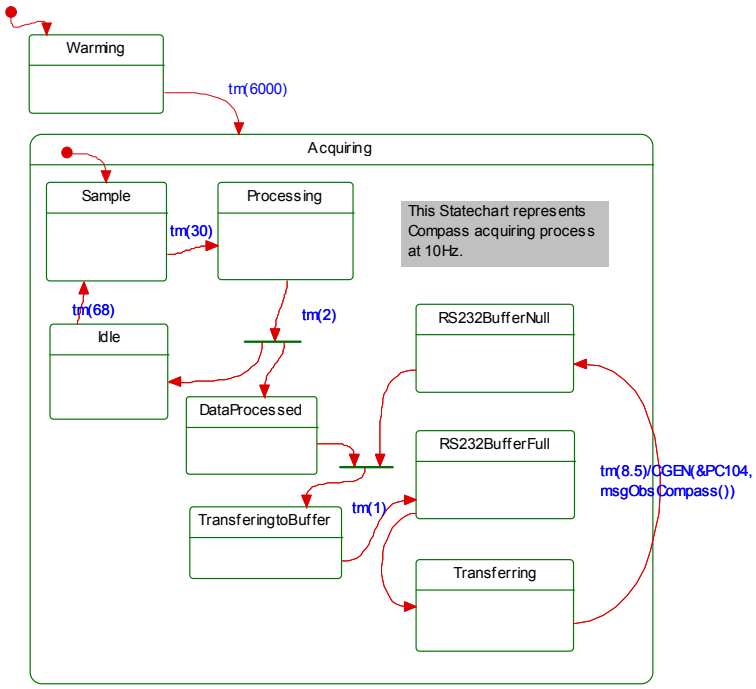

Figura 5-10: SysML Statechart - Aircraft_CPS.

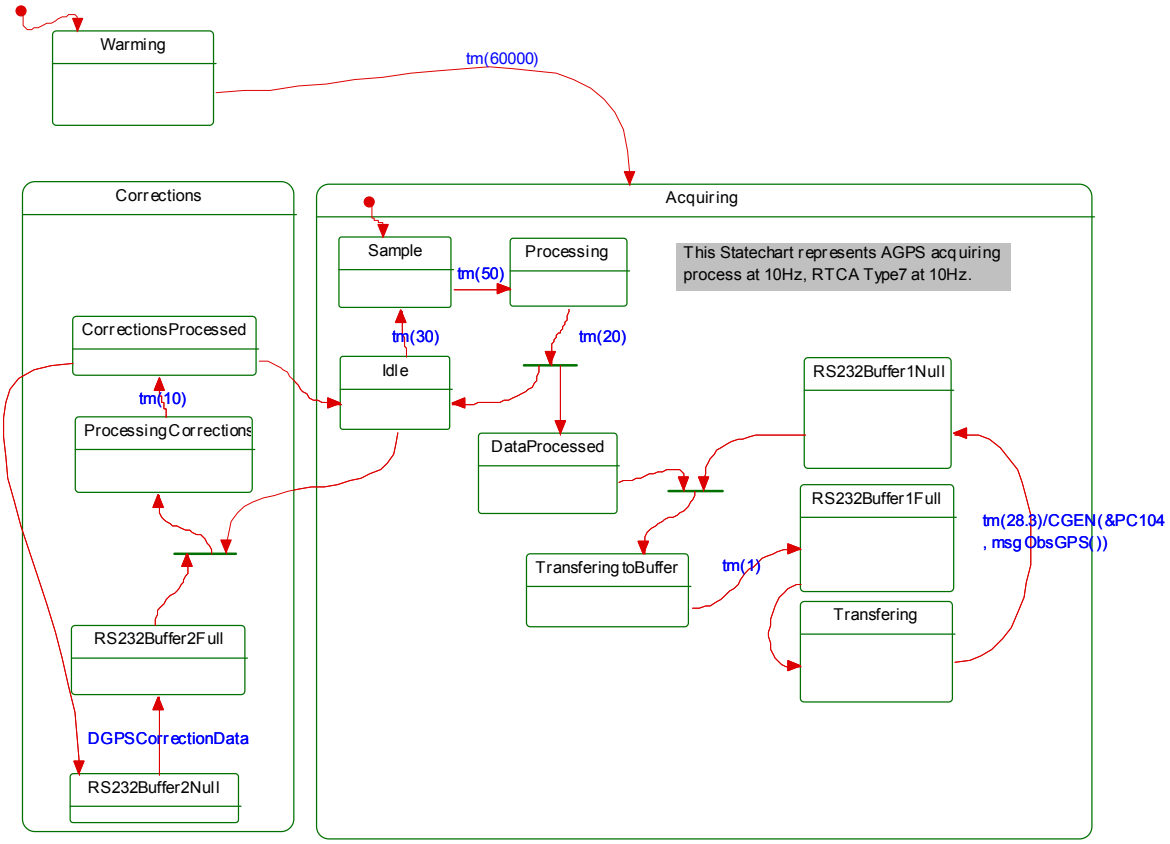

Figura 5-11: SysML Statechart - Aircraft_GPS.

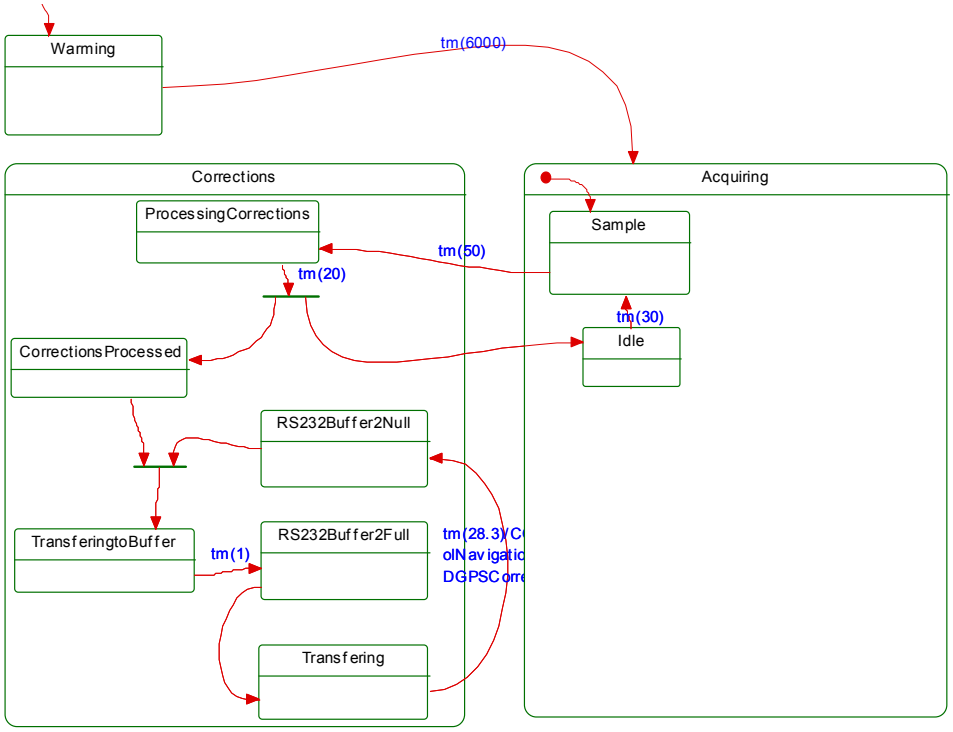

Figura 5-12: SysML Statechart - BaseStation_GPS.

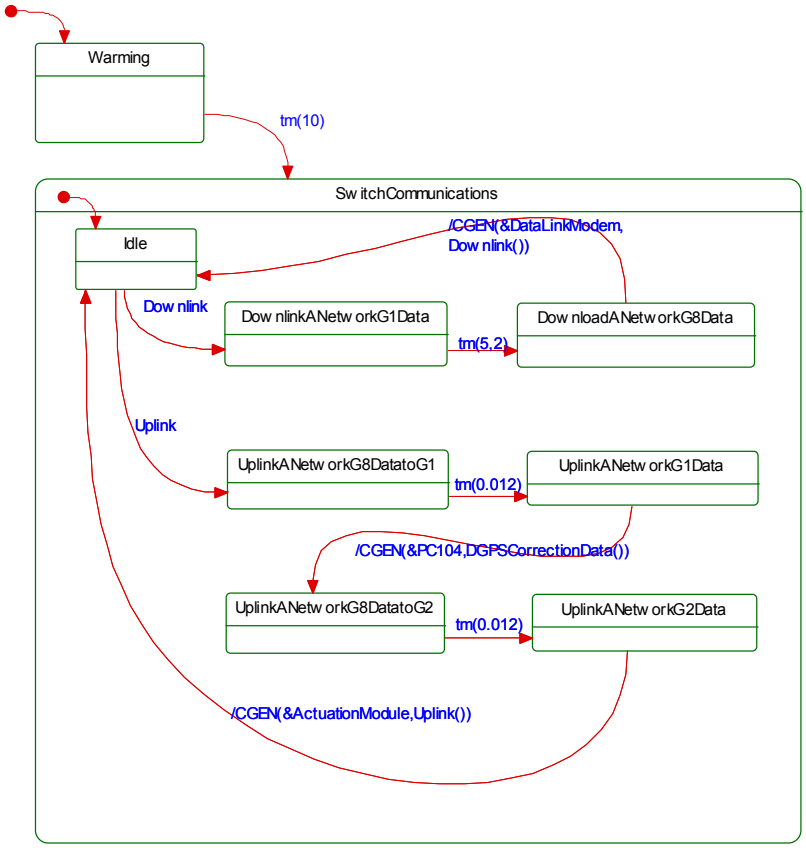

Figura 5-13: SysML Statechart - Aircraft_Switch.

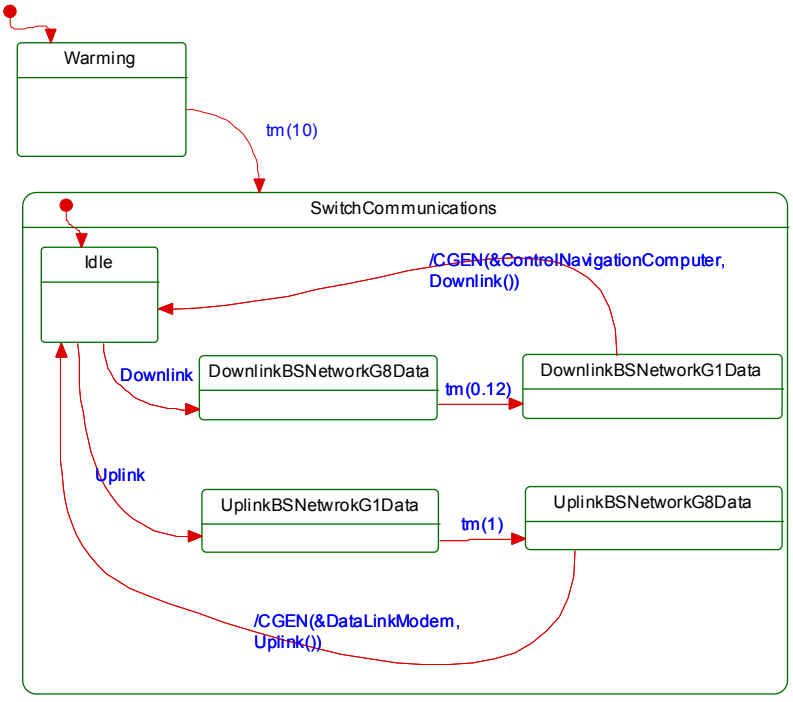

Figura 5-14: SysML Statechart - BaseStation_Switch

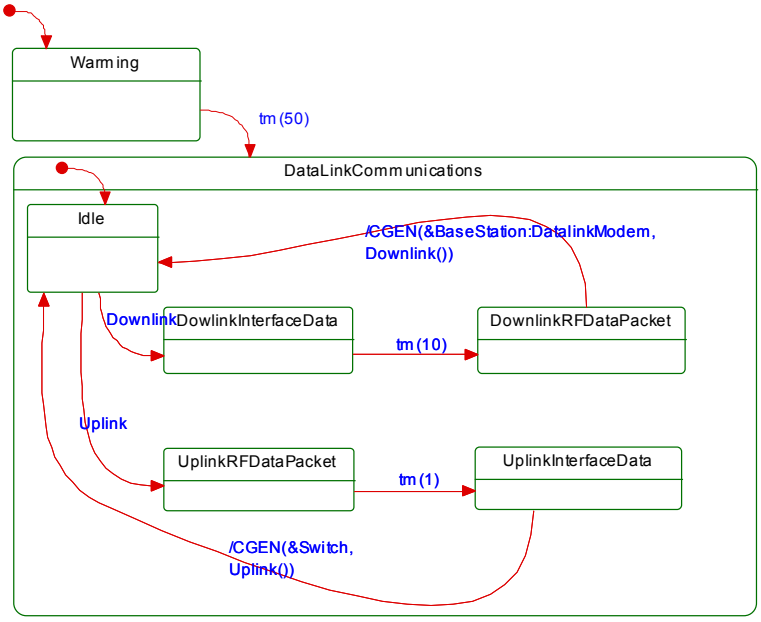

Figura 5-15: SysML Statechart - Aircraft_DataLink.

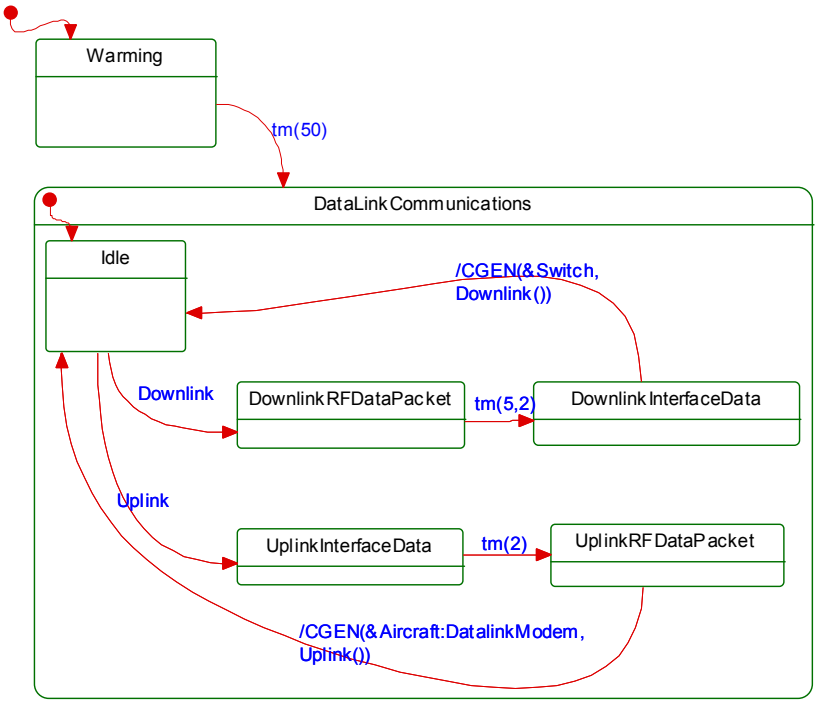

Figura 5-16: SysML Statechart - BaseStation_DataLink.

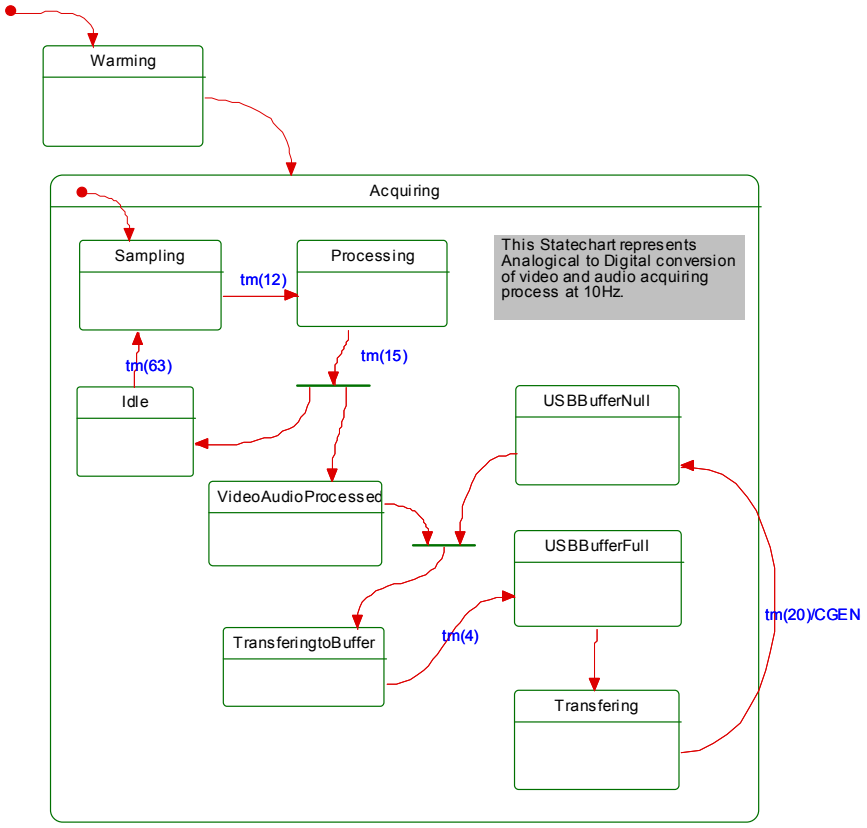

Figura 5-17: SysML Statechart - BaseStation_A2DVideoConverter.

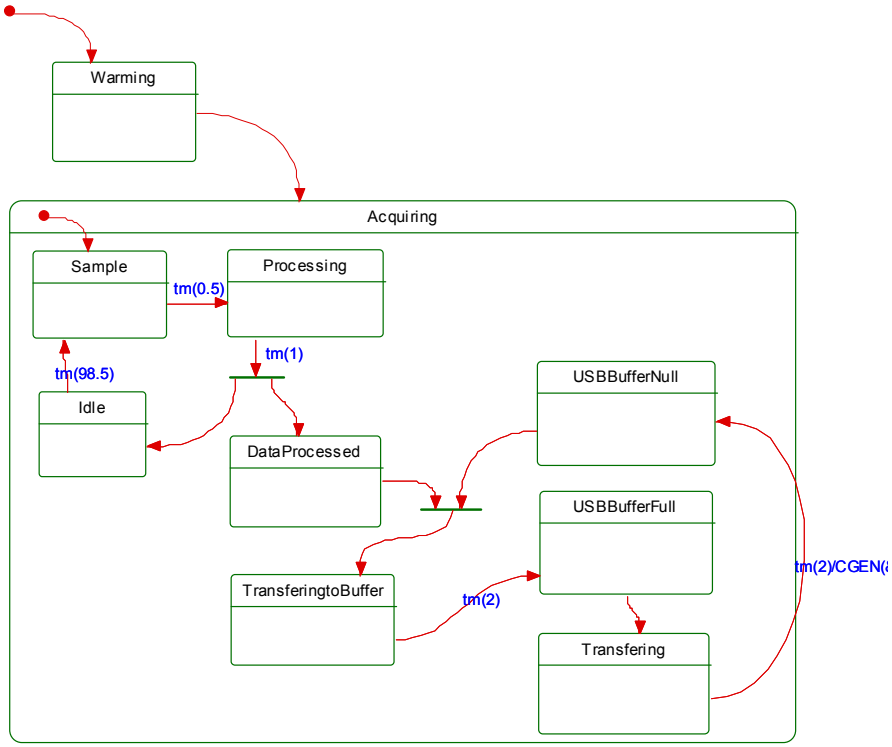

Figura 5-18: SysML Statechart - BaseStation_Joystick.

Já a modelagem das ECUs Aircraft_PC104 (Figura 5-19), Aircraft_ActutationModule (Figura 5-20) BaseStation_ControlandNavigationComputer (Figura 5-21) são uma proposta de comportamento de como os módulos podem interagir. Os tempos envolvidos são estimados e após o refinamento do sistema estes tempos definirão os requisitos temporais para o software.

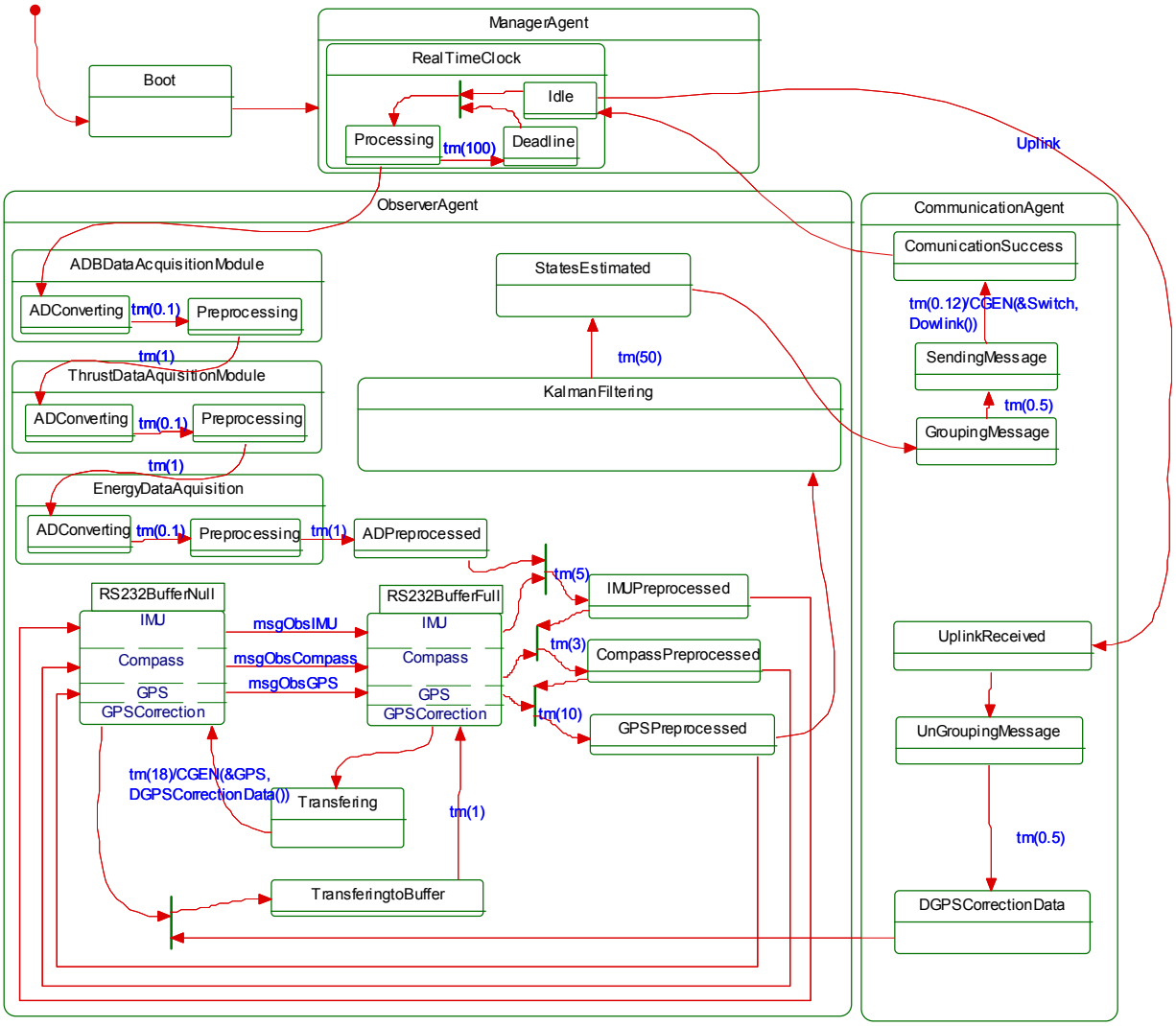

Figura 5-19: SysML Statechart - Aircraft_PC104.

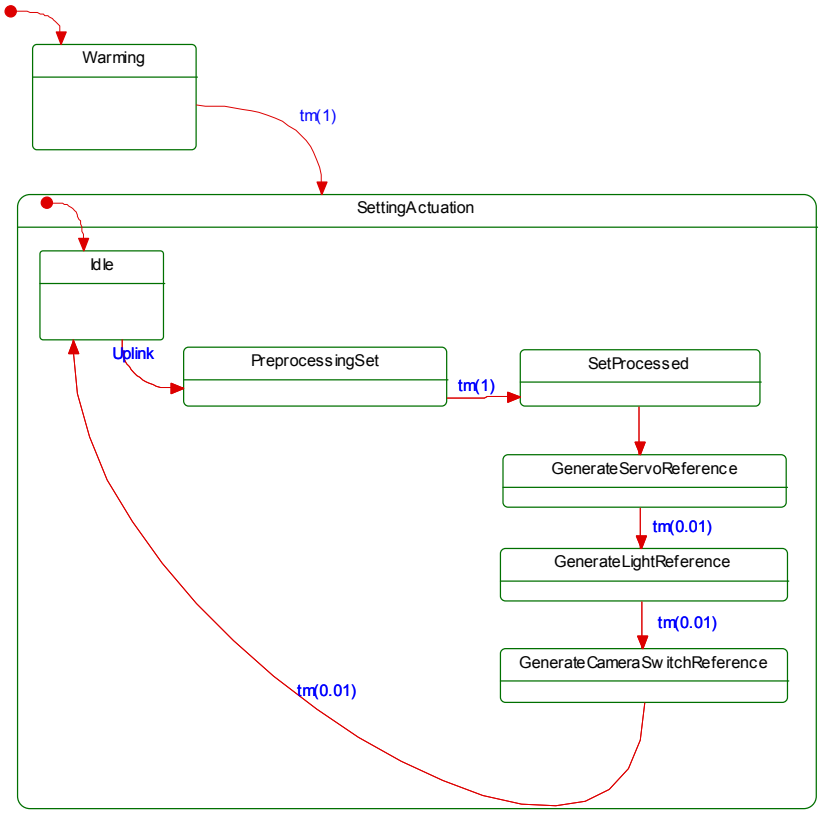

Figura 5-20: SysML Statechart - Aircraft_ActuationModule.

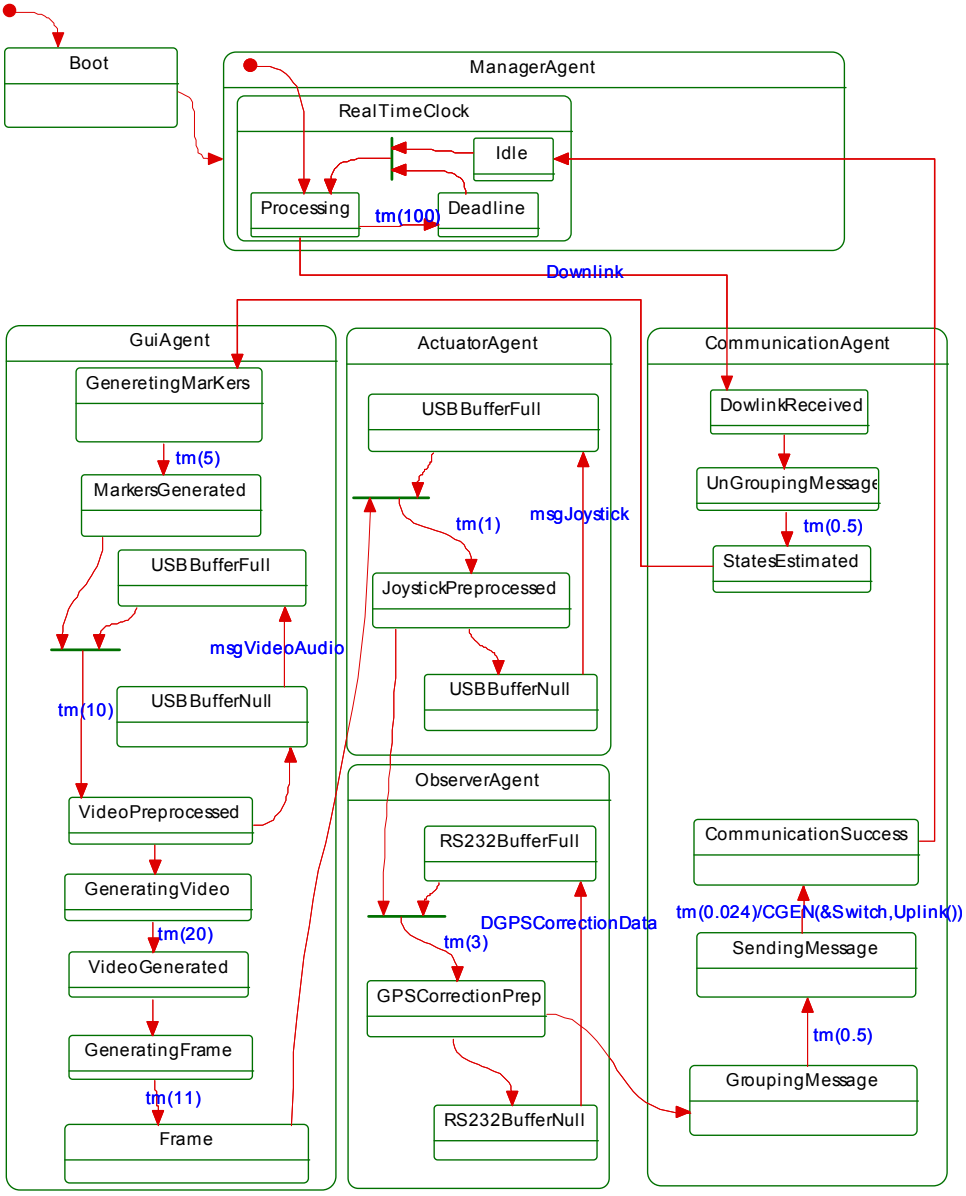

Figura 5-21: SysML Statechart - BaseStation_ControlNavigationComputer.

Após a simulação do modelo apresentado no item anterior, pode-se realizar a análise de desempenho do sistema e detectar se o mesmo atende aos dois requisitos temporais RT1 e RT2, bem como estudar as possíveis modificações de melhorar o desempenho do sistema.

Na Figura 5-22, pode-se observar a previsão da carga de processamento em cada um dos elementos que compõem o sistema normalizado para um ciclo de controle, ou seja, 100ms. Observa-se que todos os elementos possuem carga de processamento inferior a 100%. Isto prevê o atendimento ao requisito 1, uma vez que nenhum elemento precisa de mais tempo de processamento que 100ms, o que seria evidenciado por uma carga de processamento superior a 100%.

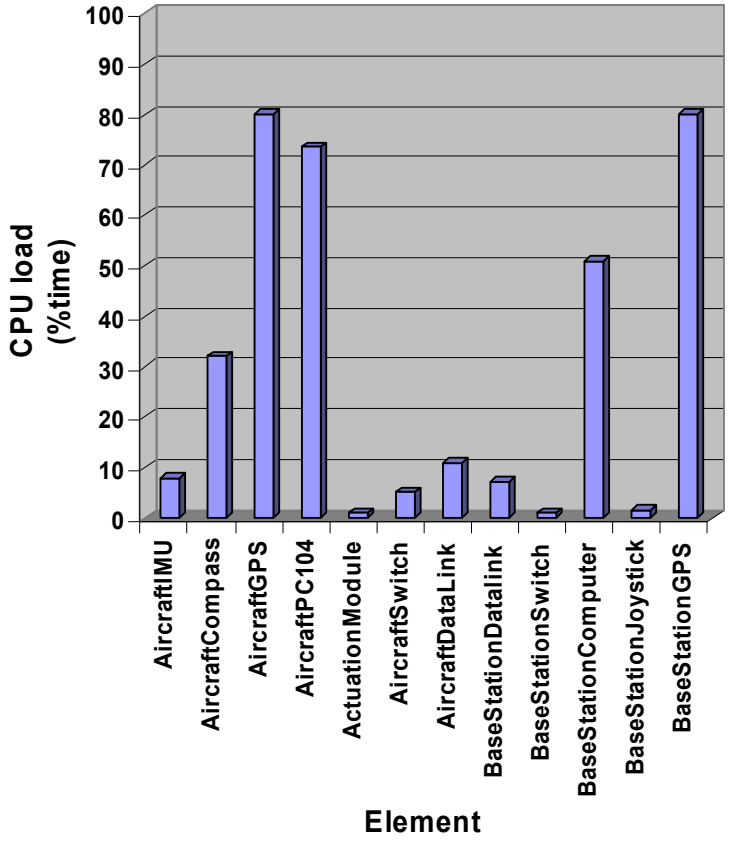

Figura 5-22: Carga de processamento em cada um dos elementos que compõem o sistema.

Entretanto, pode-se observar que quatro elementos do sistema possuem carga computacional significativa, maior que 50%: AircraftGPS, AircraftPC104, BaseStationComputer e BaseStationGPS. Destes quatro elementos, somente no AircraftPC104 e no BaseStationComputer podem ser implementadas modificações para a melhoria de desempenho (redução da carga computacional), pois, os GPSs possuem hardware e software proprietários e foram projetados para esta condição de operação revelando-se otimizados na integração hardware x software.

Na Figura 5-23, pode-se observar a distribuição de carga de processamento em cada um dos passos de processamento do AircraftPC104. Observa-se que o processamento concentra-se no filtro que Kalman, pois o mesmo é caracterizado por uma serie de cálculos matriciais que envolvem inversões e mau condicionamento. Entretanto existem dezenas de métodos computacionais para implementação deste filtro que melhoram seu desempenho para uma dada condição de operação e para esta aplicação deve-se estudar o algoritmo que melhor se adequa às características de dinâmica da aeronave e do filtro bem como freqüência de cálculos.

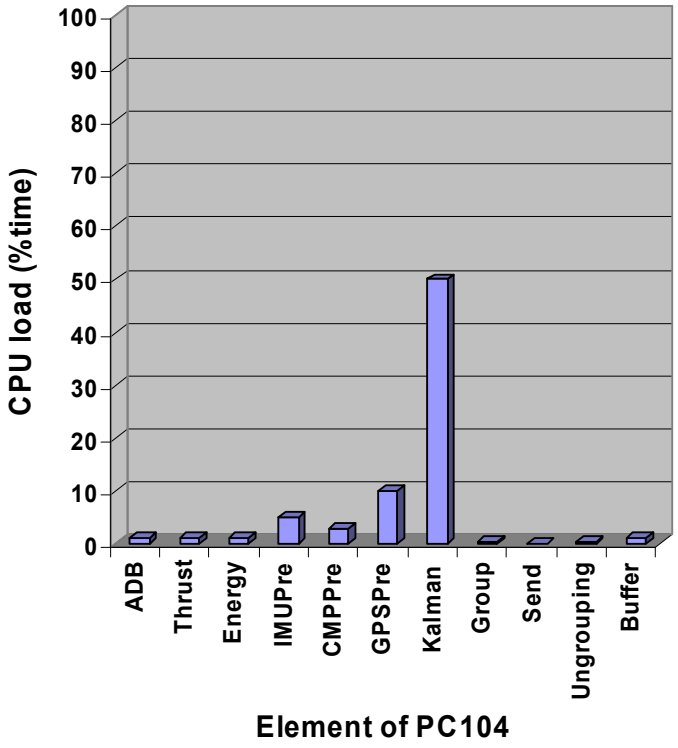

Figura 5-23: Carga de processamento no elemento AircraftPC104.

Na Figura 5-24, pode-se observar a distribuição de carga de processamento em cada um dos passos de processamento do BaseStationComputer. Observa-se que a carga de processamento concentra-se no processamento de imagens que consiste nos seguintes passos: 1**-geração de marcadores** que envolve a rotação e translação de imagens, **2-preprocessamento de vídeo** que corresponde à captura do mesmo do buffer; **3-geração do vídeo** que é a conversão do frame de vídeo para o formato de disparo na tela e finalmente a **4-geração do frame** que corresponde sobreposição dos marcadores à imagem de vídeo e atualizados na tela.

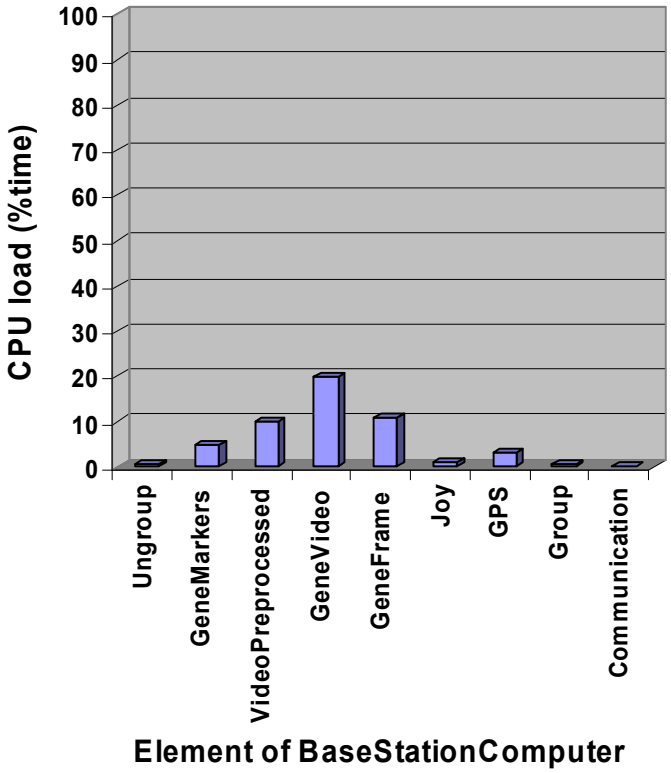

Figura 5-24: Carga de processamento no elemento BaseStationComputer.

Na Figura 5-25, verifica-se em azul os atrasos gerados por cada um dos elementos do sistema tanto no *Dowlink* como no *Uplink* bem como em vermelho os atrasos acumulativos do sistema tanto no *Downlink* como no *Uplink*. Os atrasos acumulativos revelam que o tempo máximo de *delay* é de 148,5 milisegundos e, portanto o sistema atende ao requisito RT2. Já os atrasos dos elementos do sistema revelam que os maiores geradores de atraso são o PC104 e o BaseComputer e se forem comparados com a carga de processamento em cada um dos elementos, verifica-se que todo o processamento do PC104 e todo o processamento do BaseComputer geram atraso. Isto não acontece com o grupo IMU, CPS e GPS que embora tenham cargas computacionais significativas, geram atrasos reduzidos uma vez que os mesmos processam as informações das amostras em paralelo com o processamento do PC104. Assim para melhorar o desempenho de atraso, deve-se trabalhar o processamento do PC104 e o processamento do BaseComputer que por sua vez recaem na otimização do filtro de Kalman e do processamento de imagens, respectivamente.

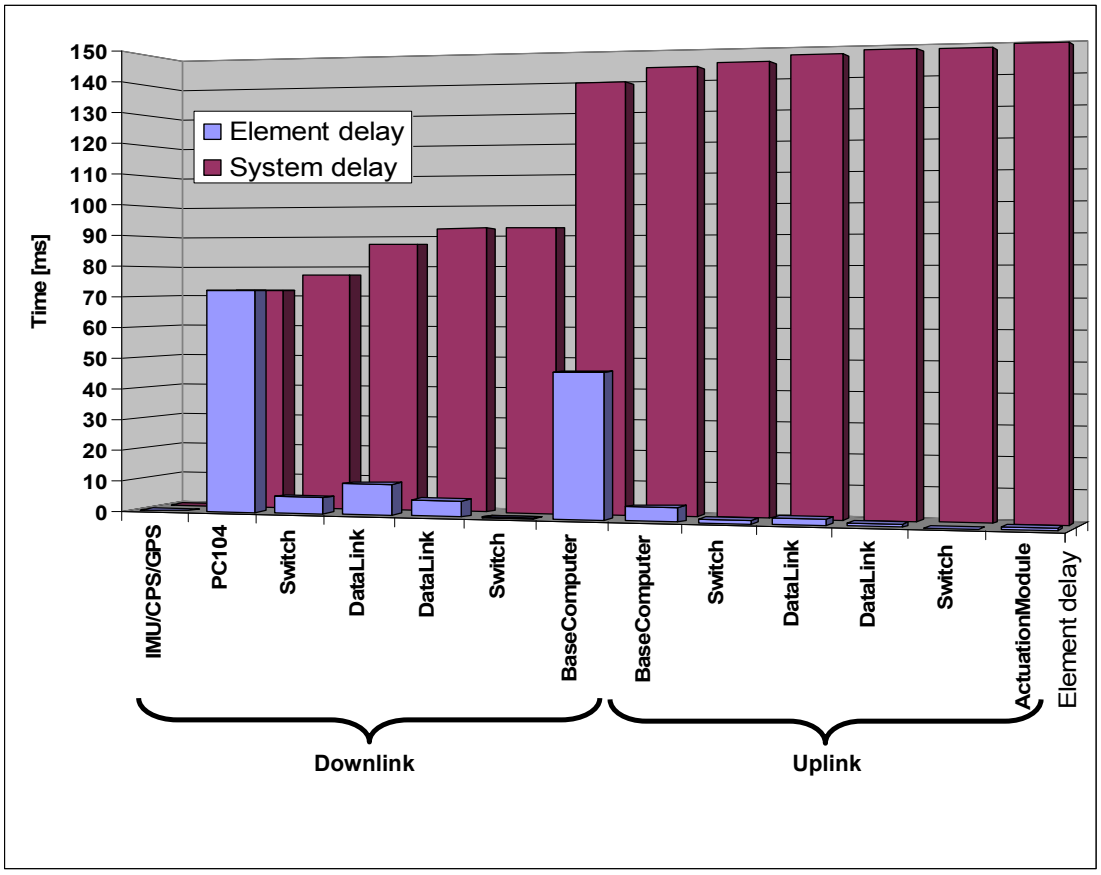

Figura 5-25: Atraso gerado por cada elemento do sistema (element delay) e atraso acumulado do sistema (system delay).

5.3 Análise dos Requisitos do Software Embarcado

A partir dos requisitos do sistema definidos na seção 5.1, pôde-se modelar na seção 5.2 o hardware e software tanto embarcado como da estação base. Este modelo por sua vez gerou os requisitos para o software embarcado que são apresentados nesta seção.

5.3.1 Requisitos de Implantação

O hardware de processamento do sistema embarcado é composto por duas ECUs de funções distintas.

A primeira é a ECU de teste que é utilizada em bancada, pois utiliza componentes de baixo custo, para o dia a dia de desenvolvimento: placa mãe genérica com barramento PCI e ISA, processador x86 A462, disco rígido genérico, *ethernet* PCI 3C906 e expansão serial ISA 16c550. Os hardwares foram escolhidos de forma a haver *drivers* disponíveis no QNX, como pode ser observado no diagrama de implantação da Figura 5-26.

A segunda é a ECU embarcada que é utilizada na aeronave. Ela é composta por duas placas da pilha PC104: placa de processamento com processador x86, memória RAM

BGADDR, *ethernet* 82562, *flash disk* SSD32 ATA, e placa de expansão serial 16c550, como pode ser observado no diagrama de implantação da Figura 5-27.

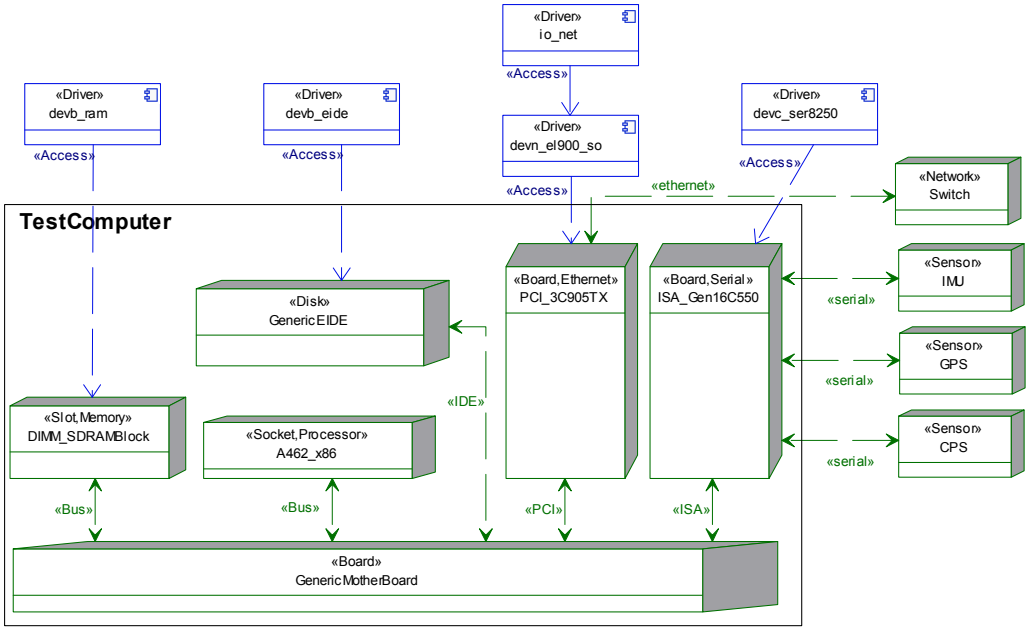

Figura 5-26: UML Deployment - ECU de teste.

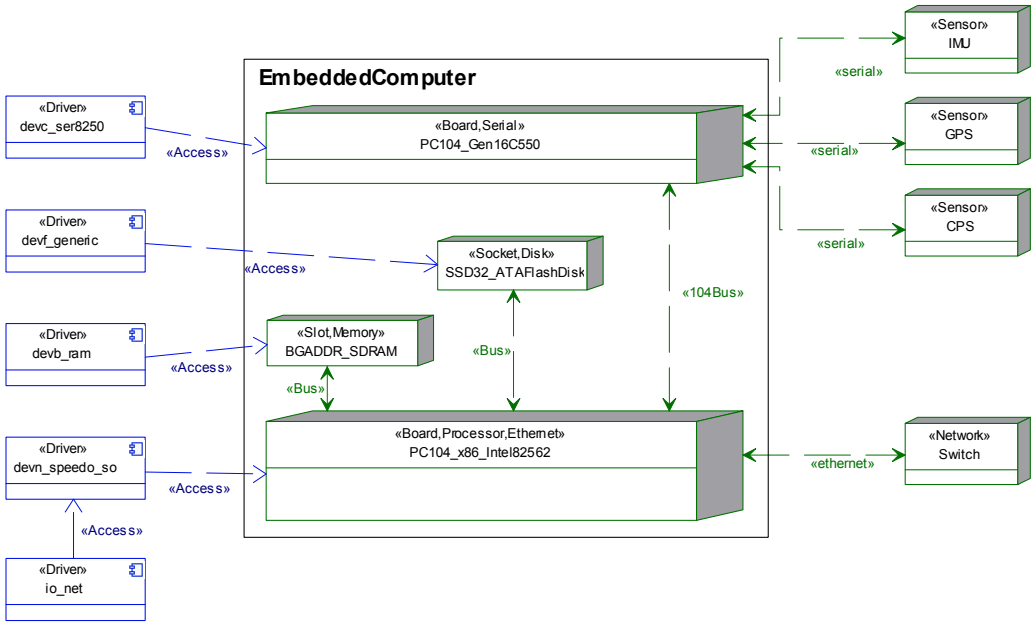

Figura 5-27: UML Deployment - ECU embarcada.

5.3.2 Requisitos Funcionais do Software

Os requisitos funcionais são representados por meio de casos de uso. Na Figura 5-28 pode-se observar uma visão geral dos casos de uso do software embarcado, ou seja, a base da estrutura hierárquica de requisitos funcionais. Na Figura 5-29, Figura 5-30, Figura 5-31 e

Figura 5-32 tem-se respectivamente a especificação dos casos de uso *Manage System*, *Manage Sensors*, *Manage Communication* e *Manage Actuation.*

Ressalta-se que todos os componentes modelados neste trabalho que estão na cor azul ciano diferenciam as partes do sistema que não foram implementadas, mas que foram especificadas.

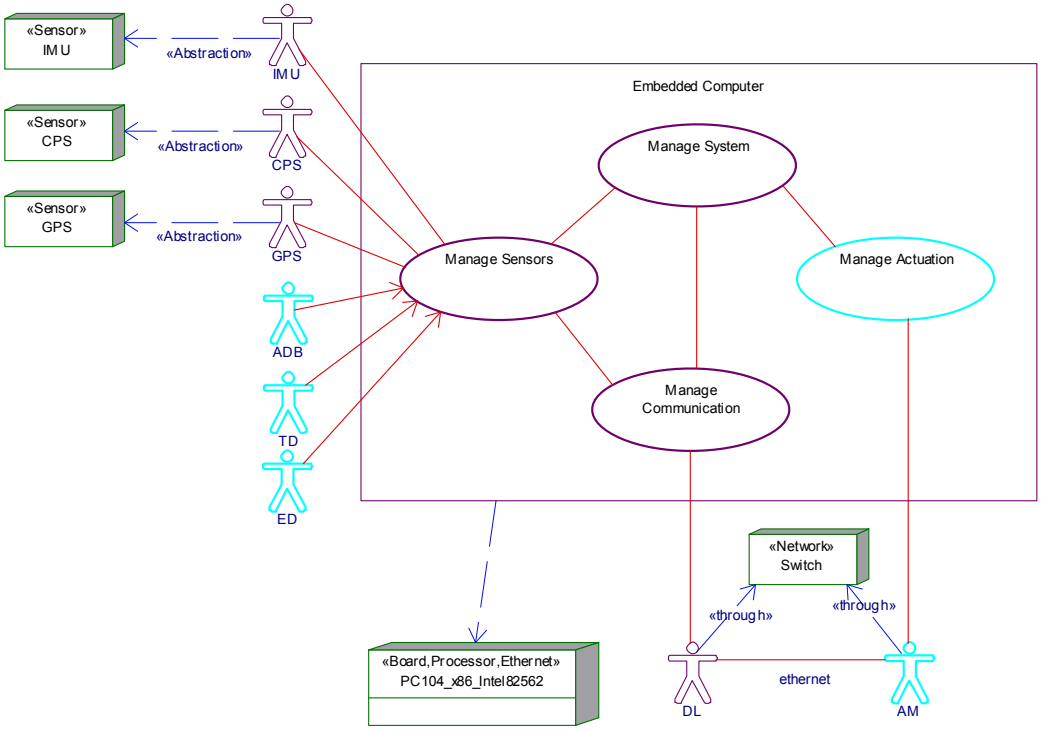

Figura 5-28: UML Use Case - Visão geral do sistema.

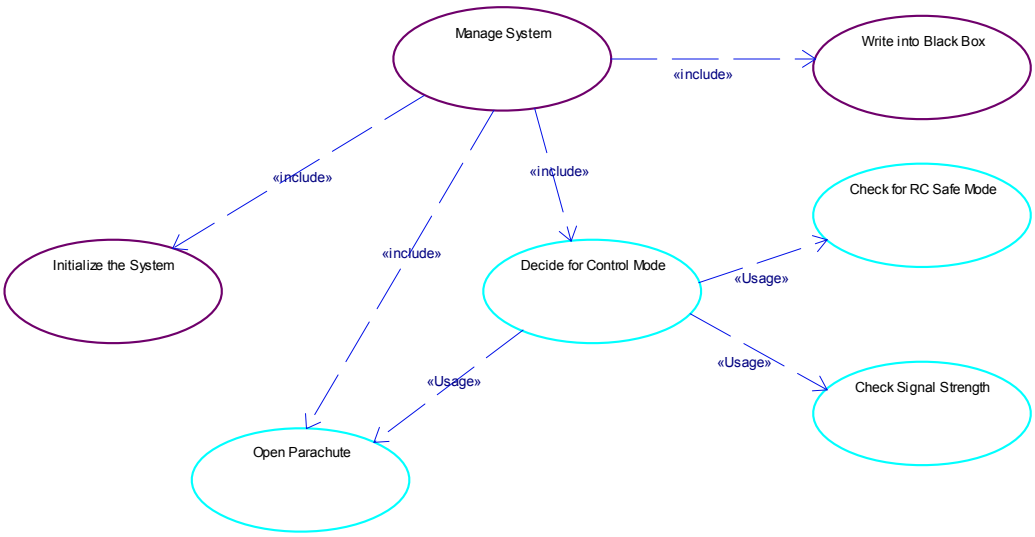

Figura 5-29: UML Use Case - Gerencia sistema.

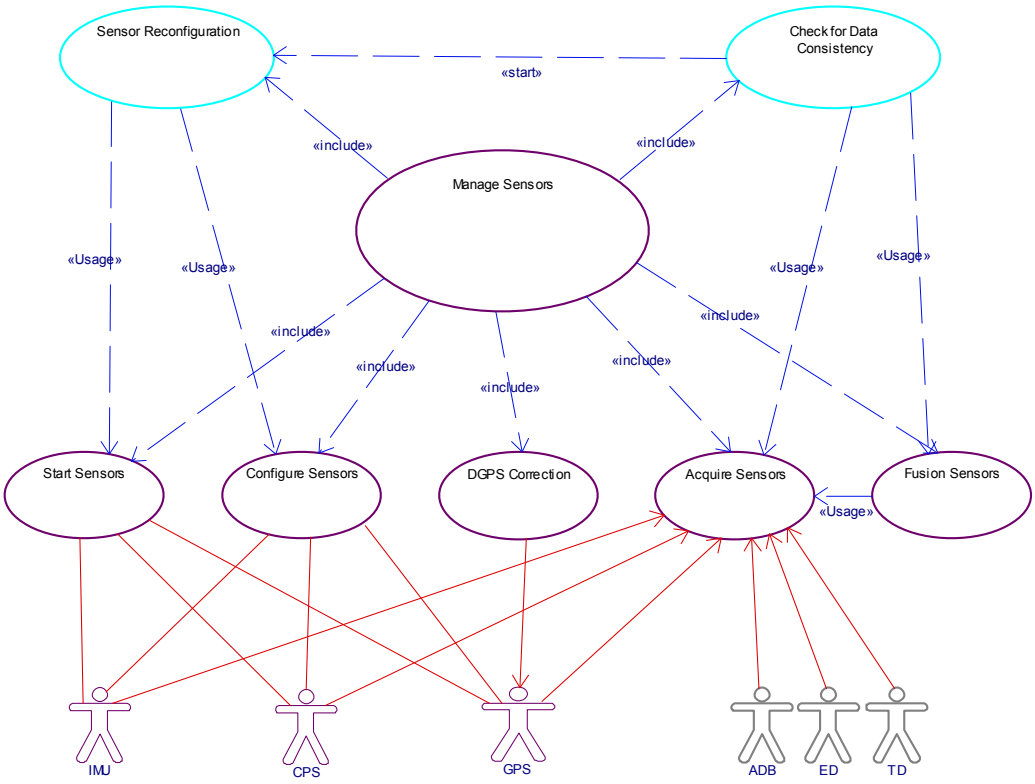

Figura 5-30: UML Use Case - Gerencia sensores.

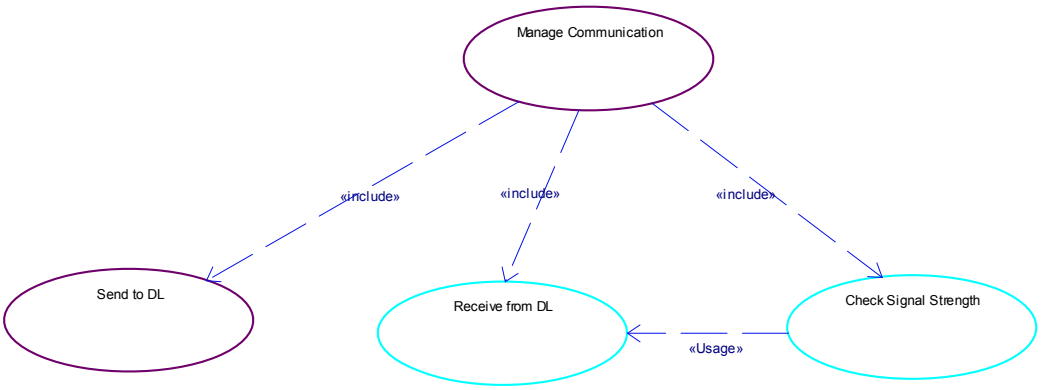

Figura 5-31: UML Use Case - Gerencia comunicação.

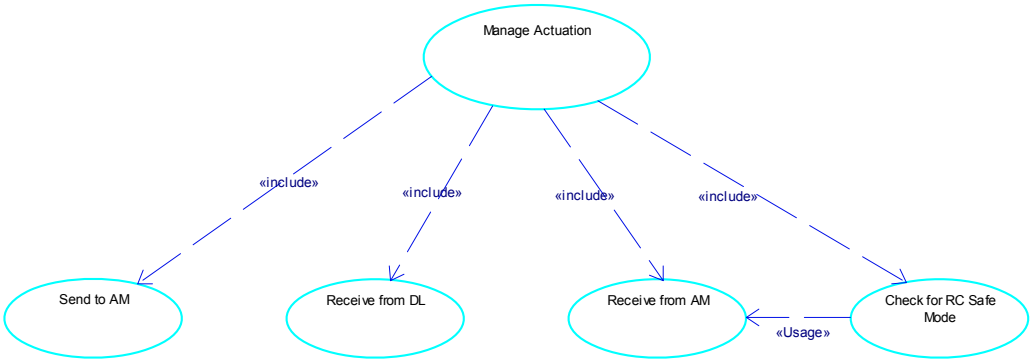

Figura 5-32: UML Use Case - Gerencia atuação.

5.3.3 Requisitos Temporais do Software

Os requisitos temporais são representados por diagramas de seqüência dos principais casos de uso, como pode ser observado da Figura 5-33 à Figura 5-38. Na Figura 5-33 tem-se a visão geral da seqüência de eventos composta por aquisições de sensores, ciclos de predição e atualização da fusão sensorial, envio de mensagem à base, recebimento de mensagem de atuação bem como de correção do DGPS.

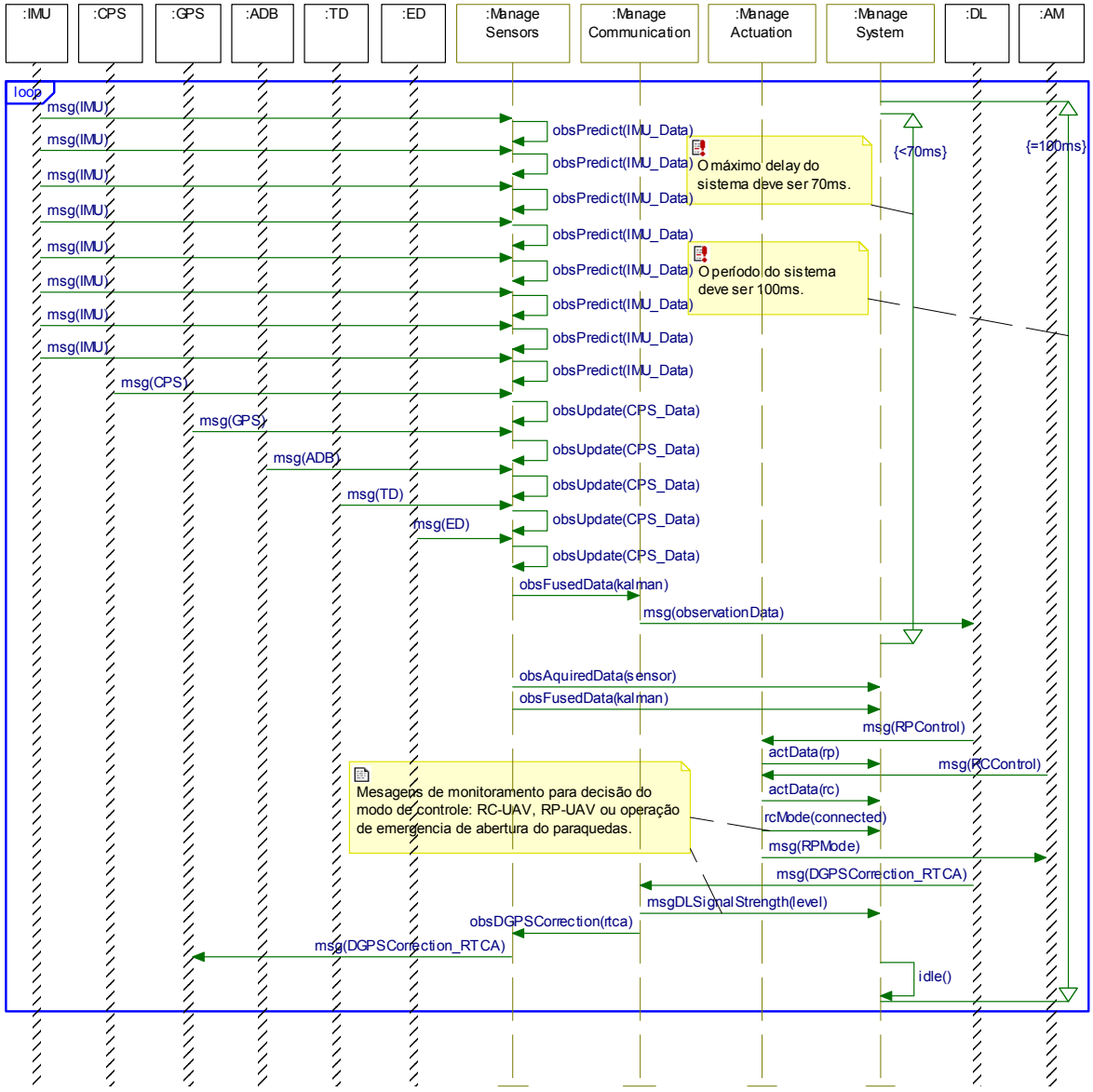

Figura 5-33: UML Sequence - Visão geral de um ciclo com sucesso.

Na Figura 5-34, tem-se o gerenciamento do sistema em casos de sucesso e em casos de falhas. Os casos de falhas foram compostos por recuperação da operação do RP-UAV pelo RC-UAV ou por recuperação da operação do RP-UAV pela abertura de pára-quedas e aborto de missão.

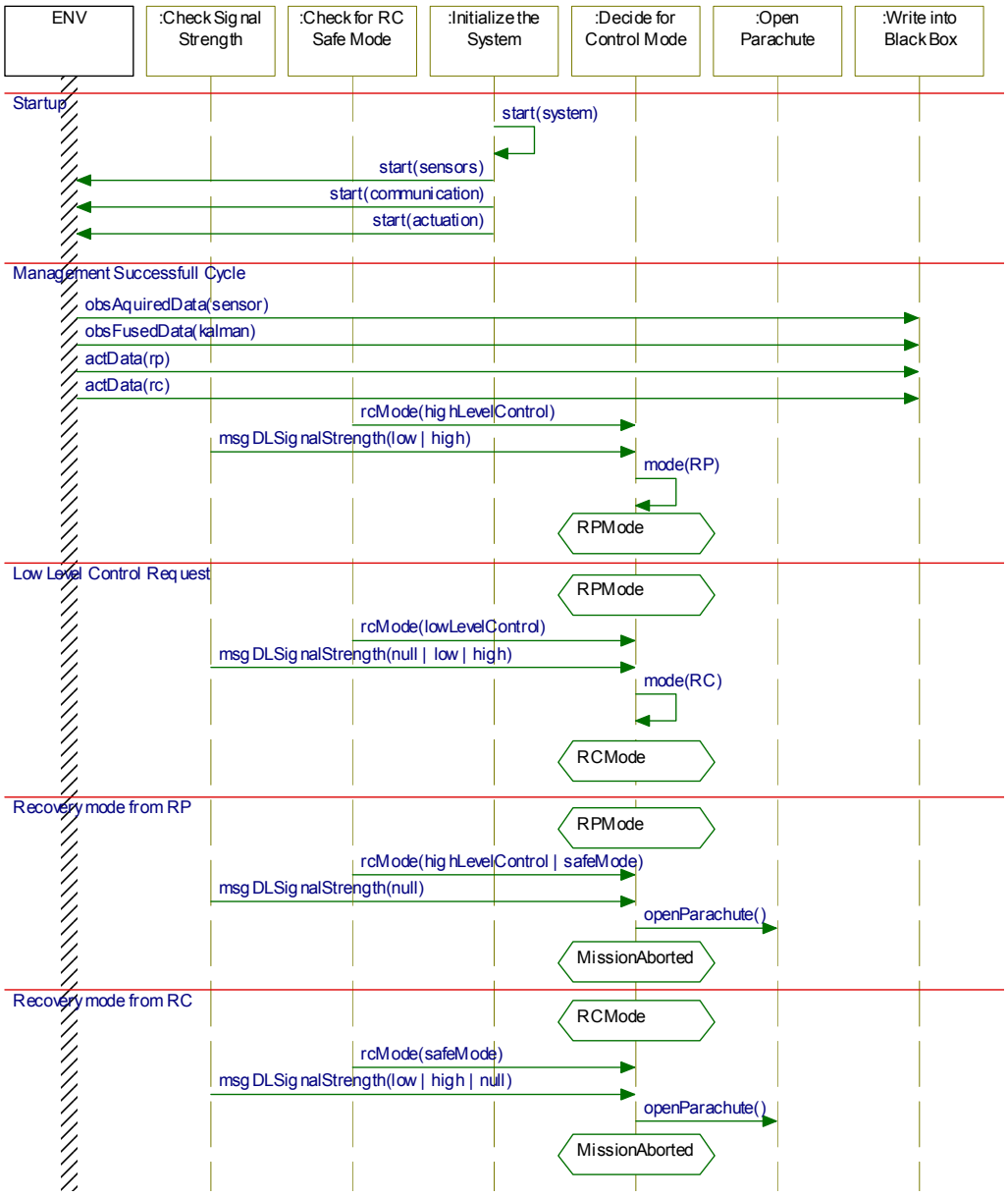

Figura 5-34: UML Sequence - Gerenciamento sistema em casos de sucesso e falha.

Na Figura 5-35, tem-se um detalhamento do funcionamento dos sensores em casos de sucesso. Já na Figura 5-36 dois casos de falha: dados negados devido sua inconsistência e dados negados seguido de tratamento, pois o número de falhas detectadas era maior que o máximo.

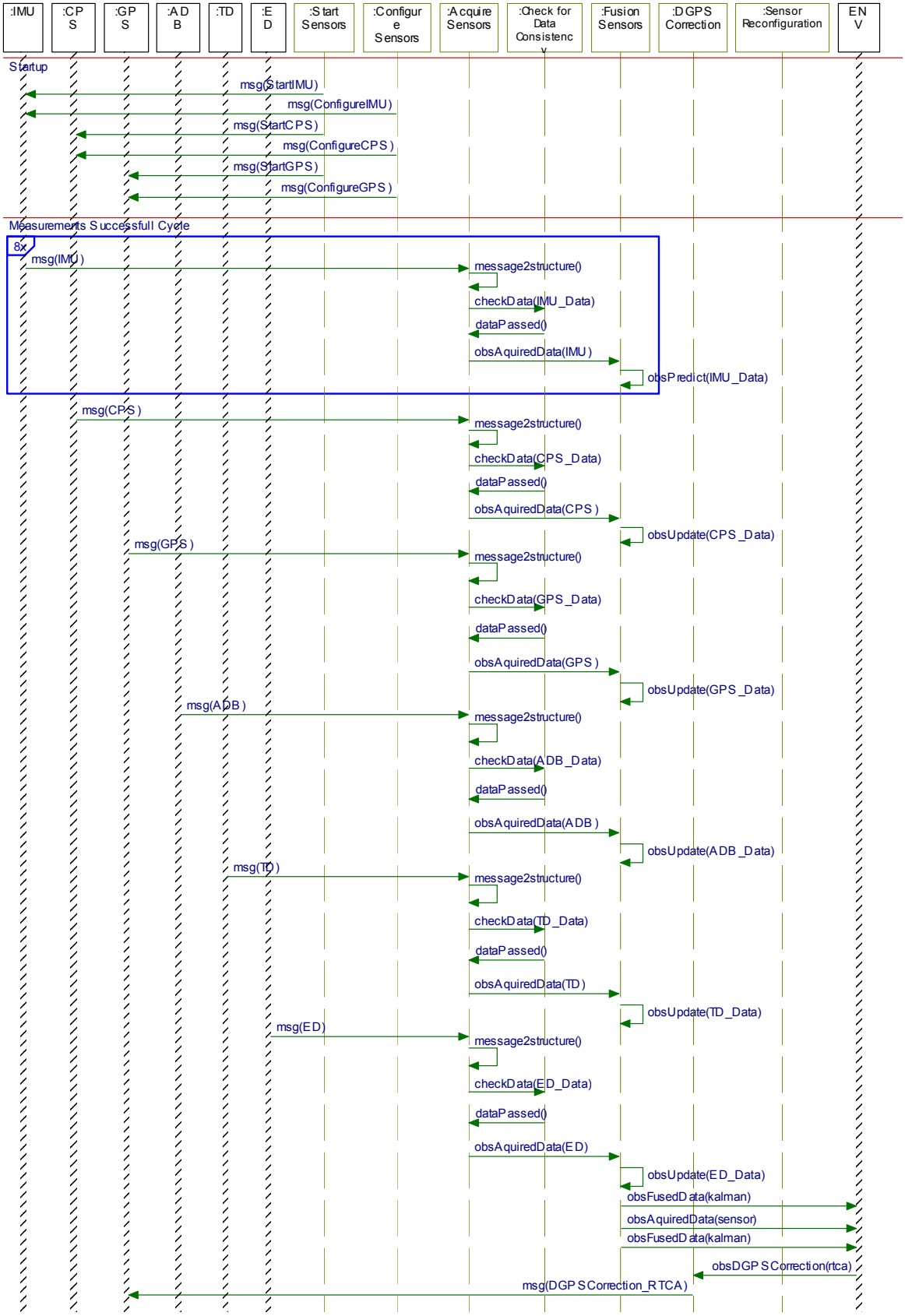

Figura 5-35: UML Sequence - Gerenciamento dos sensores em casos de sucesso e falha.

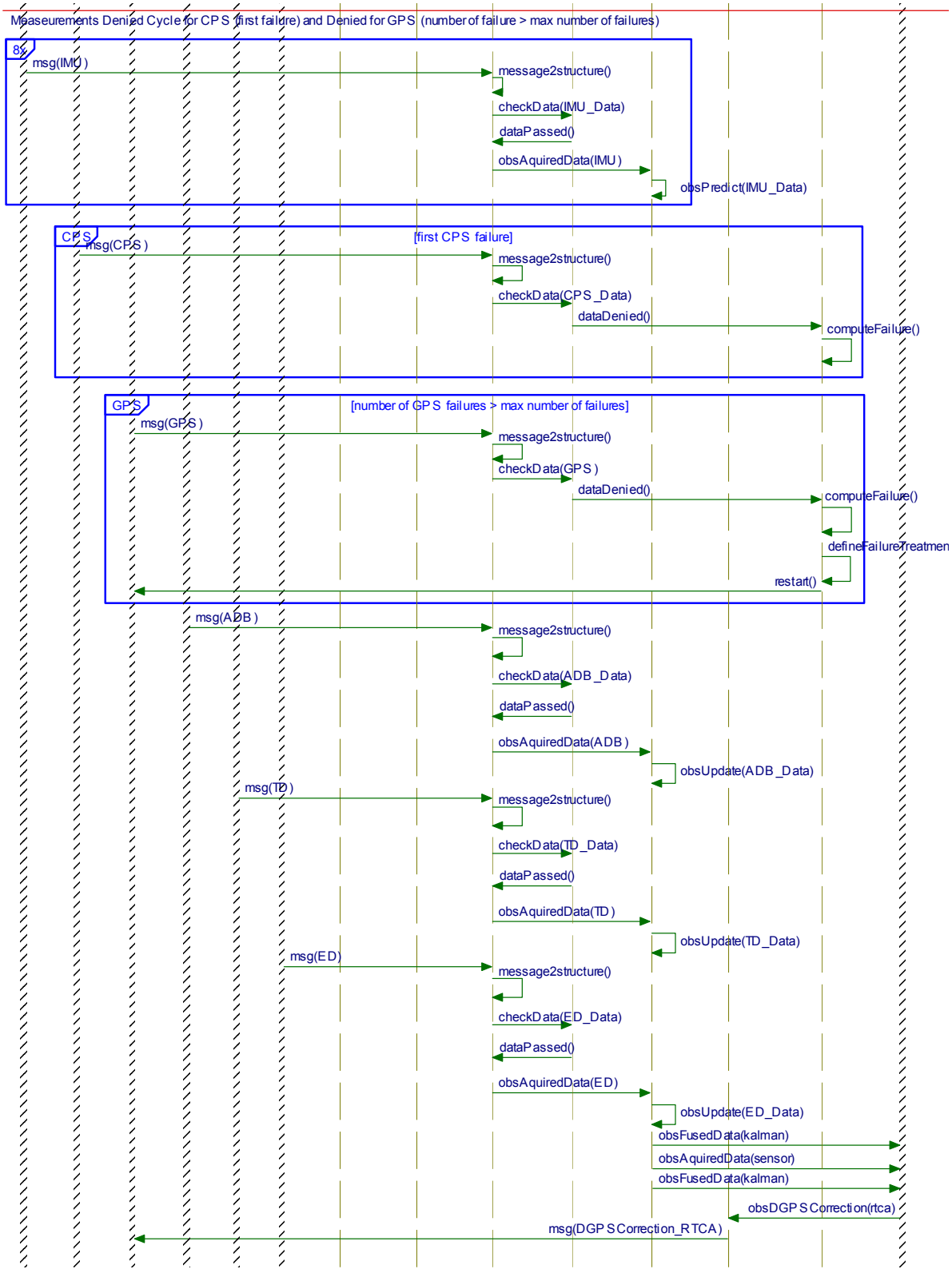

Figura 5-36: UML Sequence - Gerenciamento dos sensores em casos de sucesso e falha (Continuação).

Na Figura 5-37, tem-se o detalhamento do gerenciamento do sistema de comunicação em casos de sucesso, sinal fraco e falha do sinal.

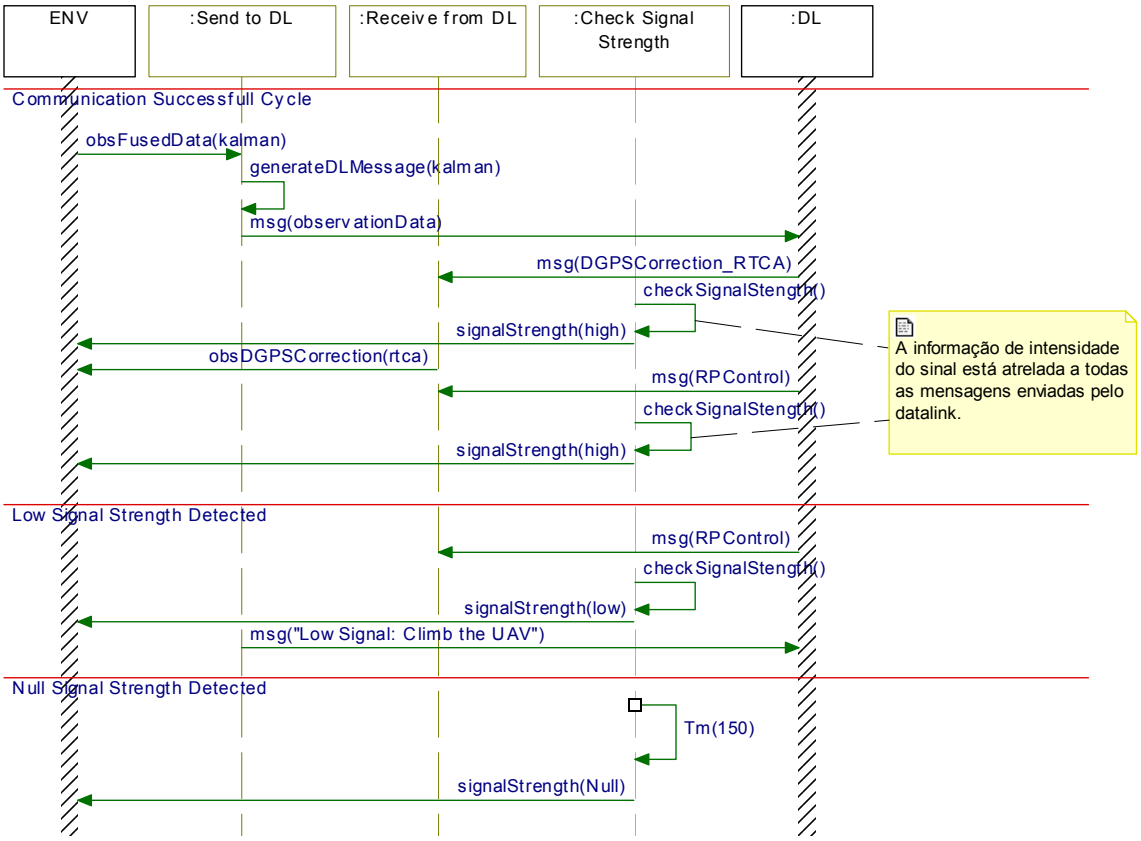

Figura 5-37: UML Sequence - Gerenciamento da comunicação em casos de sucesso e falha.

Na Figura 5-38, tem-se o detalhamento do gerenciamento da atuação em casos de sucesso, requisição de controle do sistema RC-UAV e RC *safe mode*.

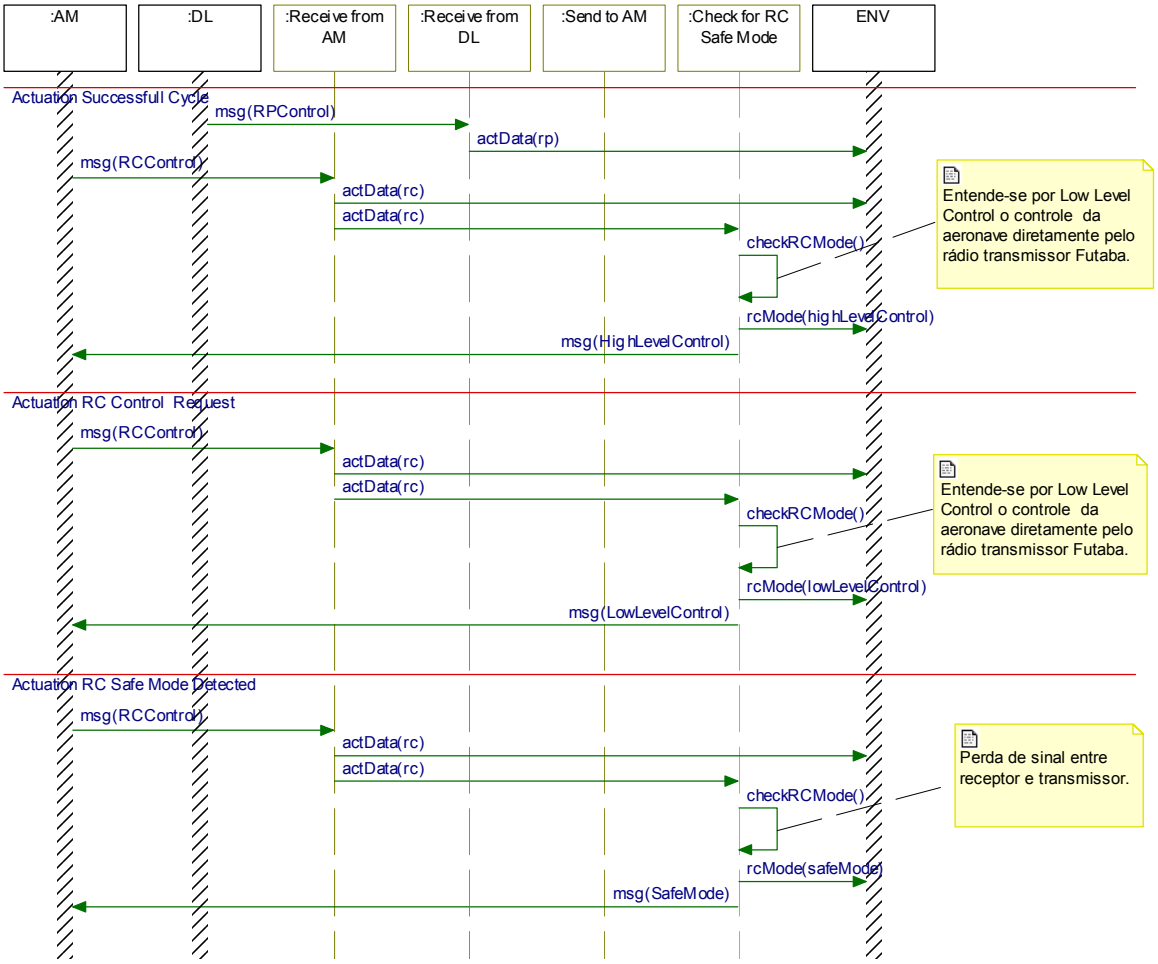

Figura 5-38: UML Sequence - Gerenciamento da atuação em casos de sucesso e falha.

5.4 Considerações finais deste capítulo

Neste capítulo foi realizada a captura de requisitos que o SANT (composto por operador, plataforma, hardware e software) impõe ao sistema aviônico (composto por hardware e software). Os requisitos funcionais foram modelados via um caso de uso: o controle e navegação remotos da aeronave. Já os requisitos temporais foram levantados a partir da dinâmica da plataforma e dinâmica de reação humana, resumindo-se em: freqüência do sistema de 10 Hz e atraso máximo do sistema de 150ms.

Em seguida, foi modelado e simulado a arquitetura e comportamento do sistema aviônico, com base na arquitetura de hardware relatada na seção 3.2. A partir da previsão da carga de processamento de cada um dos módulos que compõem o sistema, foram verificados os requisitos de período e *delay*. Observou-se que todos os elementos possuem carga de processamento inferior a 100%. Isto prevê o atendimento ao requisito de período, uma vez que nenhum elemento precisa de mais tempo de processamento que 100ms. Observou-se que para um atraso máximo de 150ms, os atrasos máximos admissíveis para o computador embarcado e da estação base foram 70ms e 45ms, respectivamente.

De posse dos requisitos, modelo e simulação do sistema, foi possível capturar os requisitos de software. Os requisitos de implantação foram modelados por diagramas de implantação que especificaram em que hardware e sistema operacional o software rodará bem como os *drivers* que o software interagirá. Os requisitos funcionais foram modelados por diagramas de casos de uso que especificaram todas as tarefas que o software deverá realizar. Já os requisitos temporais foram modelados por diagramas de seqüência que especificaram para os casos de uso principais, qual o comportamento que o software deve ter. Assim o software projetado e implementado nos capítulos 6 e 7, respectivamente, deverá atender a estes três grupos de requisitos.

6 PROJETO DO SOFTWARE AVIÔNICO EMBARCADO

6.1 Projeto da arquitetura de software aviônico

A arquitetura de software, basicamente, estabelece a estrutura de relacionamento entre os diversos módulos do software. Assim, revela-se imprescindível que a arquitetura de software deva satisfazer as seguintes características:

- **deliberação e reatividade**: os ambientes operacionais são dinâmicos e ações planejadas devem ser fundidas com reações a situações imprevistas;
- **previsibilidade**: o comportamento do sistema deve ser facilmente previsível a partir de informações de sensores e da missão;
- **tolerância a falhas**: o sistema deve ser robusto às falhas mais comuns;
- **detecção e recuperação de erros**: o sistema deve ser capaz de detectar erros e tentar corrigi-los caso possível, se não, disparar rotinas de emergência e indicar os módulos danificados;
- **programabilidade**: o usuário deve ser provido de ferramentas (geralmente interfaces gráficas) para a especificação e verificação de missões;
- **generalidade**: os módulos devem ser suficientemente genéricos para que possam ser intercambiados entre diferentes veículos;
- **extensibilidade**: capacidade de adaptar-se facilmente à inclusão de novos módulos.

A generalidade e extensibilidade são as características mais importantes para a arquitetura em desenvolvimento, uma vez que a mesma está sendo desenvolvida para ser aplicada não somente em VANTs, mas também em outros robôs móveis como UUVs (*Unmanned Underwater Vehicles*) e UGVs (*Unmanned Ground Vehicles*). Assim a arquitetura proposta neste trabalho foi batizada de GESAM – "*Generic and Extensible Software Architecture for Mobots*".

Nos próximos itens, serão detalhados os meios de atingir as características que a arquitetura de software deve satisfazer. Neste caso, programabilidade será ignorado uma vez que o foco está no desenvolvimento do sistema embarcado e esta característica é intrínseca a estação base.

6.1.1 Arquiteturas deliberativas e reativas

Considerando a classificação proposta por Hall (1992) baseada na divisão de decisão e funções de controle, basicamente, as arquiteturas de software podem ser classificadas em hierárquicas, heterárquicas, e híbridas.

Arquitetura Hierárquica

A arquitetura hierárquica divide os sistemas em níveis. Os níveis superiores são responsáveis pelos objetivos gerais da missão, enquanto os níveis inferiores são responsáveis por solucionar problemas específicos da missão. Consiste de uma estrutura de comunicação serial direta entre dois níveis adjacentes. A representação bem definida da estrutura revela-se a principal vantagem desta arquitetura o que torna mais fácil avaliar seu desempenho. A desvantagem é a falta de flexibilidade e tempo de resposta grande dado que não existe comunicação direta entre os níveis superiores e inferiores e, como conseqüência, o sistema não apresenta comportamento dinâmico real quando exigido em situações imprevistas.

Arquitetura Heterárquica

A arquitetura heterárquica usa uma estrutura paralela onde todos os módulos do sistema podem comunicar-se diretamente entre si, sem níveis intermediários ou supervisores. A flexibilidade revela-se a principal vantagem. A desvantagem é a falta de supervisão, ou seja, problema de controlabilidade, e comunicação intensa entre módulos.

Arquitetura Híbrida

A arquitetura híbrida tenta tirar vantagens dos dois tipos de abordagem anteriormente descritos. De maneira geral, grande parte das pesquisas em arquiteturas de software tem buscado as vantagens da abordagem híbrida como forma de superar as limitações enfrentadas pelas outras abordagens. Dentre os sistemas híbridos, a arquitetura de agentes visa aliar os conceitos de sistemas multi-agentes com a utilização de metodologias de desenvolvimento orientado a objetos para o projeto e implementação.

A GESAM baseia-se na abordagem de arquiteturas de agentes multi-camadas (*layered architectures*) (MÜLLER, 1996) cuja idéia é estruturar o sistema em uma comunidade de agentes, distribuídos em várias camadas de abstração, que interagem entre si de forma a se alcançar comportamentos híbridos de reatividade e deliberação. Os agentes de camadas de abstração maior têm responsabilidades de alto nível como gerenciamento enquanto agentes de camadas de abstração menor têm a responsabilidades de baixo nível como subsistemas de hardware e software.

Definição de Agente adotada na arquitetura GESAM

O conceito de agente é apresentado na literatura de maneira bastante diversificada e está longe de apresentar uniformidade mesmo entre os especialistas. Segundo Franklin (1996) as definições de agentes existentes na literatura especializada surgem do conjunto de exemplos de agentes que é adotado pelos seus respectivos autores, o que explica a diversidade de definições. Evidentemente, este fato não impede que se encontrem idéias comuns ou até mesmo complementares entre as diversas definições existentes.

Ainda em Franklin (1996) e em Ferreira (2003) tem-se uma análise das diversas definições de agentes encontradas na literatura especializada e, a partir destas definições, é

possível enumerar algumas propriedades (Tabela 6-1) que podem ser atribuídas a um agente. Mas Franklin (1996) ressalta que as quatro primeiras características são consideradas as únicas características essenciais a agentes.

Tabela 6-1: Principais características de agentes.

Propriedade	Significado
Autonomia	O agente exerce controle sobre suas próprias ações. Este controle é atingido pela união de encapsulamento, comportamento reativo e deliberativo, interface e concorrência.
Comportamento Reativo	O comportamento do agente reage continuamente a estímulos gerados pelo ambiente.
Comportamento Deliberativo (pro-atividade)	O comportamento do agente é definido por um planejamento baseado em um modelo do ambiente para atingir algum objetivo e não age simplesmente em resposta ao ambiente.
Concorrência (temporalmente contínuo)	Está em contínuo processo de execução.
Encapsulamento	Todas as demais propriedades como atributos, métodos, comportamento e interface estão encapsulados em um corpo coeso.
Interface (sociabilidade)	Permite a comunicação com outros agentes para colaboração.
Mobilidade	Capaz de locomover-se em seu ambiente.
Aprendizagem (adaptabilidade)	Muda seu comportamento baseado em suas experiências prévias.
Caráter	Possui personalidade e estado emocional
Flexibilidade	Não age de forma previsível ou segundo um plano.

Já em Douglass (1999) encontra-se a seguinte definição de agente:

"A característica intrínseca a objetos é a encapsulação (atributos e métodos) e, a adição de comportamento, interface e concorrência permitem a estes atuarem como uma máquina separada e autônoma, mais conhecido como Agente de software. Cada agente não precisa ser muito inteligente, mas deve ter suas responsabilidades e colaborar com outros agentes para atingir metas de maior nível. Entretanto o nível de inteligência permite que o agente tome decisões de forma autônoma escolhendo confiar e cooperar ou não com outros agentes."

Da fusão da definição essencial de agente proposta em Franklin (1996) e Ferreira (2003) (autonomia, comportamento reativo, comportamento deliberativo e concorrência) com a definição de agente proposta por Douglass (1999) (encapsulação, comportamento, interface e concorrência), é possível gerar uma definição consistente com a proposta da GESAM. Assim os agentes da GESAM serão dotados de autonomia, comportamento reativo, comportamento

deliberativo, concorrência, encapsulamento e interface, conforme pode ser observado na Tabela 6-2.

A característica mobilidade corresponde a agentes físicos revelando-se inconsistente com a GESAM, uma vez que esta arquitetura é composta por uma comunidade de agentes de software. Já as características aprendizagem, caráter e flexibilidade entram em conflito com o determinismo que é uma característica básica de sistemas homologáveis, principalmente na área de sistema de tempo real crítico. Como a GESAM é uma arquitetura para sistemas homologáveis, o determinismo é uma característica imprescindível e, desta forma, características imprevisíveis como aprendizagem, caráter e flexibilidade devem ser descartadas.

Tabela 6-2: Principais características de agentes da GESAM.

Propriedade	Significado
Autonomia	Atingida pela união de encapsulamento, comportamento reativo e deliberativo, interface e concorrência.
Comportamento Reativo	O comportamento do agente reage continuamente a estímulos gerados pelo ambiente.
Comportamento Deliberativo	O comportamento do agente é definido por um planejamento baseado em um modelo do ambiente para atingir algum objetivo e não age simplesmente em resposta ao ambiente.
Concorrência (temporalmente contínuo)	Está em contínuo processo de execução.
Encapsulamento	Todas as demais propriedades como atributos, métodos, comportamento e interface estão encapsulados em um corpo coeso.
Interface (sociabilidade)	Permite a comunicação com outros agentes para colaboração.

6.1.2 Arquiteturas confiáveis e seguras

A utilização de arquiteturas confiáveis e seguras (DOUGLASS, 1999; JOHNSON, 1988) garante o atendimento à previsibilidade, tolerância a falhas e detecção e recuperação de erros. Confiabilidade é a medida de disponibilidade e, para aumentar a confiabilidade de um sistema, utiliza-se principalmente arquiteturas com redundâncias, pois, se um componente falhar, outro toma seu lugar garantindo a manutenção do funcionamento com funcionalidades reduzidas. Deste modo, sistemas confiáveis devem continuar a prover alguma funcionalidade mesmo na presença de falhas e,se o sistema falhar seguramente ou não, a confiabilidade do sistema continua a mesma. Assim, um sistema confiável não falha frequentemente, mas se falhar, não garante nenhum comportamento. Já um sistema seguro garante um determinado comportamento em situações de falha, ou seja, garante a manipulação da falha de forma que

todo o sistema se mantenha seguro. Uma forma de aumentar a segurança de um sistema é adotá-lo de formas de detecção e recuperação de erros.

Existem muitos padrões de projeto que afetam ambos, confiabilidade e segurança. Tais padrões são primariamente arquiteturais porque eles afetam mais aspectos de um sistema. Em seguida serão discutidos estes padrões.

Single Channel Protected Designs (SCPD)

Um canal é um caminho estático de dados e controle que pega alguma informação e produz alguma saída. Qualquer falha de qualquer componente do canal gera uma falha no canal inteiro. Na arquitetura SCPD, existe somente um canal para o controle de um mesmo processo. Por exemplo, pode-se considerar um sistema de freio conforme a Figura 6-1.

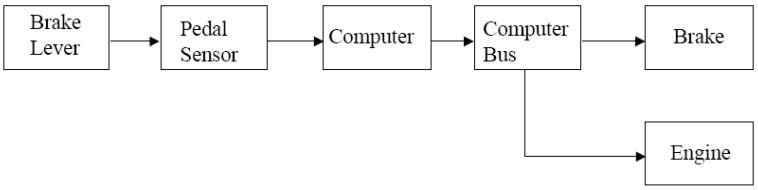

Figura 6-1: Arquitetura de confiabilidade e segurança SCPD (DOUGLASS, 1999).

Dual Channel Designs (DCD)

Uma arquitetura DCD separa o controle da medição, ou provê múltiplos canais de controle independentes. Cada um destes é um canal único que pode falhar. Se o canal de controle falha, o canal de monitoramento alerta o usuário. Se o canal de monitoramento falha o sistema continua controlando de forma apropriada, desde que são canais separados e distintos. Uma condição a garantir é que não haja modos de falha comuns de ambos os canais. Considere um acelerador linear médico, conforme a Figura 6-2.

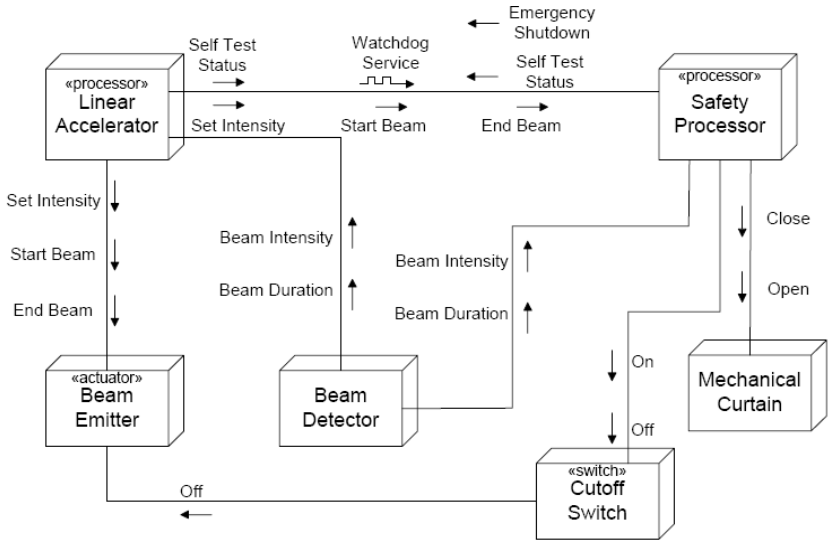

Figura 6-2: Arquitetura de confiabilidade e segurança DCD (DOUGLASS, 1999).

Multi-Channel Voting Pattern (MCVP)

O MCVP é uma arquitetura muito popular em sistemas críticos e seguros. Geralmente utiliza-se um número impar de canais redundantes que votam pela validade do dado entrando ou do controle saindo. Em caso de desacordo, a maioria vence. Estes sistemas podem ser homogêneos ou diversificados, dependendo do custo e do tipo de proteção à falha desejada.

Homogeneous Redundancy Pattern (HRP)

O HRP aumenta a confiabilidade, pois, utiliza canais redundantes e idênticos que simultaneamente desenvolvem a mesma computação. Um controlador então decide como lidar com discrepâncias baseado em uma ou mais políticas de negociação. A estrutura básica da arquitetura é mostrada na Figura 6-3.

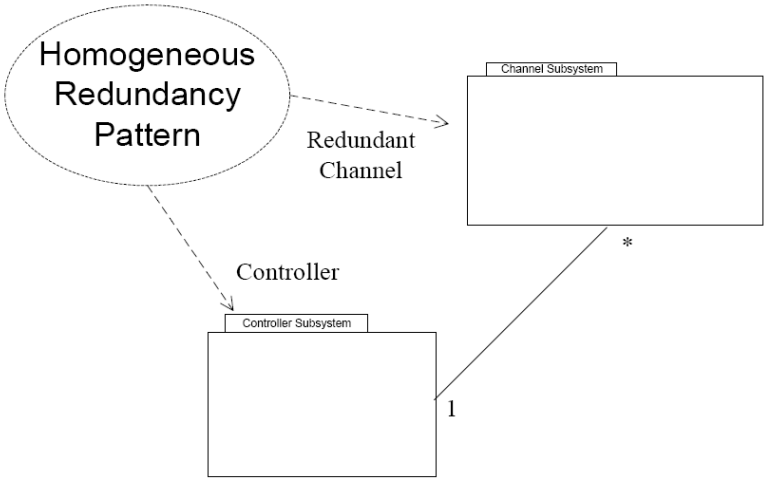

Figura 6-3: Arquitetura de confiabilidade e segurança HRP (DOUGLASS, 1999).

A arquitetura HRP é tolerante a falhas randômicas, como aquelas que ocorrem quando um dispositivo físico falha. Esta também tem a vantagem do baixo custo de implementação, dado que todos os canais são idênticos. Por outro lado, tem a desvantagem dos canais serem sensíveis a erros sistemáticos, pois estes serão comuns a todos os canais.

Diverse Redundancy Pattern (DRP)

A arquitetura DRP provê benefícios adicionais à arquitetura HRP, pois protege contra erros sistemáticos como erros de software. Este benefício é possível, pois a arquitetura provê múltiplos canais que possuem semântica e interface idênticos, porém diferem em termos de implementação, conforme pode ser observado na Figura 6-4. Então na presença de um erro sistemático em um canal, este erro provavelmente não existirá em canais cuja implementação seja diferente.

A arquitetura DRP exige maior custo de implementação, pois os canais devem ser implementados diferentemente. Entretanto oferece proteção adicional contra erros sistemáticos.

Existem muitas variantes de DRP. Em uma variante, um canal é o canal primário de computação e os ademais provêem checagens mais leves no primeiro. Outra variante é a arquitetura Monitor-Actuator Pattern, discutido a seguir.

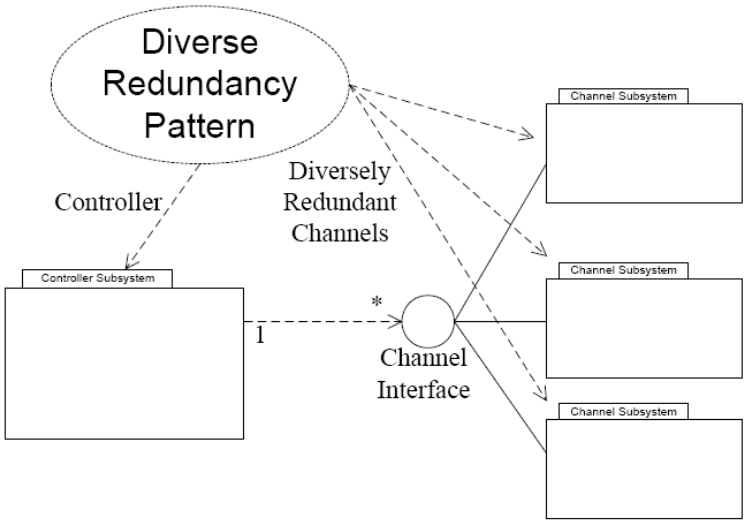

Figura 6-4 Arquitetura de confiabilidade e segurança DRP (DOUGLASS, 1999).

Monitor-Actuator Pattern (MAP)

A arquitetura MAP é uma variante da DRP, conforme pode ser observado na Figura 6-5. O conceito básico da MAP é que um canal controla o sistema de atuação enquanto outro canal verifica a sua consistência. Se o canal de atuação falha, o canal monitor adverte esta falha ao controlador. Se o canal de monitoramento falha, então o canal de atuação continua a trabalhar corretamente. O canal monitor deve então ser conferido com o objetivo de garantir que este canal está adequado para a função de segurança. Isto é usualmente feito com a combinação de *Power-On Self Tests* (POSTs), *Built-In Tests* (BITs), e troca de mensagens de vida entre os canais do atuador e do monitor.

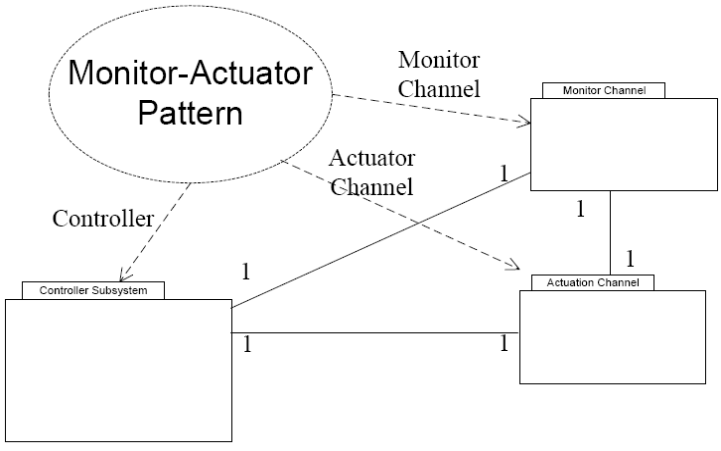

Figura 6-5: Arquitetura de confiabilidade e segurança MAP (DOUGLASS, 1999).

Safety Executive Pattern (SEP)

A última arquitetura considerar é a SEP que é uma arquitetura de larga escala. Esta consiste de um subsistema controlador chamado *Safety Executive*, um ou mais subsistemas *watchdog* que verificam a saúde do sistema garantindo que uma apropriada atuação está ocorrendo, um ou mais canais de atuação, e um subsistema de recuperação de falhas chamado *Fail-Safe Processing Channel*. Na Figura 6-6 pode-se observar uma representação da SEP, que é particularmente apropriada quando:

- Um conjunto apropriado de estados *fail-safe* a serem entrados quando falhas são identificadas;
- A determinação das falhas é complexa;
- Muitas ações relacionadas à segurança são controladas simultaneamente;
- Ações relacionadas com segurança não são independentes;
- A determinação da ação de segurança apropriadas em um evento de falha pode ser complexa.

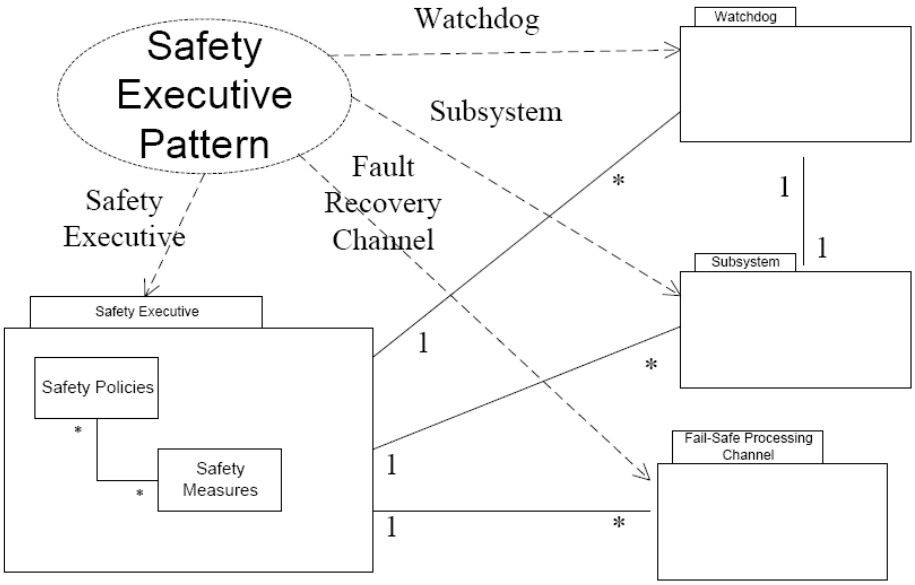

Figura 6-6: Arquitetura de confiabilidade e segurança SEP (DOUGLASS, 1999).

Como solução de segurança e confiabilidade, será utilizado na arquitetura GESAM um híbrido dos padrões HRP, DRP e SEP. O objetivo é desenvolver uma arquitetura de software reconfigurável em tempo de execução de forma a suportar significativa degradação, onde falhas causam redução da qualidade ou capacidade do sistema de controle, mas não resulta em falha total do sistema, como observado em Rawashdeh (2005).

6.1.3 Arquitetura Genérica e Extensível para Mobots

Neste trabalho, é proposta uma nova arquitetura, dentre os trabalhos analisados, baseada em generalidade e extensibilidade aplicável em robôs móveis (GESAM) que também

atende às demais características desejáveis em arquiteturas de software: reatividade-deliberação e segurança-confiabilidade.

A característica básica da GESAM é a estruturação em camadas de abstração. Na (Figura 6-7) pode-se observar as camadas utilizadas no sistema aviônico. A idéia é que camadas superiores tenham menor número de agentes com maior abstração e complexidade, já camadas inferiores tenham maior número de agentes com menor abstração e complexidade. A camada *System* é preferencialmente imutável, pois cria a base para todo o sistema. Desta forma o sistema deve crescer a partir desta camada. Obviamente quanto mais inferior a camada, maior a quantidade de mudanças previstas. Na Figura 6-8 pode-se observar o projeto da camada *System* para o sistema aviônico.

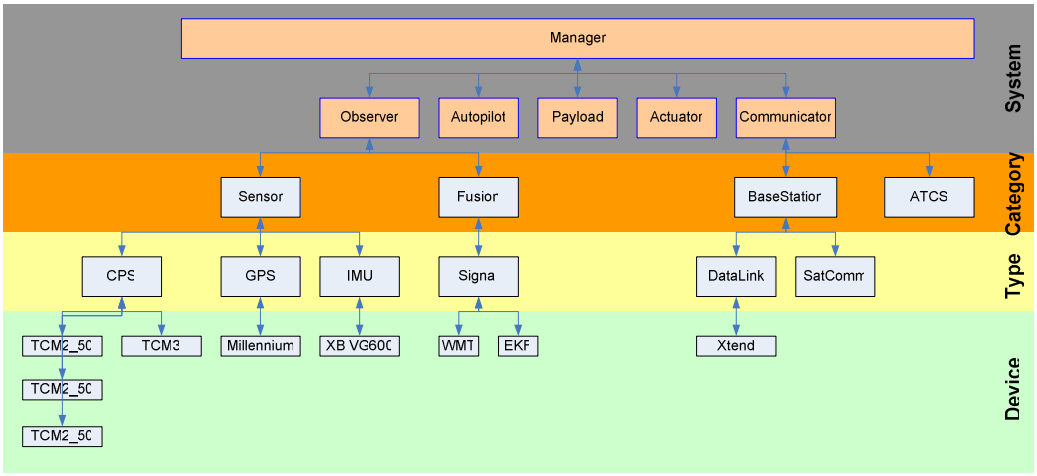

Figura 6-7: Camadas da arquitetura de software (System, Category, Type e Device).

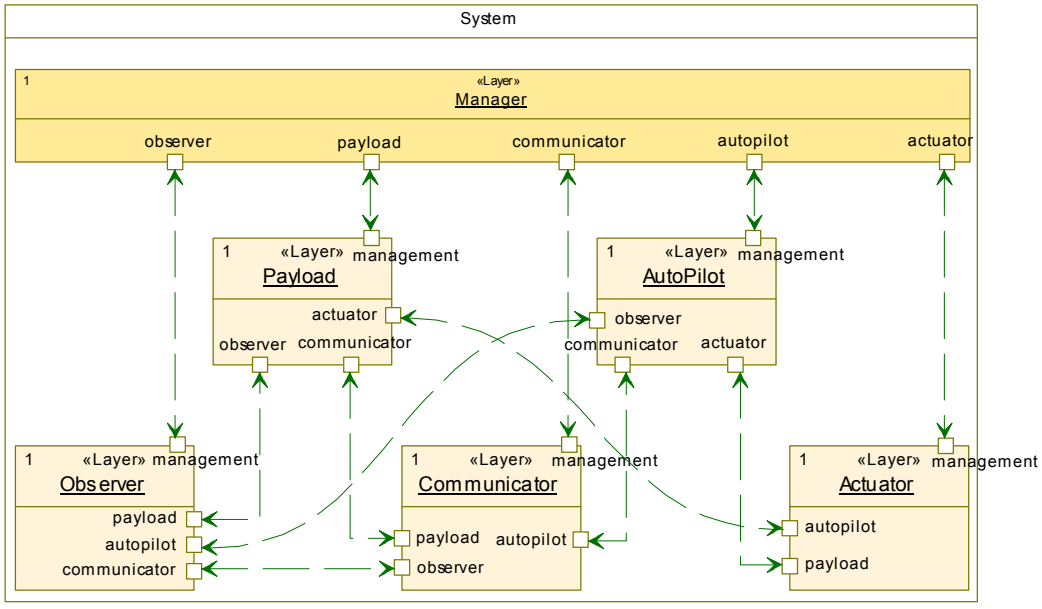

Figura 6-8: UML Structure - Camada base arquitetura GESAM.

As camadas *Category, Type, Device* são flexíveis, entretanto devem respeitar um padrão de projeto que abstrai a camada imediatamente inferior. Basicamente a presente camada possui um agente **Manager** que se conecta às camadas inferiores que estão dentro da camada atual. Como exemplo, tem-se o compósito **Sensor** (Figura 6-9) que está na camada *Category*. Um compósito é a mesma coisa que uma *composite class* que agrupa diversas classes e, um compósito junto com o seu respectivo *manager*, na GESAM, também é considerado um agente. Dentro de sensores tem GPS e IMU (está dentro de sensor porque sensor é mais abstrato que GPS e IMU). Já o **SensorManager** abstrai de agente **GPS** e **IMU** para agente **Sensor**. A abstração inclui estrutura de dados (de dados específicos de GPS para dados mais gerais de sensor), detecção e tratamento de erros, entre outros.

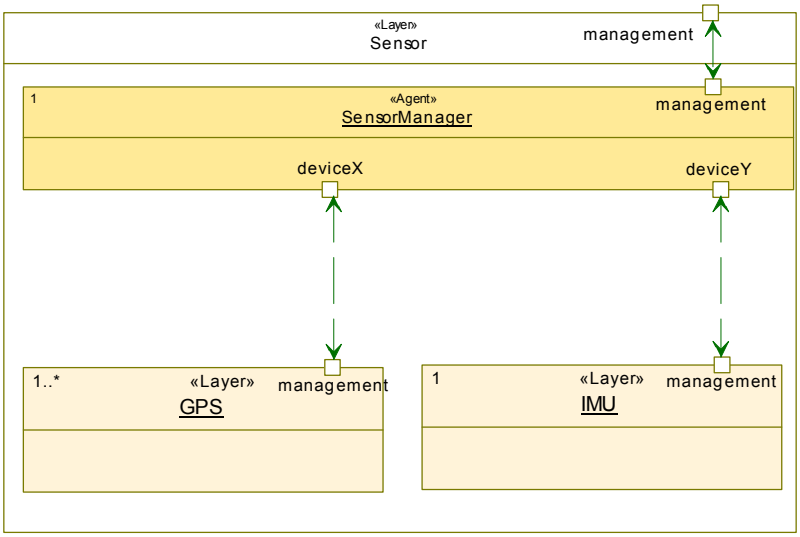

Figura 6-9: UML Structure - Padrão de abstração da arquitetura GESAM.

Na Figura 6-10 tem-se outro exemplo do padrão de abstração. O compósito **CPS** possui os compósitos **TCM2-50** e **TCM3**. Neste caso, **TCM2-50** e **TCM3** não são mais compósitos, pois não há como abstrair mais, o que evidencia o nível de dispositivo.

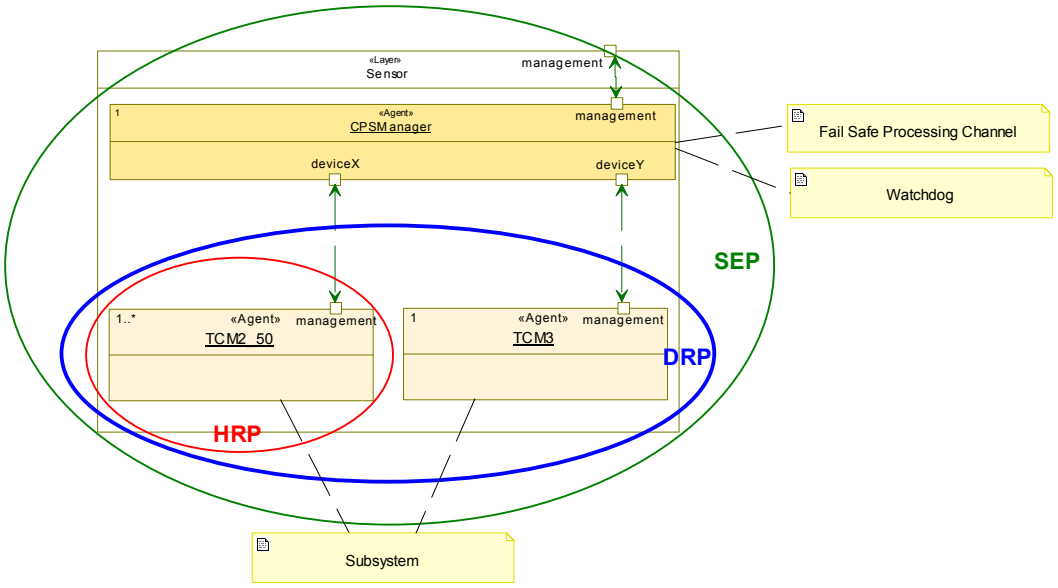

Figura 6-10: UML Structure - Segurança e confiabilidade do padrão de abstração da arquitetura GESAM.

Ainda na Figura 6-10 tem-se uma análise da segurança e confiabilidade do padrão de abstração proposto. O agente **TCM2-50** pode ter redundância n, caracterizando HRP. O agente **TCM2-50** junto com o agente **TCM3** caracteriza DRP. E finalmente, os agentes **TCM2-50** e **TCM3** junto com o agente **CPSManager** caracteriza SEP. O processamento de *FailSafe* da SEP é executado pelo comportamento padrão (Figura 6-11) do agente **Manager**. Este comportamento padrão estende a todos os agentes do sistema.

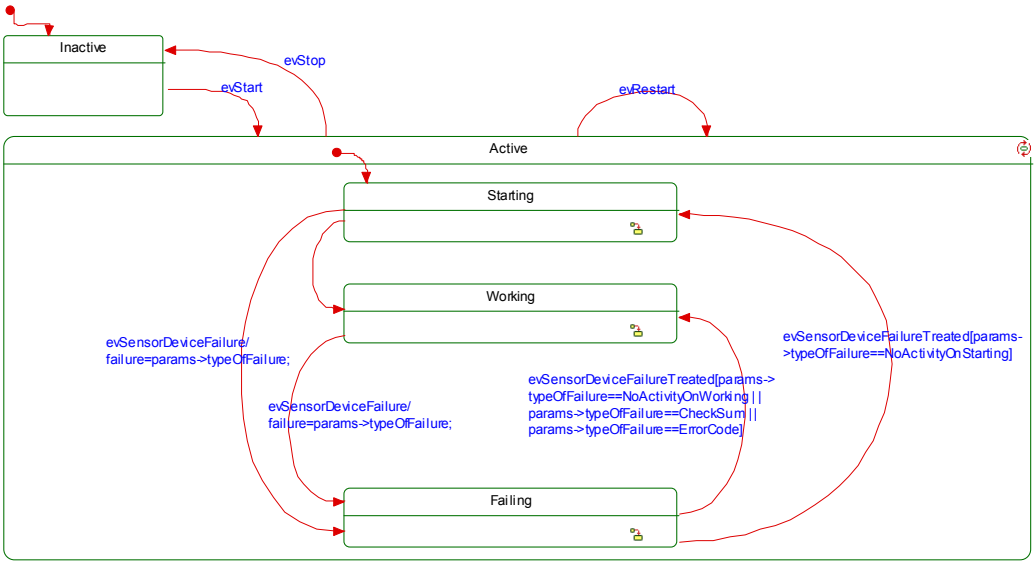

Figura 6-11: UML Statechart - Base comportamental dos agentes.

6.2 Realização da Arquitetura GESAM

Esta seção corresponde à fase de projeto mecanicista do processo Harmony. Esta fase consiste na realização da arquitetura projetada na seção anterior por meio de diagramas de estruturas atendo-se aos agentes do sistema bem como às interfaces de comunicação entre eles. Antes disso, definem-se os pacotes básicos do software e suas relações. Na Figura 6-12 pode-se observar um modelo que representa onde serão armazenados os requisitos do sistema (*_Requirements*) definidos na seção 5.3, onde será armazenada a arquitetura do sistema realizada nesta seção (*Architecture*), onde serão armazenados os objetos que são instanciados pela arquitetura (*Implementation*), e onde residirão os testes realizados na arquitetura e implementação por meio da cobertura dos requisitos.

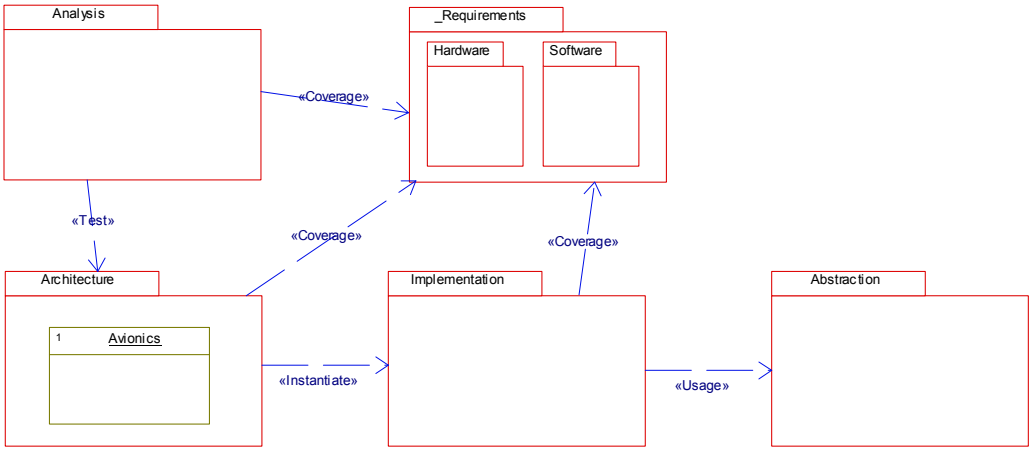

Figura 6-12: UML Object Model - Organização do software.

A arquitetura terá sua realização embarcada no computador de testes no qual roda o RTOS QNX. Ela acessará os arquivos de *log* de sistema e dados, bem como acessará os *drivers* de blocos de memória RAM (devb_ram), *drivers* de blocos de disco rígido (devb_eide), *drivers* de memória flash (devf_generic), *drivers* de caractere via porta serial (devc_ser8250), *drivers* de rede *ethernet* (devn_speedo_so e devn_el900_so) conforme o diagrama de componentes da Figura 6-13.

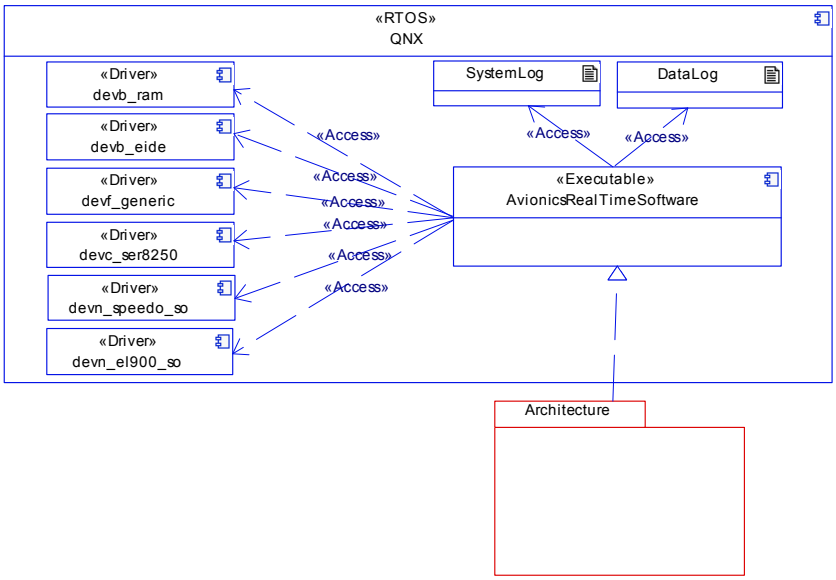

Figura 6-13: UML Component - Componentes do software.

A realização da camada base da arquitetura GESAM pode ser observada na Figura 6-14. Nela observam-se todas os agentes básicos (**Manager**, **Payload**, **Autopilot**, **Observer**, **Communicator** e **Actuator**) bem como as portas e contratos de comunicação e seus modelos de objetos apresentados nas Figura 6-15, Figura 6-16 e Figura 6-17.

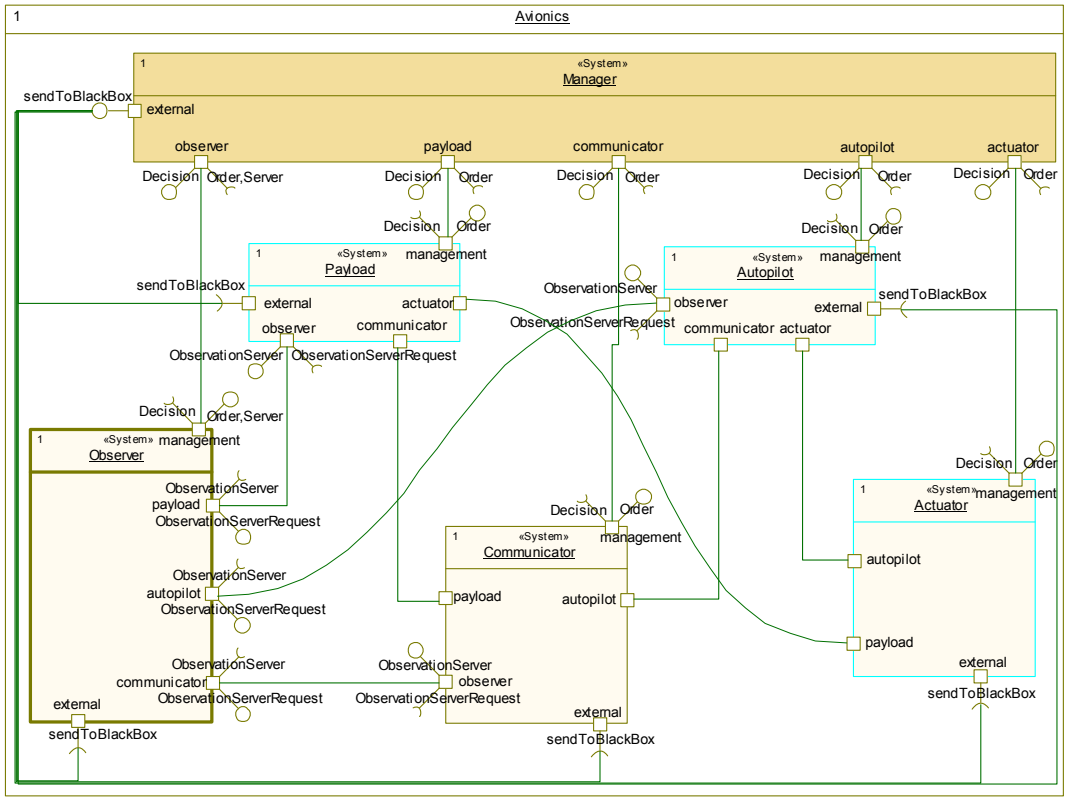

Figura 6-14: UML Structure - Visão geral da Arquitetura Aviônica.

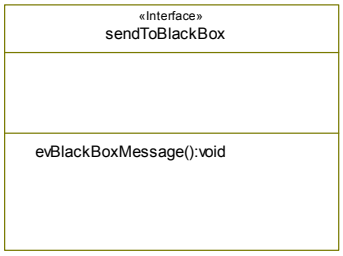

Figura 6-15: UML Interface - Contrato de envio para o BlackBox.

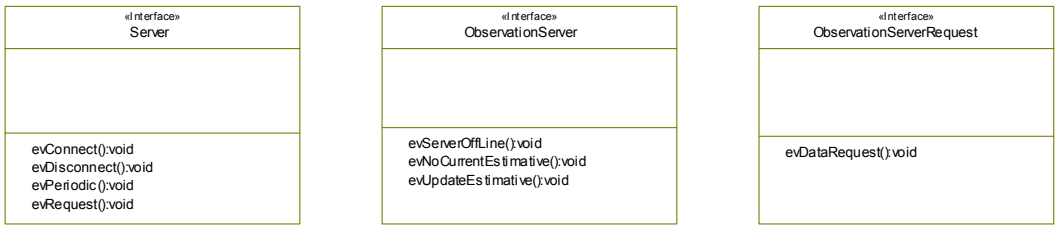

Figura 6-16: UML Interface - Contratos do servidor de observação.

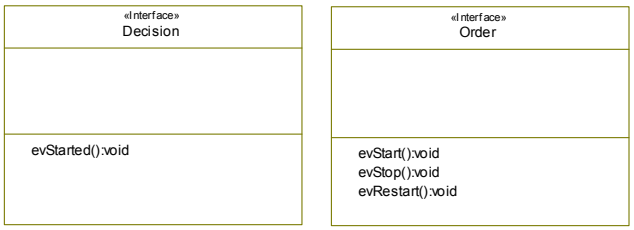

Figura 6-17: UML Interface - Contratos de gerenciamento.

6.2.1 Agente Manager

O agente **Manager,** que é composto pelo compósito **Manager** e pelo objeto ativo **ManagerManager** (Figura 6-18) é o líder de toda a arquitetura e sua função é gerenciar (inicialização, monitoramento e tratamento de erros) seus subordinados, como todo *manager* da arquitetura. Neste caso específico o agente **Manager** possui uma caixa preta na qual todos os demais agentes guardam informações de sistema e de dados.

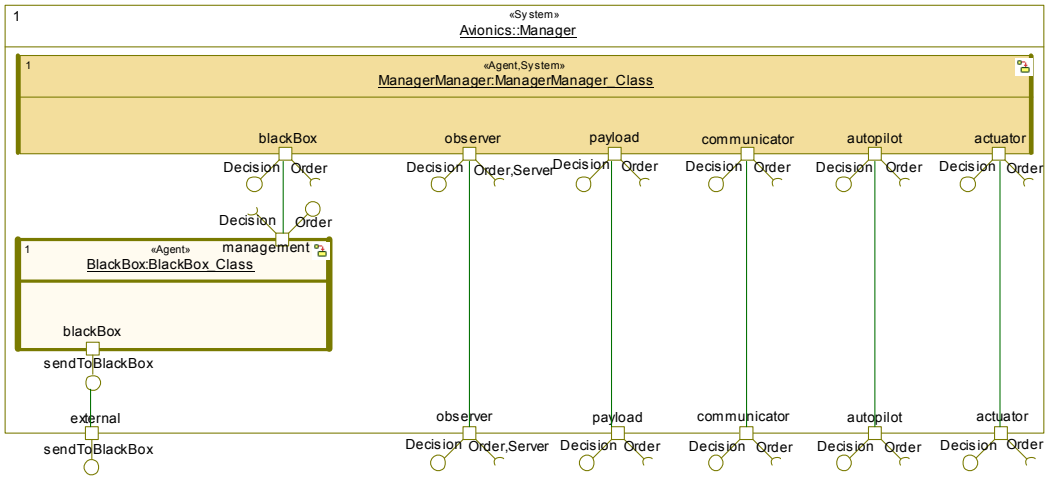

Figura 6-18: UML Structure - Gerenciamento do sistema.

6.2.2 Agente Observer

O agente **Observer** (Figura 6-19), que é composto pelo compósito **Observer** e pelo objeto ativo **ObserverManager**, gerencia todos os sensores bem como a fusão das informações geradas pelos mesmos. O mesmo possui também interfaces de transferência de dados e de detecção e tratamento de erros de sensores (Figura 6-20) bem como transferência de dados de observação (Figura 6-21).

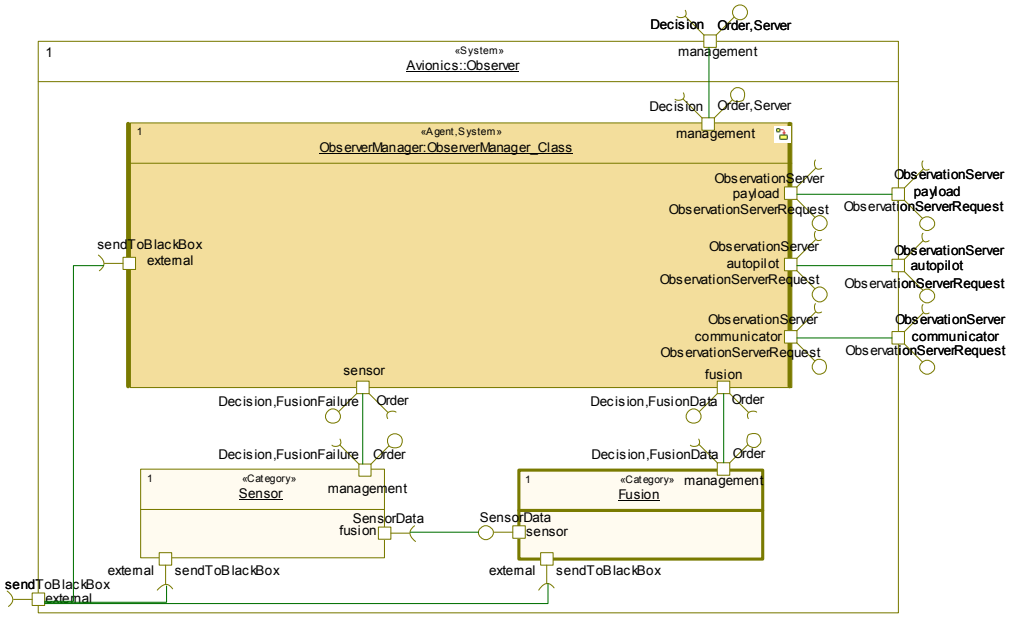

Figura 6-19: UML Structure - Agente Observer.

Figura 6-20: UML Interface - Contratos com sensores.

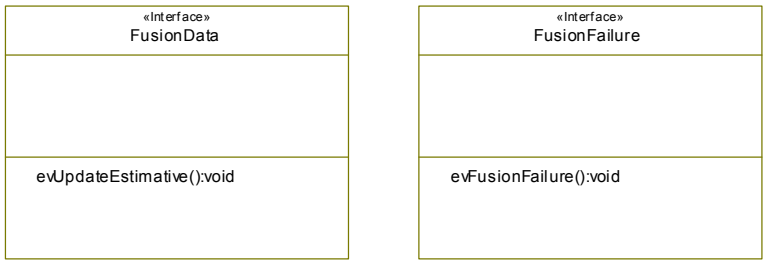

Figura 6-21: UML Interface - Contratos de fusão.

O agente **Sensor** (Figura 6-22) agrupa todos os tipos de sensores entre eles: unidades de medição inercial (agente **IMU**), sistema de posicionamento global (agente **GPS**), bússola (agente **CPS**), dados do ar (agente **ADB**), dados de propulsão (agente **TD**), dados de energia (agente **ED**), dados de vídeo no espectro visível (agente **EO**), dados de vídeo no espectro infravermelho (agente **IR**), entre outros. A interface de tipo (Figura 6-23) tem a função de transferir informação de tipo de sensor para o **SensorManager** abstrair informações de "tipo de sensor" para "sensor".

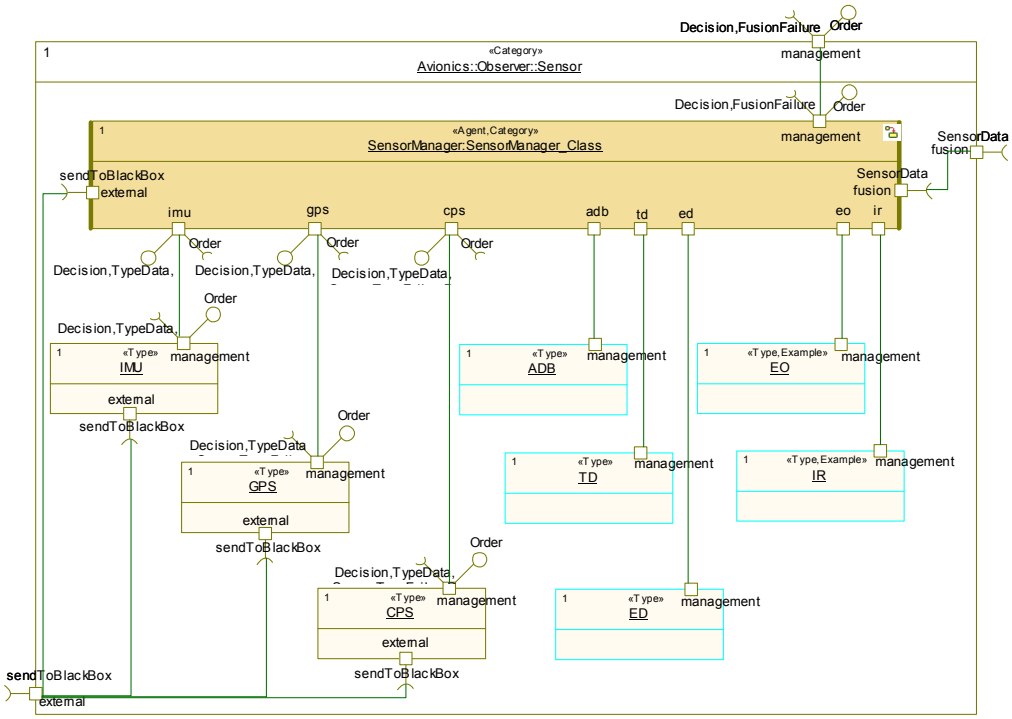

Figura 6-22: UML Structure - Agente Sensor.

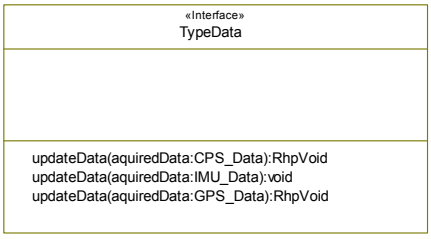

Figura 6-23: UML Interface - Contrato de dados de tipo.

O agente **CPS** (Figura 6-24) agrupa todos os dispositivos do tipo CPS, ou seja, no caso dos recursos de hardware disponíveis, a bússola modelo TCM2_50 (agente **TCM2_50**) e modelo TCM3 (agente **TCM3**), por exemplo. Como padrão de apresentação, adotou-se que todos os objetos cuja borda é azul ciano são exemplos e não foram implementados. A interface de dispositivo (Figura 6-25) tem a função de transferir informação de dispositivo CPS para o **CPSManager** abstrair informações de "dispositivo de CPS" para "CPS". As mesmas

considerações se aplicam à IMU (Figura 6-26 e Figura 6-27) bem como ao GPS (Figura 6-28 e Figura 6-29).

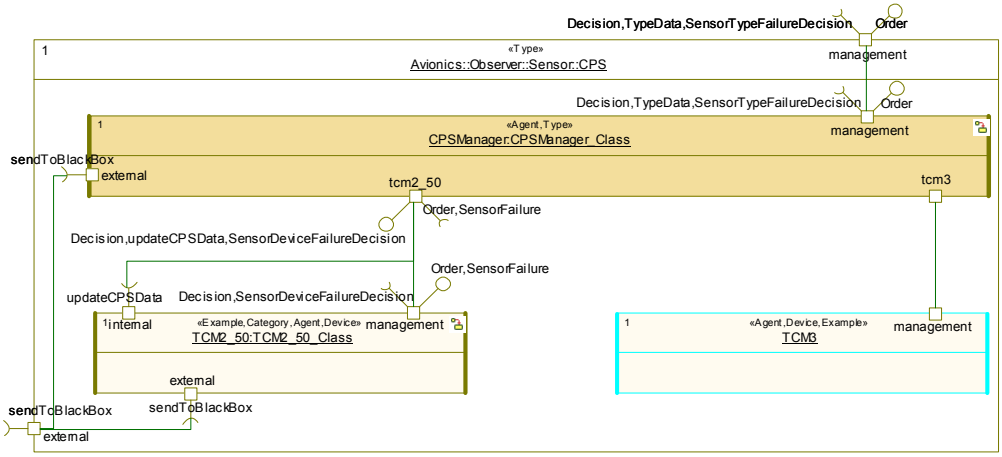

Figura 6-24: UML Structure - Agente CPS e Agente TCM2-50.

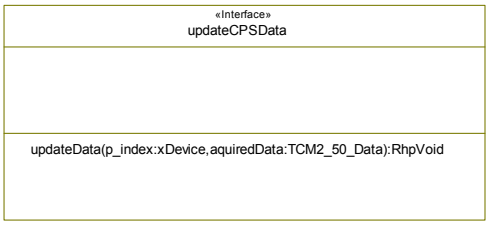

Figura 6-25: UML Interface - Contrato de dados de CPS.

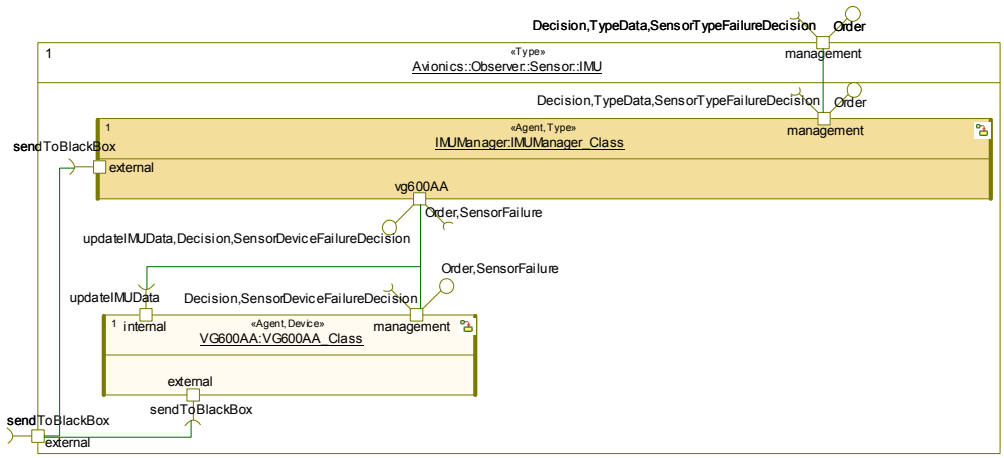

Figura 6-26: UML Structure - Agente IMU e Agente VG600AA.

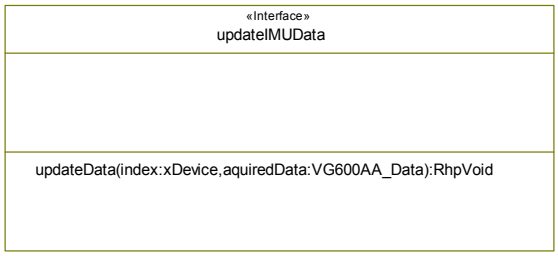

Figura 6-27: UML Interface - Contrato de dados de IMU.

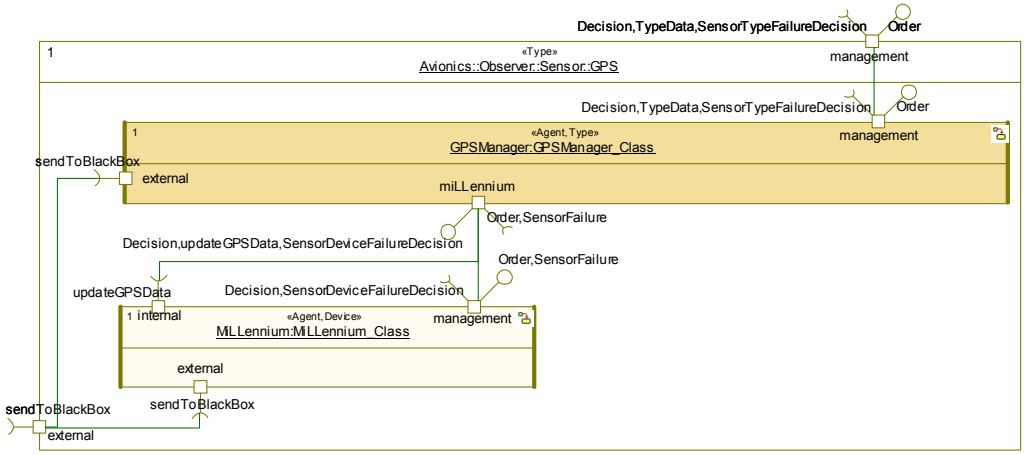

Figura 6-28: UML Structure - Agente GPS e Agente Millennium.

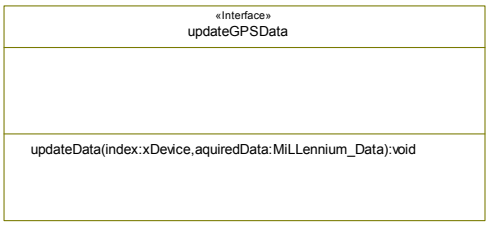

Figura 6-29: UML Interface - Contrato de dados de GPS.

A fusão sensorial da informação adquirida por múltiplos sensores ou por um único sensor em um determinado instante pode ser efetuada em diferentes níveis de representação. Uma categorização usual é considerar o processo de fusão sensorial dividido em quatro níveis (LUO, 1995): nível de sinal, ponto, característica e símbolo. A implementação do agente **Fusion** baseou-se na proposta de Luo (1995) como pode ser observado na Figura 6-30.

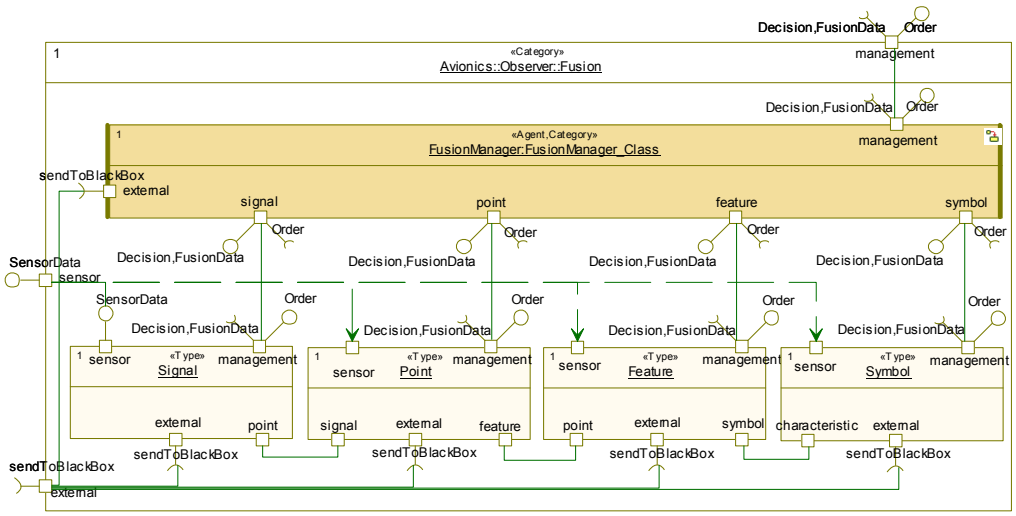

Figura 6-30: UML Structure - Agente Fusion.

A grande maioria dos sensores utilizados na prática fornece informação que pode ser combinada em um ou mais destes níveis, podendo envolver a utilização de diversas técnicas. Estas técnicas são frequentemente aplicadas em conjunto ou sequencialmente. A Figura 6-31 apresenta um exemplo de como pode ser aplicada à fusão sensorial em diferentes níveis.

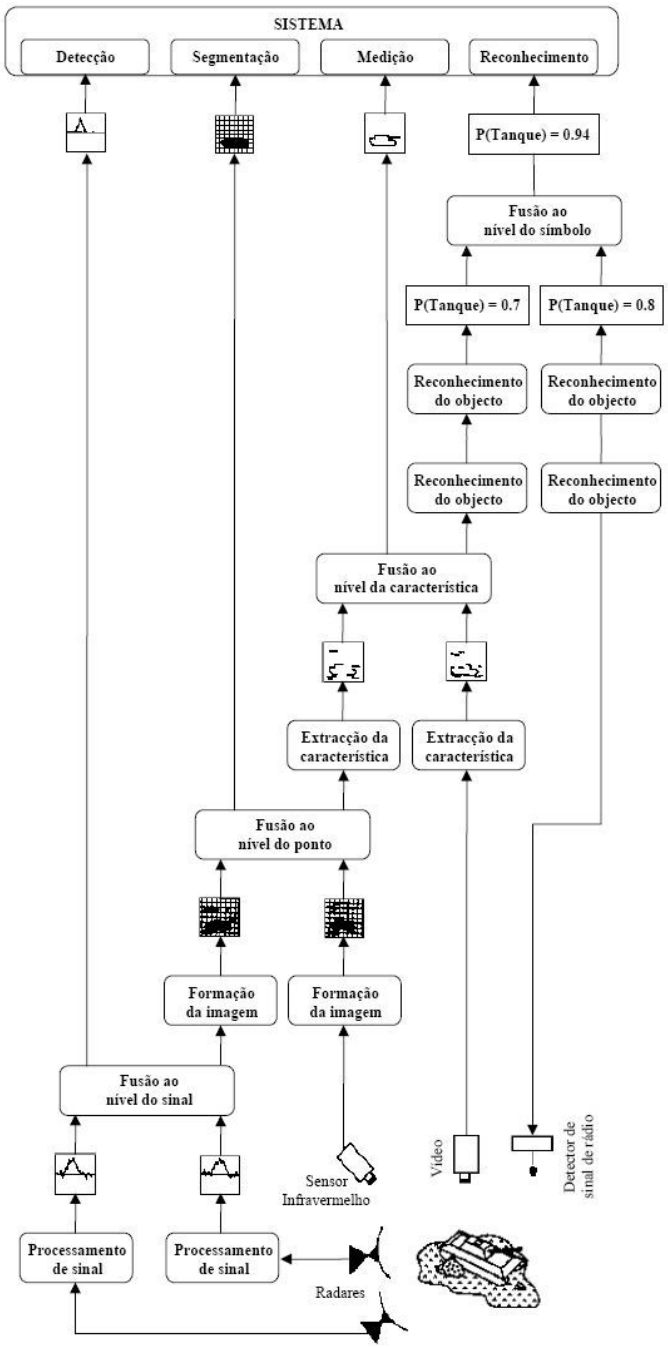

Figura 6-31: Diferentes níveis de fusão sensorial (sinal, ponto, característica e símbolo) no reconhecimento automático de um tanque.

A fusão no nível de sinal, realizada pelo agente **Signal** (Figura 6-32), refere-se à combinação dos sinais provenientes de um grupo de sensores com o objetivo de fornecer um sinal que é usualmente do mesmo tipo dos originais, mas com melhor qualidade. O filtro de médias ponderadas pelo tempo (WMT) e o filtro de kalman (EKF) são exemplos de fusão no nível de sinal e estes dois foram implementados neste trabalho. Para tanto, se estabeleceu uma interface de condições iniciais Figura 6-33.

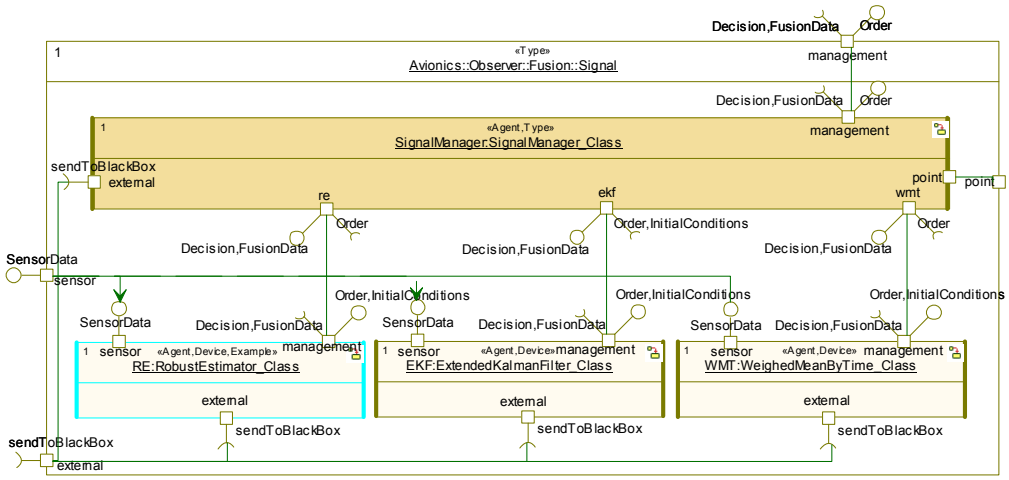

Figura 6-32: UML Structure - Agente Signal, EKF e WMT.

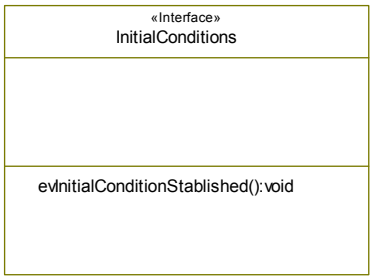

Figura 6-33: : UML Interface - Contrato de condições iniciais dos estimadores de estado.

A fusão no nível de ponto, realizada pelo agente **Point** (Figura 6-34), é exclusivamente utilizada para fornecer uma melhoria no desempenho de algumas tarefas de processamento de imagem.

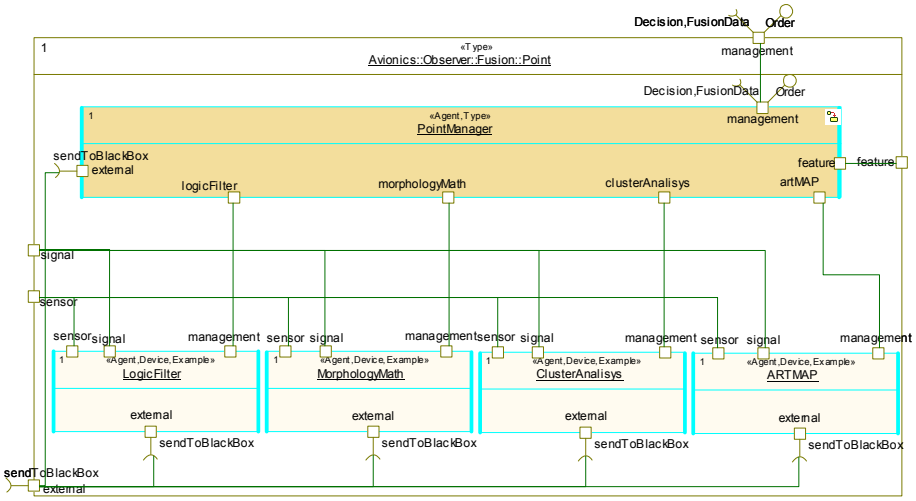

Figura 6-34: UML Structure - Agente Point.

A fusão no nível de característica, realizada pelo agente **Feature** (Figura 6-35), pode ser utilizada com duas finalidades: para incrementar a probabilidade da característica extraída a partir da informação fornecida por um sensor corresponder a um aspecto importante do ambiente; ou como meio de gerar características compostas adicionais para posterior utilização do sistema. Características típicas extraídas a partir de uma imagem e utilizadas no processo de fusão são os contornos e regiões de similar intensidade e profundidades.

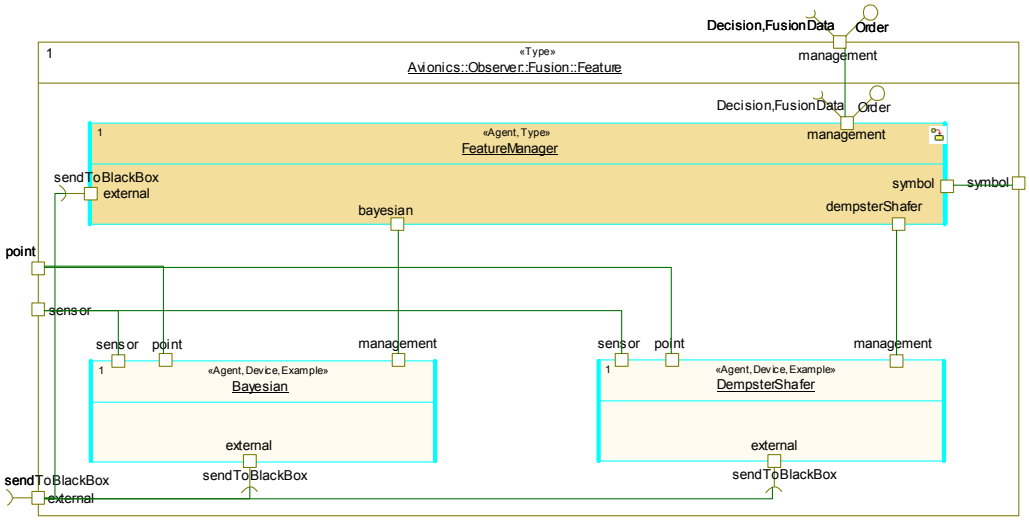

Figura 6-35 UML Structure - Agente Feature.

A fusão no nível de símbolo, realizada pelo agente **Symbol** (Figura 6-36), permite que a informação de vários sensores possa ser utilizada com um alto nível de abstração.

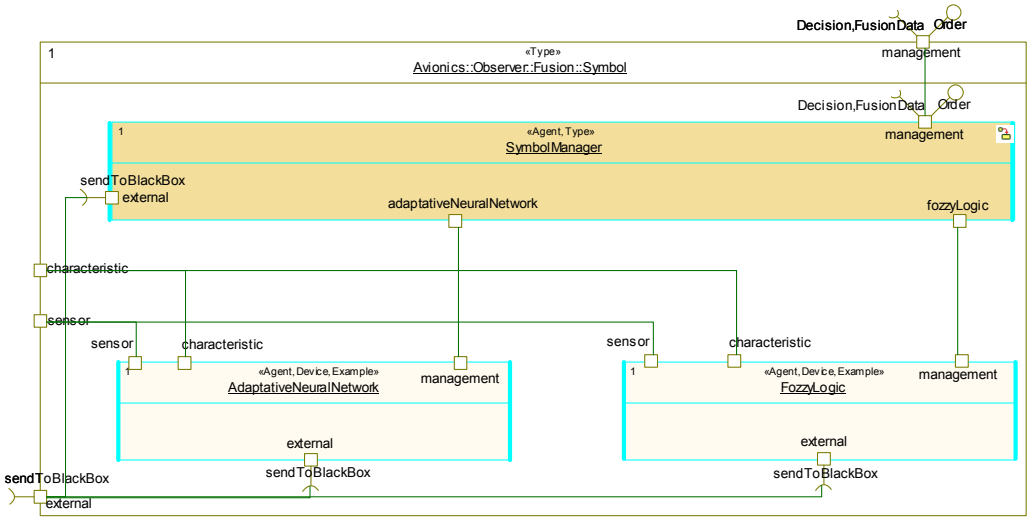

Figura 6-36: UML Structure - Agente Symbol.

Este tipo de fusão pode ser somente um meio de combinar a informação sensorial quando os sensores são muito diferentes ou quando observam diferentes características do ambiente. As decisões são normalmente tomadas com base na comparação dessas

características com a informação tirada do modelo. Os símbolos usados tem normalmente associado a ele um grau de crença que indica o peso que essa informação sensorial tem no modelo.

6.2.3 Agente Communicator

O agente **Communicator** (Figura 6-37) gerencia todas as formas de comunicação do VANT com o meio externo, mais especificamente, comunicação com a estação base (agente **BaseStation**), com o sistema de controle de tráfego aéreo (agente **ATCS**) e com outros agentes físicos de forma a trabalharem de forma coordenada para atingir objetivos em comum (agente **Coordination**).

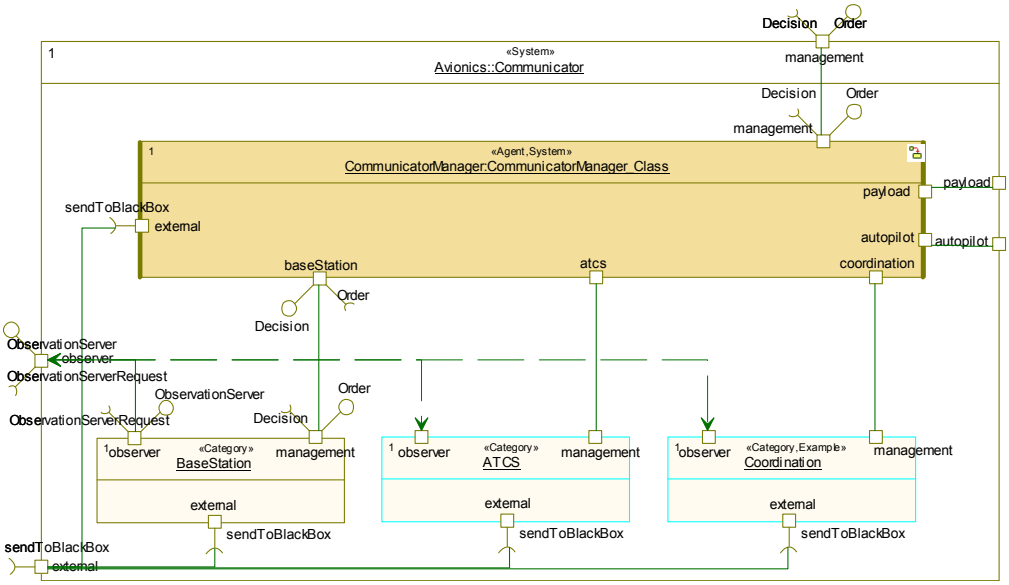

Figura 6-37: UML Structure - Agente Communicator.

O agente **BaseStation** (Figura 6-38) gerencia todos os tipos de comunicação com a estação base. Estas comunicações podem ser realizadas por link direto (agente **DataLink**), por meio de satélite (agente **Satélite**) ou por meio da rede de celular GPRS (agente **GPRS**).

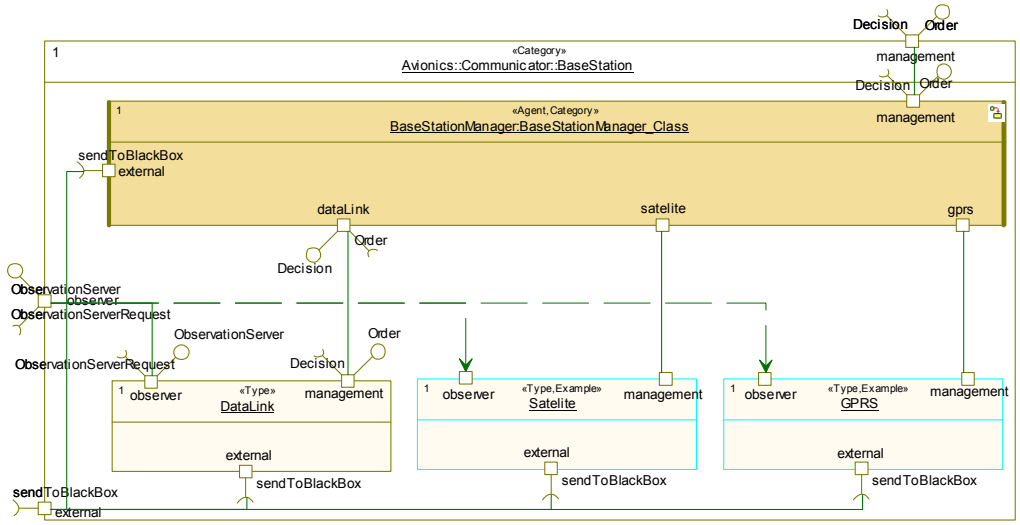

Figura 6-38: UML Structure - Agente Base Station.

O agente **DataLink** (Figura 6-39) gerencia todos os dispositivos de comunicação do tipo *data-link*. Neste projeto utilizou-se o *data-link* da Xtend (agente **Xtend**).

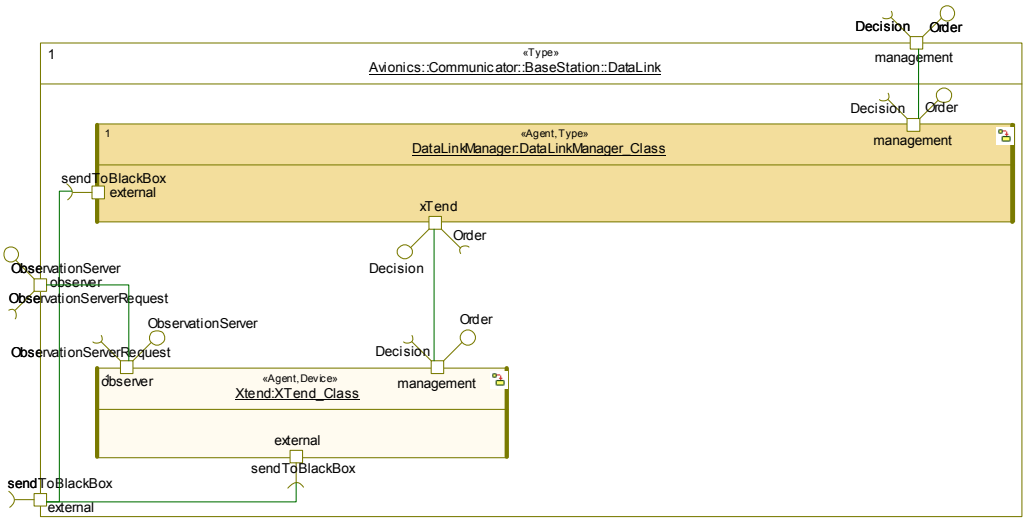

Figura 6-39: UML Structure - Agente Data Link e Xtend.

O agente **ATCS** (Figura 6-40) gerencia todos os tipos de comunicação com o sistema de controle de tráfego aéreo. Estas comunicações são realizadas principalmente por meio de *transponder* (agente **Transponder**).

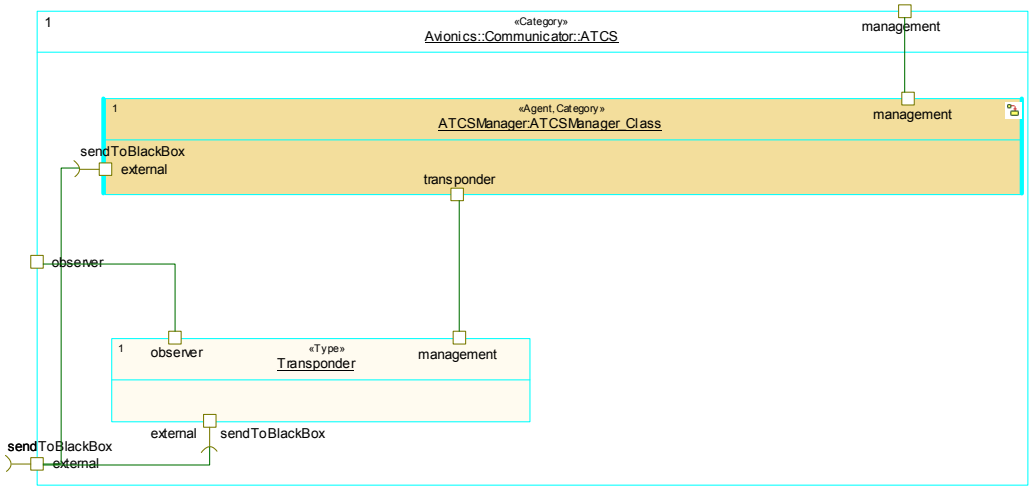

Figura 6-40: UML Structure - Agente ATCS.

O agente **Transponder** (Figura 6-41) gerencia todos os dispositivos de comunicação do tipo *Transponder*. Embora não tenha sido implementado nenhum agente de dispositivo **Transponder**, na aviação geralmente utiliza-se os *transponders* tipo A e C (agentes **TypeA** e **TypeC**).

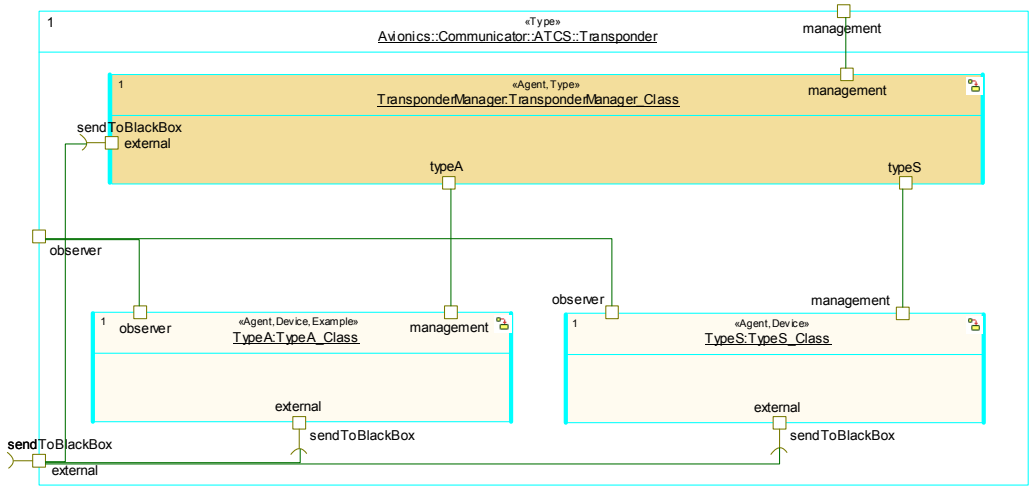

Figura 6-41: UML Structure - Agente Transponder.

6.3 Considerações finais deste capítulo

Neste capítulo foi realizado o projeto do software aviônico embarcado com foco no projeto e realização de uma arquitetura de software genérica e extensível que possa ser utilizada não somente no projeto BR-UAV, mas também em outros projetos de robôs móveis futuros. A adoção de um padrão de projeto para alcançar generalidade e extensibilidade revela-se uma inovação desta pesquisa, dado que não foram encontradas propostas específicas para atingir estas duas características.

No projeto da arquitetura de software aviônico foram apresentadas diversas características desejáveis em arquiteturas de software, das quais se destacam a deliberação-reatividade e confiabilidade-segurança. Das arquiteturas deliberativas e reativas apresentadas, foi adotado na GESAM o paradigma de agentes com características de autonomia, comportamento reativo, comportamento deliberativo, concorrência, encapsulamento e interface. Das arquiteturas confiáveis e seguras apresentadas, foi adotado na GESAM um híbrido dos padrões HRP, DRP e SEP. Vale ressaltar neste ponto, que os padrões de projeto para alcançar deliberação-reatividade e confiabilidade-segurança são propostas bem conhecidas somente adotadas pela GESAM.

Neste ponto, ainda pode haver dúvidas que devem ser esclarecidas com relação à aplicação do paradigma de agentes na arquitetura GESAM. Em ciências da computação, há consenso de que um agente, chamado de agente de software, é um módulo de software que ajuda o usuário em diversas tarefas como, por exemplo, *data mining*. Muitos agentes são baseados em regras pré-programadas, mas são crescentes aqueles com habilidades de aprendizado e adaptação. Já na inteligência artificial, há consenso de que um agente é um ator que observa e atua em um ambiente, podendo o agente ser um robô, conhecido como agente físico, ou um software embarcado de tempo real, também conhecido como agente de software.

Na inteligência artificial, os agentes geralmente são tratados como entidades de alto nível de abstração. Os agentes físicos podem inseridos no topo da classificação de abstração e, em seqüência podem ser inseridas as arquiteturas de controle. A proposta da GESAM é que o paradigma de agentes da inteligência artificial seja estendido até o nível de implementação, melhor representado pela arquitetura de software, uma vez que o paradigma não exige uma inteligência mínima do agente.

Assim como algoritmos de campos potenciais para planejamento de trajetória podem ser considerados comportamentos deliberativos e algoritmos de *collision avoidance* podem ser considerados comportamentos reativos em arquiteturas de controle; a reconfiguração em tempo de execução para obtenção de um sistema degradado, porém seguro, pode ser interpretado como um comportamento deliberativo e a detecção e tratamento de falhas locais podem ser interpretado como comportamento reativo em arquiteturas de software.

De posse das arquiteturas de deliberação-reatividade e de confiabilidade-segurança, projetou-se a GESAM herdando as arquiteturas anteriores e incluindo conceitos de generalidade e extensibilidade. A GESAM foi estruturada em quatro camadas de abstração, sendo o nível de maior abstração batizado de *System*, e em seqüência os níveis *Category*, *Type* e *Device*, este último o de menor abstração. A camada *System* é preferencialmente imutável, pois cria a raiz para p crescimento do sistema. As camadas *Category, Type, Device* são flexíveis, entretanto devem respeitar um padrão de projeto (estrutura, interface e comportamento) desenvolvido que além de abstrair a camada imediatamente inferior, garante segurança e confiabilidade segundo os padrões HRP, DRP e SEP.

Já, na etapa de projeto mecanicista foi realizada a expansão da GESAM para o sistema aviônico do VANT Apoena, consistindo de sensores, fusão e comunicação.

7 IMPLEMENTAÇÃO DO SOFTWARE AVIÔNICO EMBARCADO

Neste trabalho, a etapa de projeto de detalhes que originalmente pertence à fase de projeto foi realizada na fase de implementação. A etapa de tradução não existe, pois a geração de código é automática. Já etapa de testes unitários não será reportada neste trabalho por questões de espaço.

O projeto de detalhes consiste basicamente em “dar vida” aos agentes definidos no projeto da arquitetura. Os agentes nada mais são que instâncias, ou seja, objetos ativos, de classes que são definidas no projeto de detalhes. Assim nesta fase são projetadas as classes do sistema com foco nos atributos, métodos e comportamento uma vez que as classes e interfaces foram definidas na arquitetura. Para tanto, inicialmente define-se os padrões de estruturação de dados (seção 7.1) e abstrações (seção 7.2) que fundamentarão a base da implementação das classes do sistema (seção 7.3).

7.1 Padrões de estruturas de dados

Para que todos os agentes do sistema consigam comunicar-se eficientemente, estabeleceu-se um padrão de estruturação de dados. Este padrão consiste de: **uma unidade básica de dados** que garante que todos os agentes do sistema consigam identificar o valor da informação, sua confiança e se a mesma está atualizada, conforme pode ser observado na Figura 7-1 e um **padrão de estruturação** que define que todas as informações são do tipo ATA_double ou ATA_time, conforme pode ser observado nas Figura 7-2, exceto as informações locais que são armazenadas em atributos conforme a Figura 7-3.

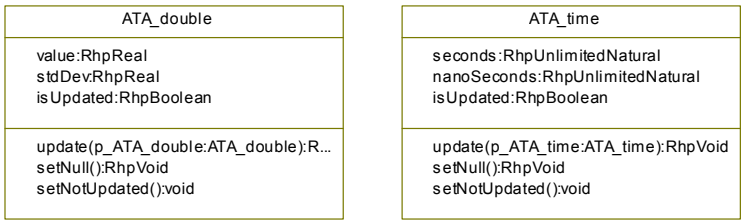

Figura 7-1: UML Object Model - Unidade básica de dados.

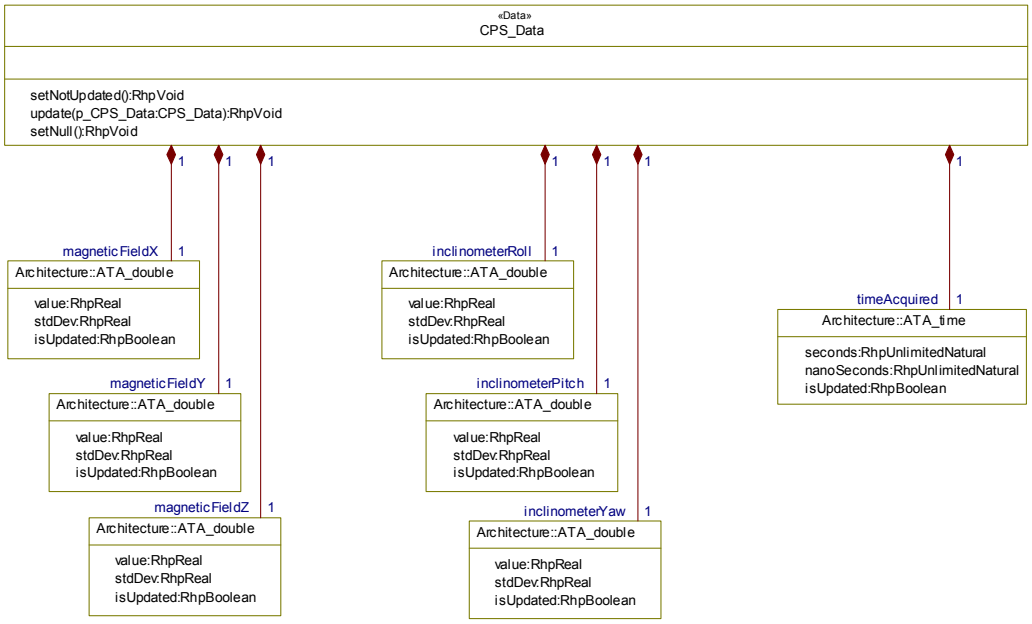

Figura 7-2: UML Object Model - Objetos de dados do sensor type CPS.

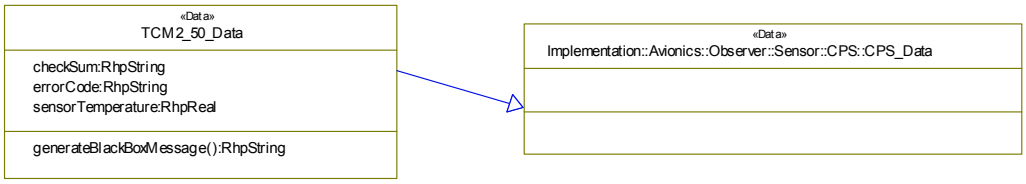

Figura 7-3: UML Object Model - Objetos de dados do sensor device TCM2-50.

Outras estruturas de dados utilizadas no sistema são apresentadas no capítulo 12 (ANEXO - Estrutura de Dados)

7.2 Padrões de Abstrações

Como o objetivo da arquitetura GESAM é generalidade e extensibilidade, estas características devem ser válidas principalmente na implementação, que concentra considerável porção do custo de desenvolvimento. Para tanto foram desenvolvidos padrões de abstrações de linguagem de programação e de abstrações de RTOS conforme a Figura 7-4.

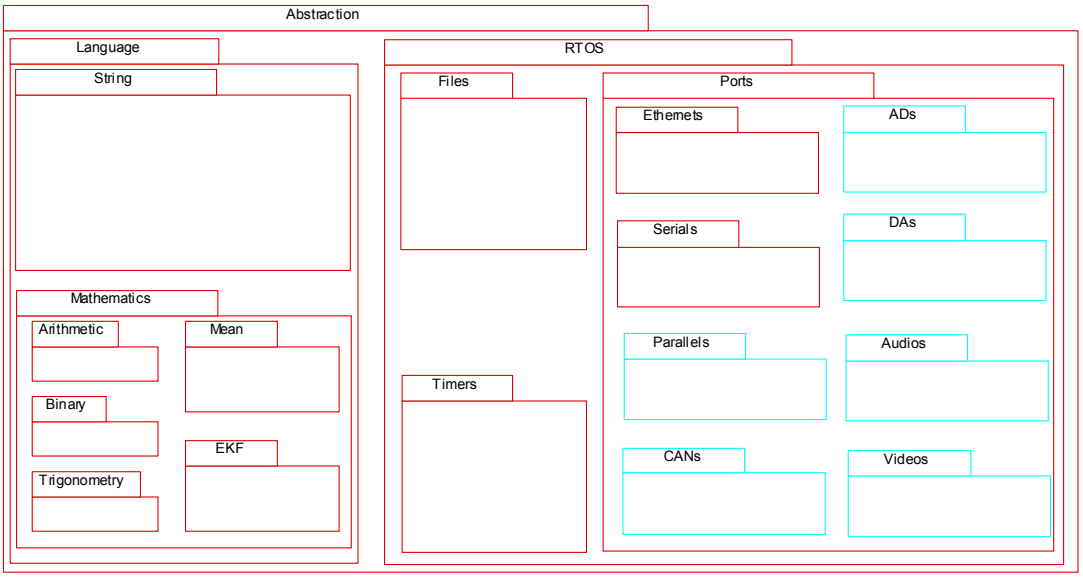

Figura 7-4: UML Object Model - Organização do package Abstraction.

7.2.1 Padrão de Abstrações de Linguagem

As abstrações de linguagem objetivam isolar a linguagem de programação do restante do sistema. Assim na mudança da linguagem de programação, como o Rhapsody implementa várias linguagens (C, C++, Java e ADA), somente este pacote deve ser modificado. Além disso, embora o sistema atenda padrões ANSI C99 e Posix 1003.1c, as abstrações de linguagem suprem deficiências de algumas ferramentas destes padrões, principalmente operações com *strings* e cálculos. Além disso, objetiva restringir acesso a algumas ferramentas de programação que produzem riscos potenciais ao sistema nas mãos de desenvolvedores menos experientes, como operações com ponteiros que podem causar erros de memória.

7.2.2 Padões de Abstrações de RTOS

As abstrações de RTOS objetivam isolar o sistema operacional para as tarefas que necessitam acessar o RTOS, de forma que a mudança do sistema operacional somente impacte na mudança do pacote RTOS que reside dentro do pacote **Abstraction** na Figura 7-4. Para tanto, desenvolveu-se as abstrações de acesso a arquivo e de acesso a timer, Figura 7-5 e Figura 7-6, respectivamente.

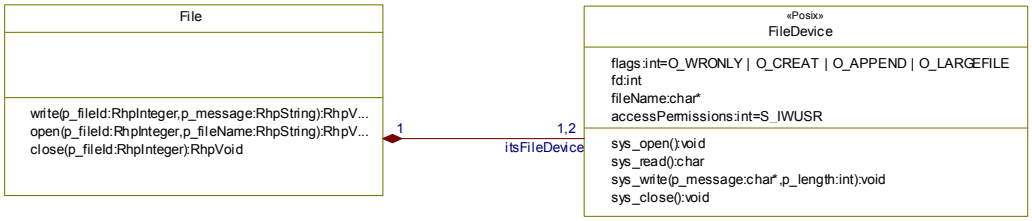

Figura 7-5: UML Object Model - Implementação da abstração de acesso a arquivos.

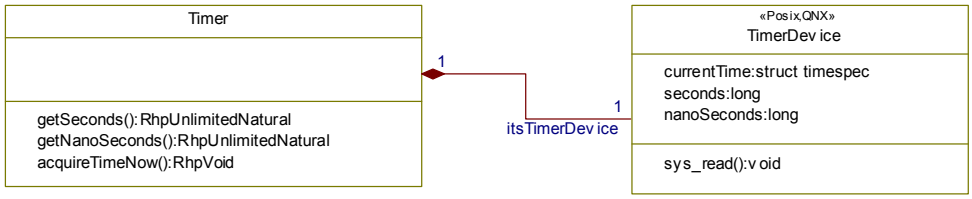

Figura 7-6: UML Object Model - Implementação da abstração de acesso a timer.

Outra abstração de RTOS desenvolvida foi a de acesso a portas. Entretanto devido à complexidade desta abstração, utilizou-se um padrão de projeto. Um padrão de projeto (*design pattern* - DOUGLASS, 1999) é uma solução generalizada para um problema comum (Figura 7-7). Para ser um padrão, o problema deve recorrer suficientemente freqüentemente para ser generalizável. A solução deve ser geral o suficiente para ser aplicada em uma larga variedade de domínios. Se este padrão se aplica somente a um domínio de aplicação, então este padrão é um padrão de análise e não um padrão de projeto. Um padrão de análise é similar a um padrão de projeto, porém aplica-se a um domínio de aplicação específico, como finanças ou aeroespacial. Padrões de análise definem caminhos para organizar problemas específicos dentro de um único domínio de aplicação.

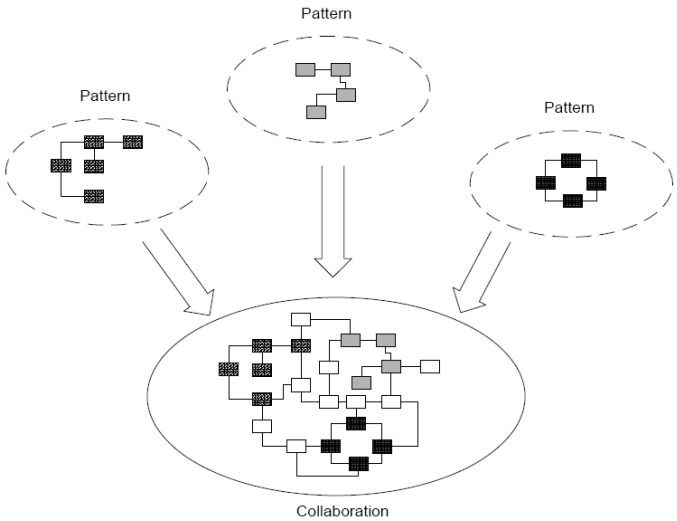

Figura 7-7: Design pattern (DOUGLASS, 1999).

O principal padrão de projeto da arquitetura GESAM é o acesso a portas como serial ou *ethernet*. Para tanto são necessárias três classes. A InterruptDrivenRead (Figura 7-8) uma *thread* cujo único comportamento (Figura 7-9) consiste em esperar interrupções da porta e enviar as leituras para a classe Port. Já a classe Port (Figura 7-8) realiza todas as funções de leitura, escrita, abertura e configuração da porta. Um nível acima destas duas classes encontra-se a classe Protocol (Figura 7-10) uma *thread* cujo comportamento (Figura 7-11) consiste em gerenciar a leitura e escrita de uma porta em um dado protocolo.

Na Figura 7-12 à Figura 7-14 tem-se um exemplo da aplicação deste padrão no protocolo serial RS232 e na Figura 7-15 à Figura 7-18 tem-se um exemplo da aplicação deste padrão no protocolo UDP-IP.

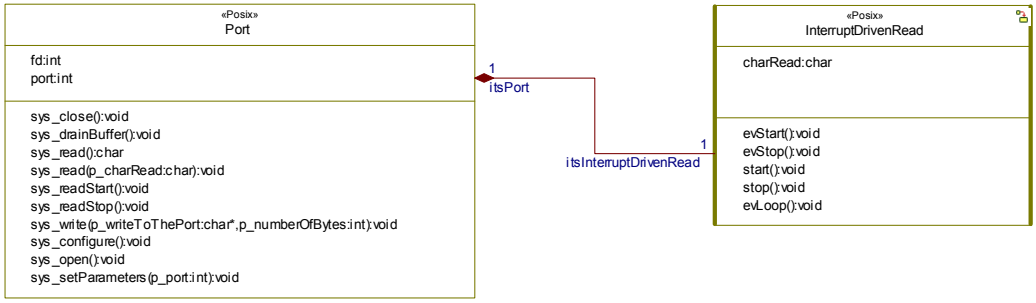

Figura 7-8: UML Object Model - Implementação da abstração de acesso a portas.

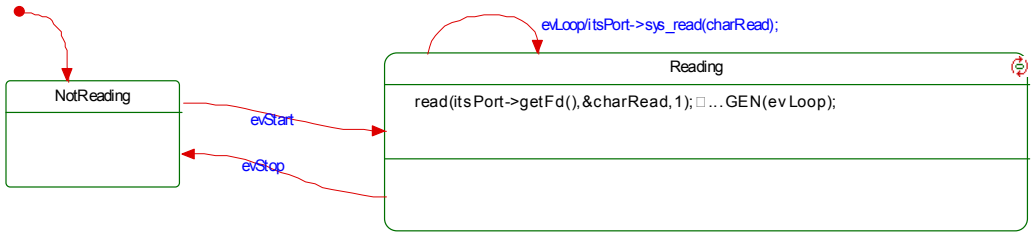

Figura 7-9: UML Statechart - Comportamento da abstração de leitura de portas.

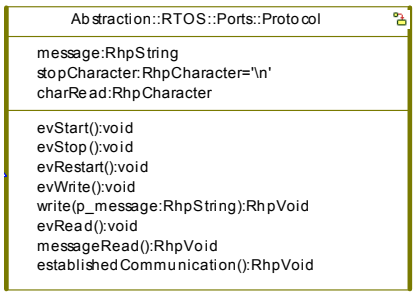

Figura 7-10: UML Object Model - Implementação da abstração de protocolos.

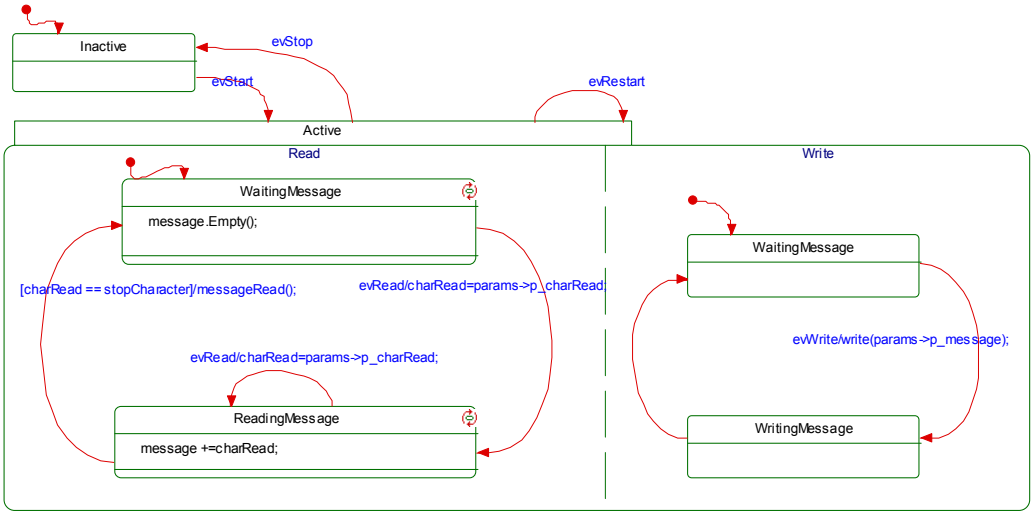

Figura 7-11: UML Statechart - Comportamento de protocolos.

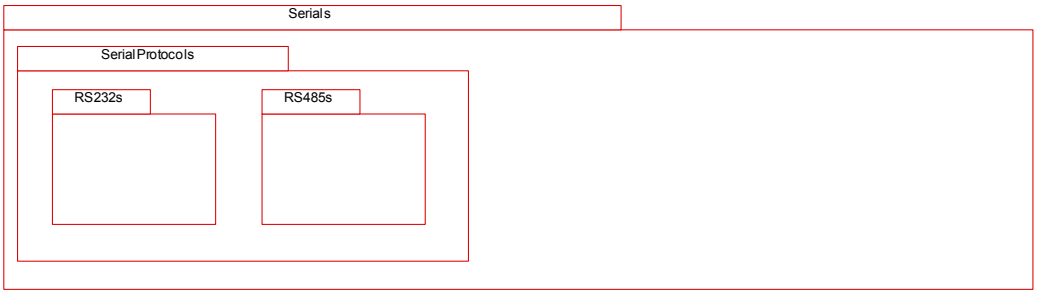

Figura 7-12: UML Object Model - Organização do package Serials.

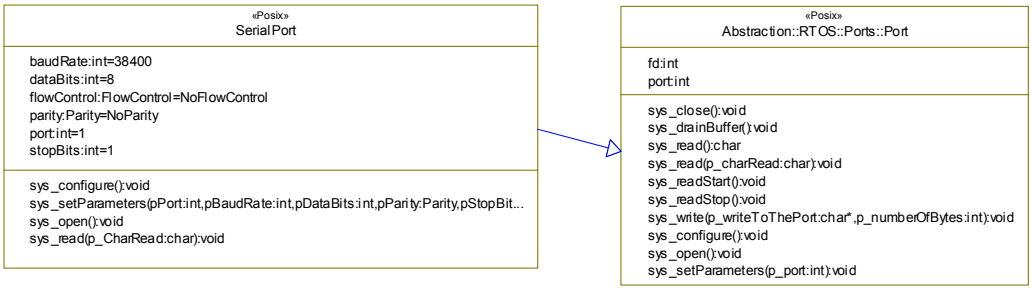

Figura 7-13: UML Object Model - Implementação da abstração de acesso a portas seriais.

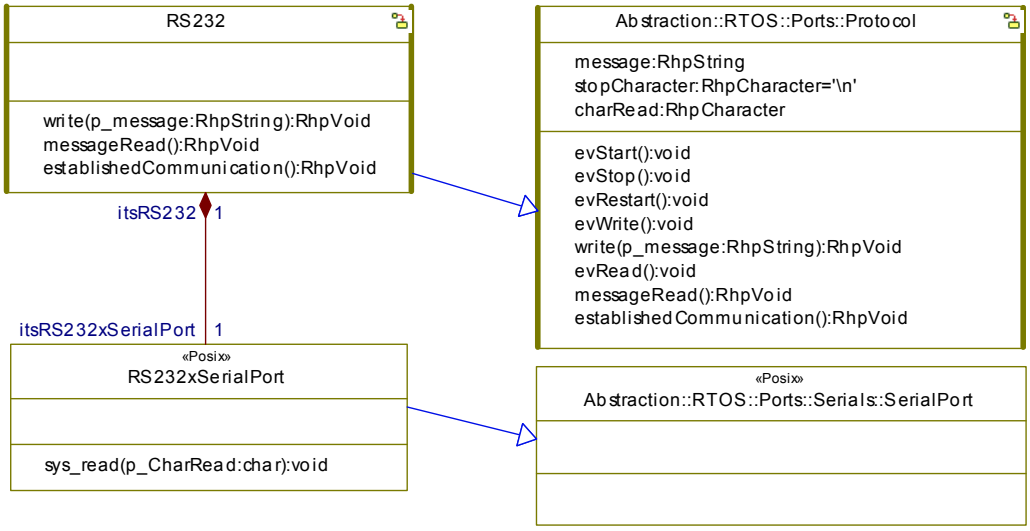

Figura 7-14: UML Object Model - Implementação da abstração de acesso a portas seriais com protocolo RS232.

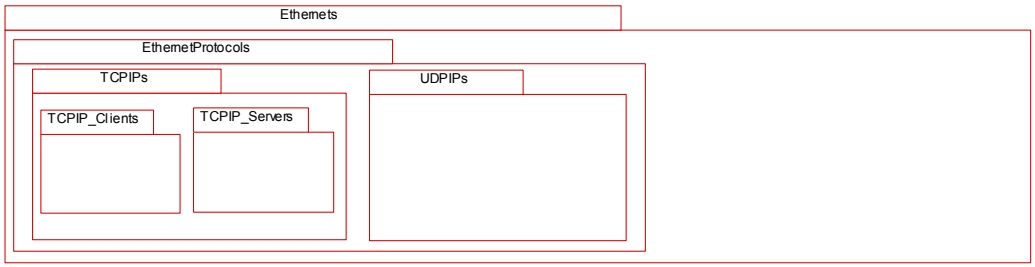

Figura 7-15: UML Object Model - Organização do package Ethernets.

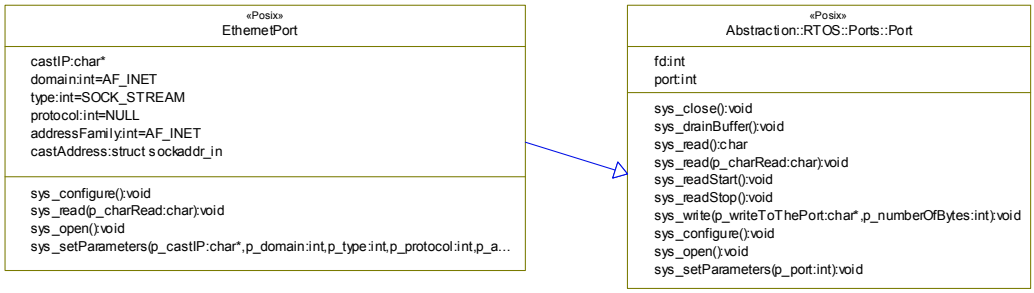

Figura 7-16: UML Object Model - Implementação da abstração de acesso a portas ethernet.

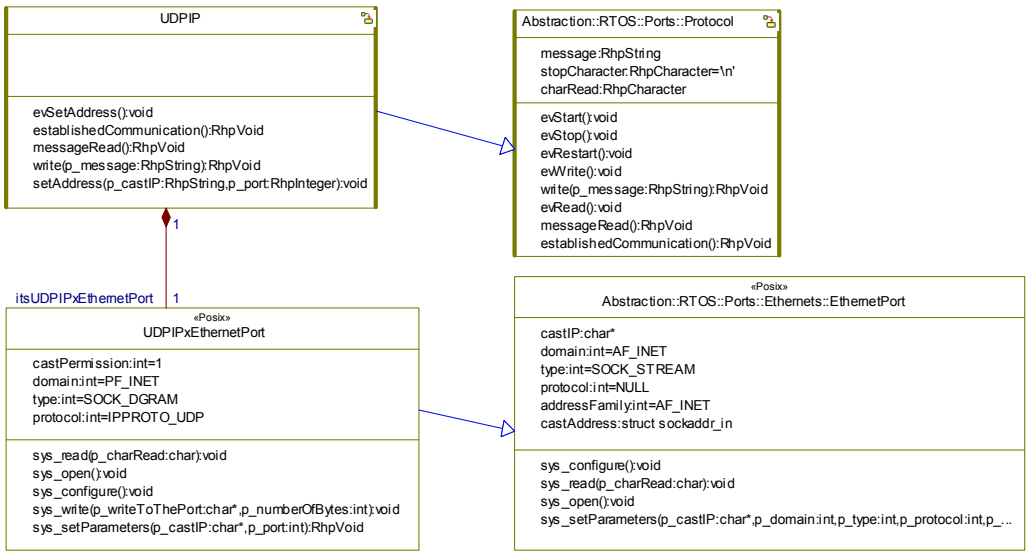

Figura 7-17: UML Object Model - Implementação da abstração de acesso a portas ethernet com protocolo UDPIP.

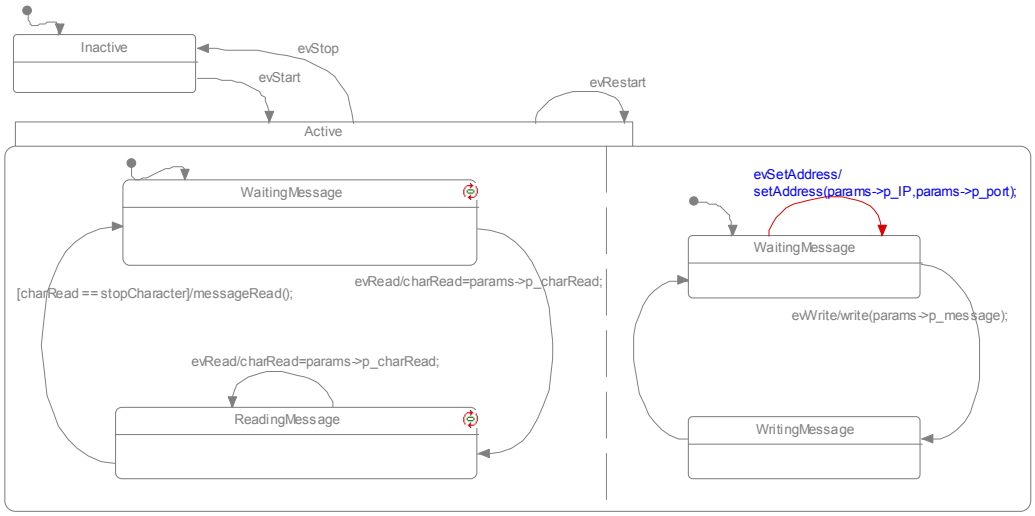

Figura 7-18: UML Statechart - Comportamento da abstração de acesso a portas ethernet com protocolo UDPIP.

7.3 Pacote de Implementação

No pacote implementação (*Implementation Package*) são guardadas todas as classes relacionadas a todos agentes do sistema. Neste trabalho, por questões de espaço, somente serão apresentados no capítulo 13 (ANEXO - Resumo dos Projetos Estado da Arte), os agentes em preto da Figura 7-19.

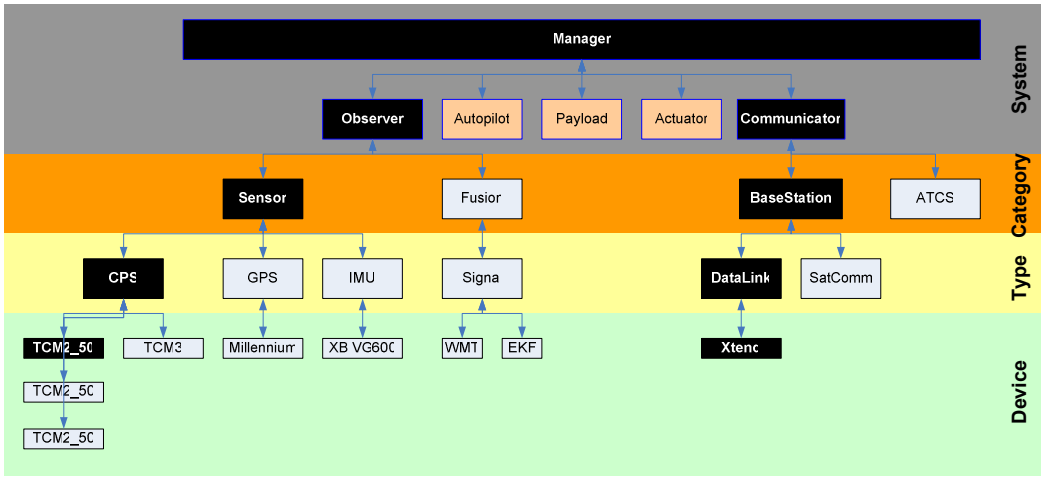

Figura 7-19: Agentes apresentados no pacote implementação anexo.

7.4 Considerações finais deste capítulo

Este capítulo apresentou o projeto de detalhes da arquitetura GESAM, mais especificamente os padrões de estruturação de dados, abstrações e implementação de classes.

Para que todos os agentes do sistema consigam se comunicar foi desenvolvido um padrão de estruturação de dados composto por uma unidade básica de dados e um padrão de estruturação.

Para potencializar a capacidade de generalidade e extensibilidade da GESAM, foram desenvolvidos padrões de abstração de linguagem e padrões de abstração de RTOS.

Finalmente, a implementação foi apresentada em anexo, dando detalhes internos de classes estratégicas e representativas do sistema.

8 TESTES DO SOFTWARE AVIÔNICO EMBARCADO

Os testes de software podem ser divididos em testes unitários e de integração. Os testes unitários foram realizados paralelamente às fases de projeto e implementação em cada um dos agentes da arquitetura e, por isso serão omitidos. Já os testes de integração foram realizados segundo o plano de testes proposto na seção 8.1. Serão apresentados os resultados realizados na configuração de operação remota com observações WMT (seção 8.2) e EKF (8.3).

8.1 Plano de testes de integração

Todo o plano de testes será baseado nos requisitos estabelecidos na seção 5.3 com o objetivo de garantir que o sistema atende aos requisitos do software embarcado.

8.1.1 Hardware para teste

Os hardware para teste está definido na Figura 5-26 e é composto por: VG600AA (IMU), MiLLennium (GPS), TCM2-50 (CPS), ECU de teste e *switch*.

8.1.2 Implantação dos testes

Os testes serão realizados em dois ambientes, um estático e um dinâmico. O ambiente de testes estático é composto pela bancada de desenvolvimento e de testes estáticos e pelo hardware para teste como pode ser observada na Figura 8-1.

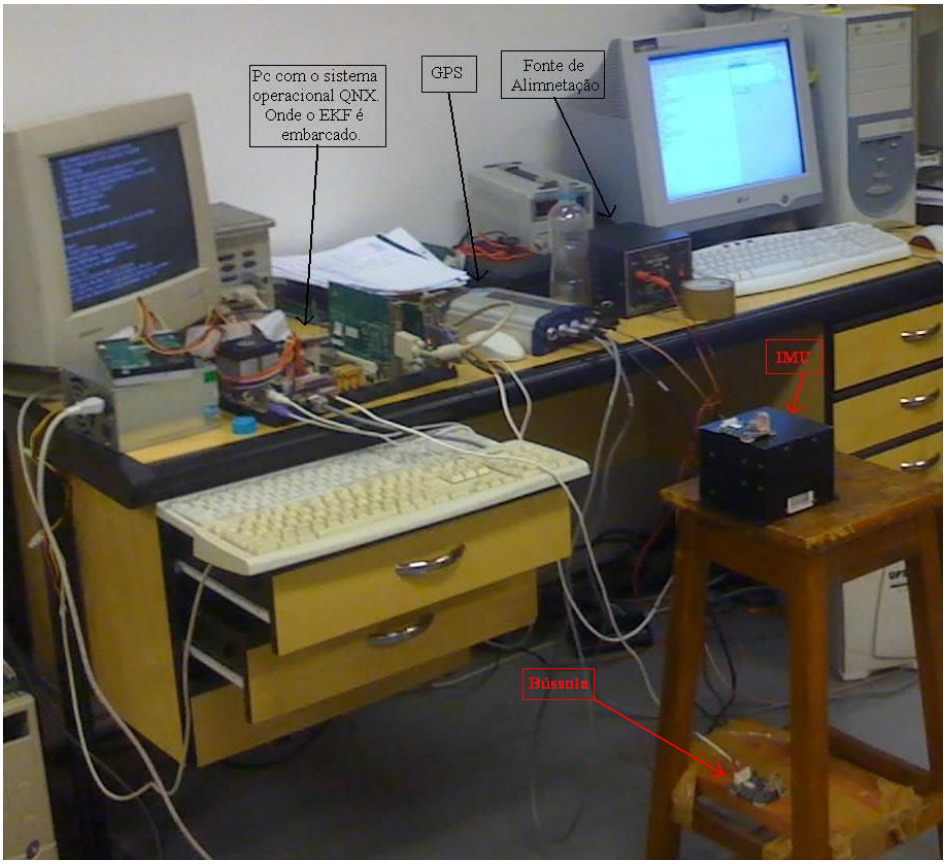

Figura 8-1: Bancada de desenvolvimento e de testes estáticos.

O ambiente de testes dinâmicos é composto pela bancada de testes dinâmicos em solo (um veículo preparado para o acoplamento do hardware de testes) e pelo hardware para teste como pode ser observada na Figura 8-2 e na Figura 8-3.

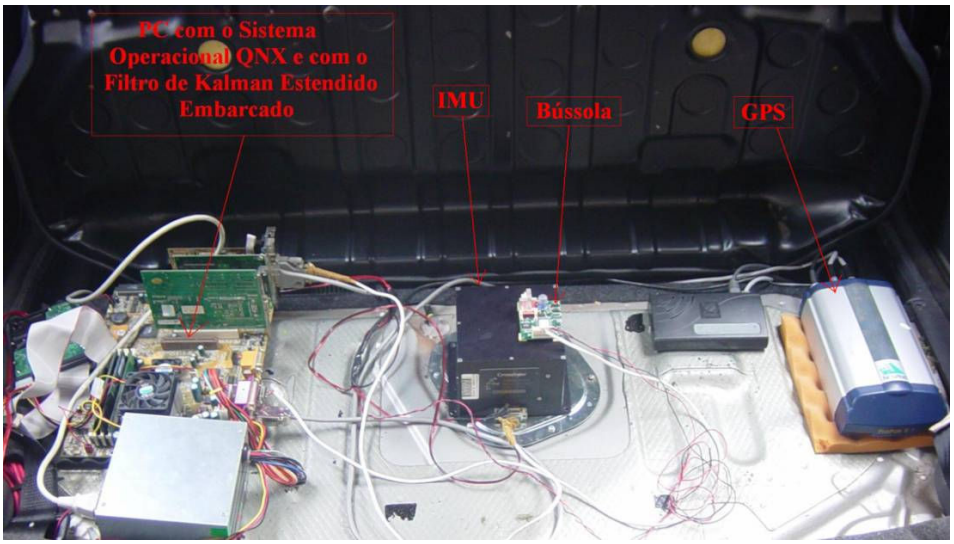

Figura 8-2: Aviônica embarcada em veículo de testes.

Figura 8-3: Antena do GPS no veículo de teste.

8.1.3 Testes funcionais

Os testes funcionais consistem em avaliar o atendimento aos casos de uso estabelecidos nas Figura 5-28, Figura 5-29, Figura 5-30 e Figura 5-31. Os resultados das funcionalidades testadas podem ser observados na Tabela 8-1.

Tabela 8-1: Resultado dos testes funcionais.

Caso de uso	Resultado	Observações
Inicializar sistema	Sucesso	
Escrever dados e *log* na caixa preta	Sucesso parcial	Após 12 horas de funcionamento, o sistema operacional matou o processo do software aviônico, pois o BlackBox armazenou 1,2 GB de informação de *log* e de dados da VG600AA (IMU) a 85 Hz, MiLLennium (GPS) a 10 Hz, TCM2-50 (CPS) a 10 Hz e EKF a 85 Hz de propagação e 20 Hz de atualização. Isto porque atingiu a capacidade de disco. Assim, deve-se implementar uma rotina de segurança contra sobre armazenamento.
Inicializar sensores VG600AA (IMU), MiLLennium (GPS) e TCM2-50 (CPS)	Sucesso	
Configurar sensores VG600AA (IMU), MiLLennium (GPS) e TCM2-50 (CPS)	Sucesso parcial	Falha na sincronização da mensagem binária da VG600AA (IMU) com a *thread* de formação da mensagem da serial do agente **VG600AA**. Assim, deve-se implementar melhor a rotina de sincronização.
Receber correção DGPS	Falha no hardware	O GPS da base envia dados de correção, entretanto o MiLLenium (GPS embarcado) não reconhece esses dados e mantém em *sigle point*.
Aquisitar Sensores VG600AA (IMU), MiLLennium (GPS) e TCM2-50 (CPS)	Sucesso	
Fundir dados sensoriais pelo método WMT e EKF	Sucesso parcial	Desempenho computacional do EKF: satisfatório. Desempenho de estimação do EKF: regular. Estabilidade do EKF: -divergência em uma hora de funcionamento o EKF na forma original. -divergência em uma hora e meia de funcionamento do EKF na forma de Joseph.
Enviar dados para o XTend (*data-link*)	Sucesso	

8.1.4 Testes de desempenho

Os testes de desempenho consistem em validar os requisitos temporais resumidos na Figura 5-33 e são a seguir definidos:

- Testes de corretude: diagrama de seqüência gerado pelo debug de modelo que corresponde ao software da aplicação instrumentado enviando mensagens dos estados ao ambiente de desenvolvimento Rhapsody. Neste teste, é avaliado se a seqüência de envio de mensagens está correta. Entretanto, não é possível verificar se os requisitos temporais estão sendo atendidos dado o alto custo computacional da instrumentação da aplicação.

- Testes processamento: por meio de diagramas de distribuição de consumo de processamento avalia-se o consumo do *kernel*, e da aplicação. No *kernel* é possível avaliar a influência no processamento da instrumentação do *kernel*. Na aplicação é possível analisar o consumo de cada uma das *threads*, dos *drivers* e interrupções utilizados pela aplicação. Assim pode-se levantar as *threads* com processamento dominante.

- Testes de eventos: por meio de diagrama de linha do tempo avalia-se cada evento do sistema embarcado (*kernel* + aplicação) como: representação de estados, sinais, semáforos, exclusão mútua, eventos de *kernel*. A maior aplicação deste testes é a verificação do atendimento aos deadlines impostos na Figura 5-33.

8.2 Resultados na configuração de operação remota com observações WMT

O primeiro resultado foi obtido a partir de debug de modelo, no qual o software instrumentado envia mensagens dos estados ao ambiente de desenvolvimento Rhapsody. Como dito, esta procedimento de teste é eficiente para analisar a corretude do sistema e não seu desempenho. Na Figura 8-4, pode-se observar a seqüência de eventos dos principais agentes do sistema: **VG600AA** (IMU), **TCM2-50** (CPS), **MiLLennium** (GPS) e **Xtend** (*data-link*). Os demais agentes estão omitidos no ENV para facilitar o entendimento da imagem. A VG600AA (IMU) está operando a 85Hz, o TCM2-50 (CPS), MiLLenium (GPS) e XTend (*data-link*) a 10Hz. Verifica-se a corretude do funcionamento do sistema uma vez que tem-se todas as mensagens esperadas relativas à operação destes dispositivos em um ciclo de 100ms, ou seja, aproximadamente 8 mensagens da VG600AA, 1 da TCM2-50, 1 do MiLLenium e 1 para o XTend.

Figura 8-4: UML Sequence - Seqüência de eventos principais.

Os próximos resultados foram obtidos através de *debug* de código, no qual o *kernel* instrumentado envia informações do sistema e da aplicação a IDE QNX Momentics. Na Figura 8-5 pode-se observar o resumo da distribuição de processamento em dois segundos de teste. Em um computador embarcado, cuja configuração é 1,2GHz de processamento e memória de 128MBytes, verificou-se o consumo de processamento total de 26,1%, incluindo aplicação de 17,5%, sistema de 5,6% e interrupções de 3,0%. Vale lembrar que no consumo do sistema e interrupções integra-se o consumo de instrumentação do *kernel*.

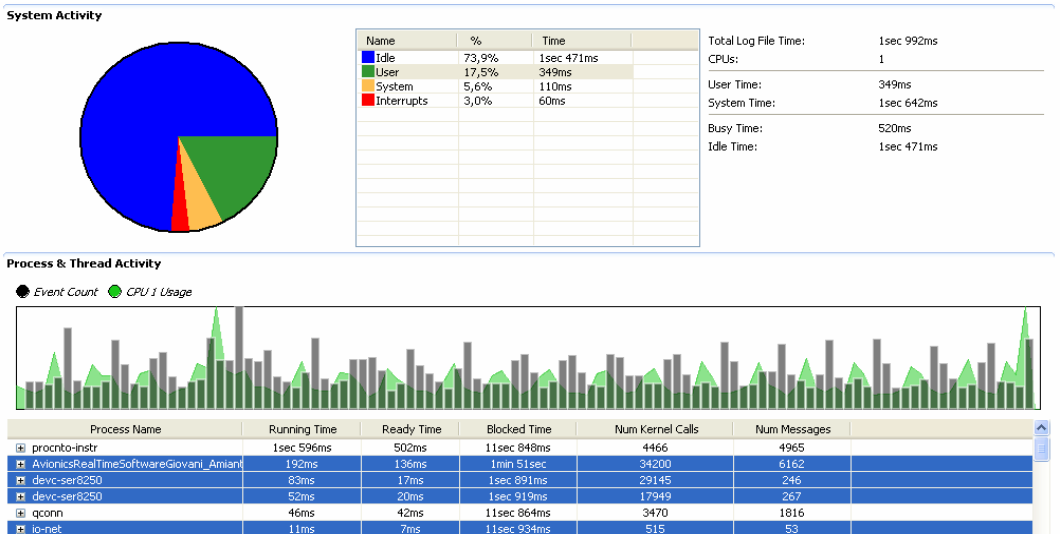

Figura 8-5: Resumo da distribuição de consumo de processamento e de eventos.

A Figura 8-6 trata da distribuição de consumo de processamento total. Observa-se entre 347ms e 508ms um repentino acréscimo de processamento, assim deve-se verificar a fonte deste para garantir que não provêm de potenciais problemas com a aplicação.

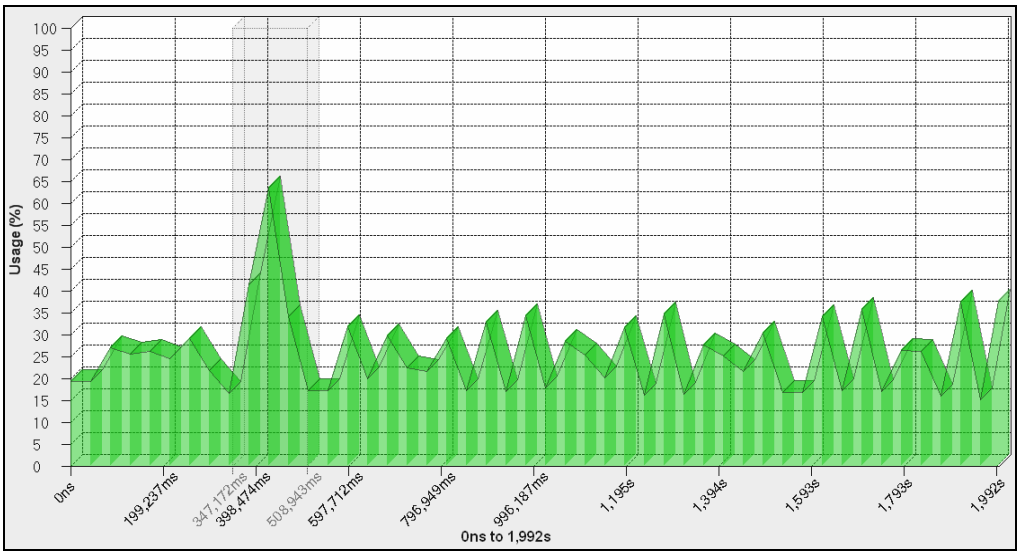

Figura 8-6: Distribuição de consumo de processamento total no tempo.

Na Figura 8-7, tem-se o consumo de processamento do sistema que é composto pelos seguintes processos: *kernel* do QNX Neutrino instrumentado (procnto-instr), servidor de

eventos do *kernel* QNX Neutrino (qconn), *driver* de rede (io-net) e *driver* da IDE (devb-eide). Já na Figura 8-8, tem-se a confirmação que o repentino acréscimo de processamento entre 347ms e 508ms é gerado principalmente pela *thread* 6 do *kernel* instrumentado. Para complementar a análise do sistema, na Figura 8-9, pode-se observar o consumo de processamento das interrupções (*interrupt*) ligadas ao sistema.

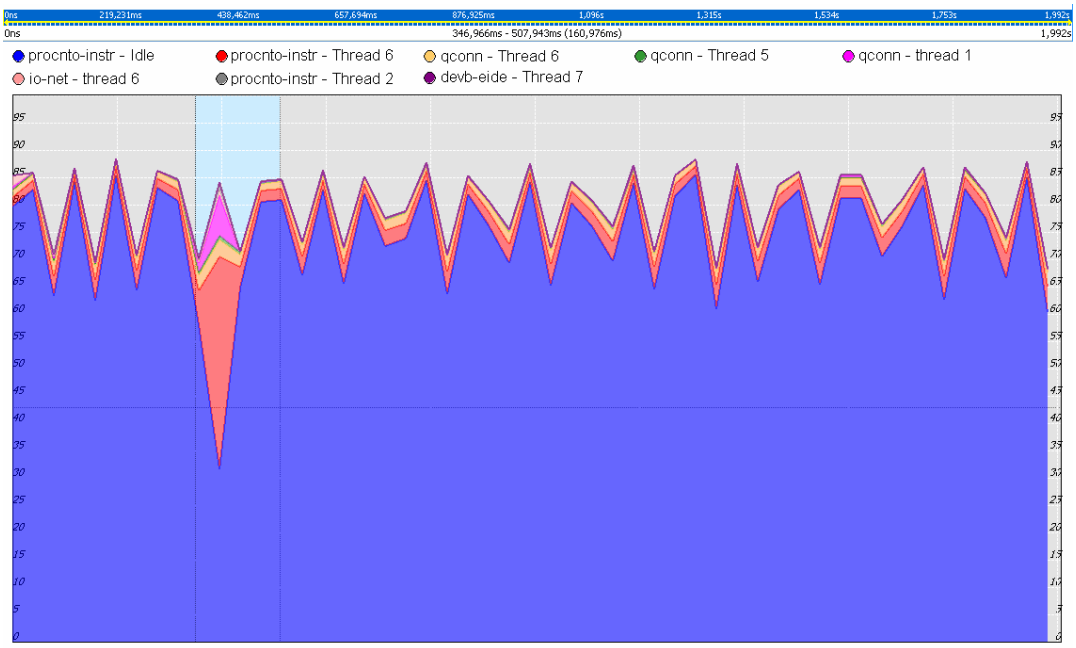

Figura 8-7: Distribuição de consumo de processamento de System - Processamento significativo (>1us/s).

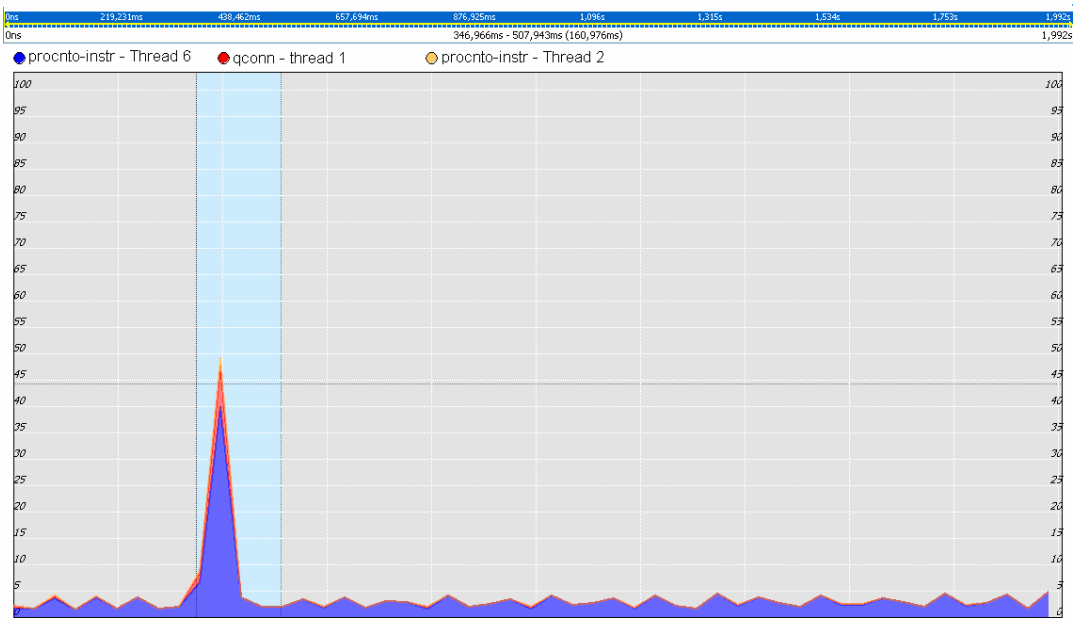

Figura 8-8: Distribuição de consumo de processamento das tarefas do sistema que geram consumo de processamento pontual.

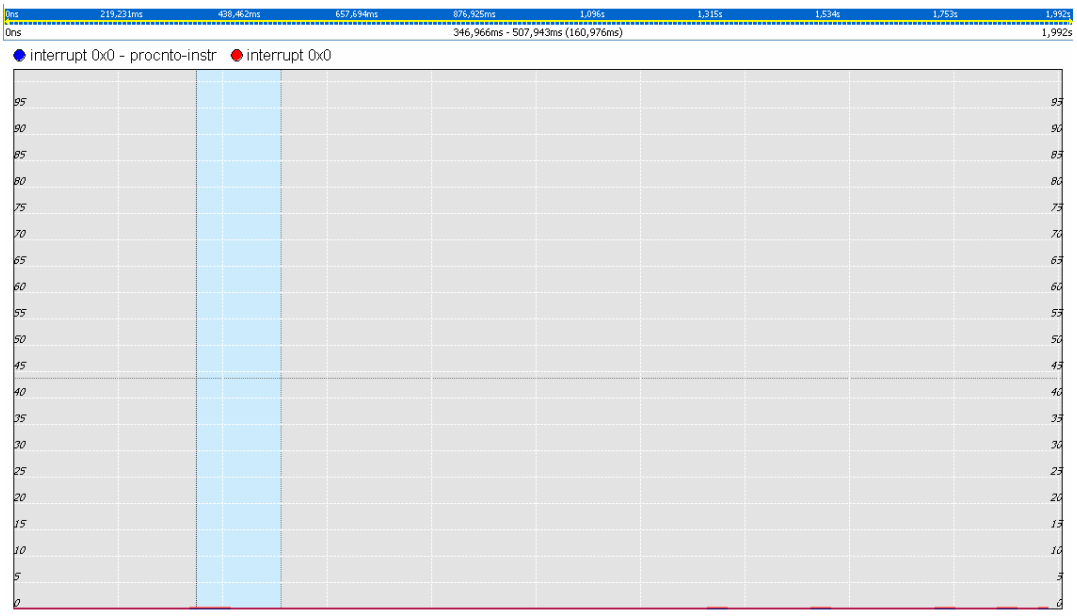

Figura 8-9: Distribuição de consumo de processamento das interrupções utilizadas pelos processos do sistema.

Na Figura 8-10 é apresentada a distribuição do consumo de processamento das *threads* da aplicação (processo **Avionics**) com processamento significativo. Vale destacar o perfil em serra cujo período é aproximadamente 100ms, que é o período de operação do sistema.

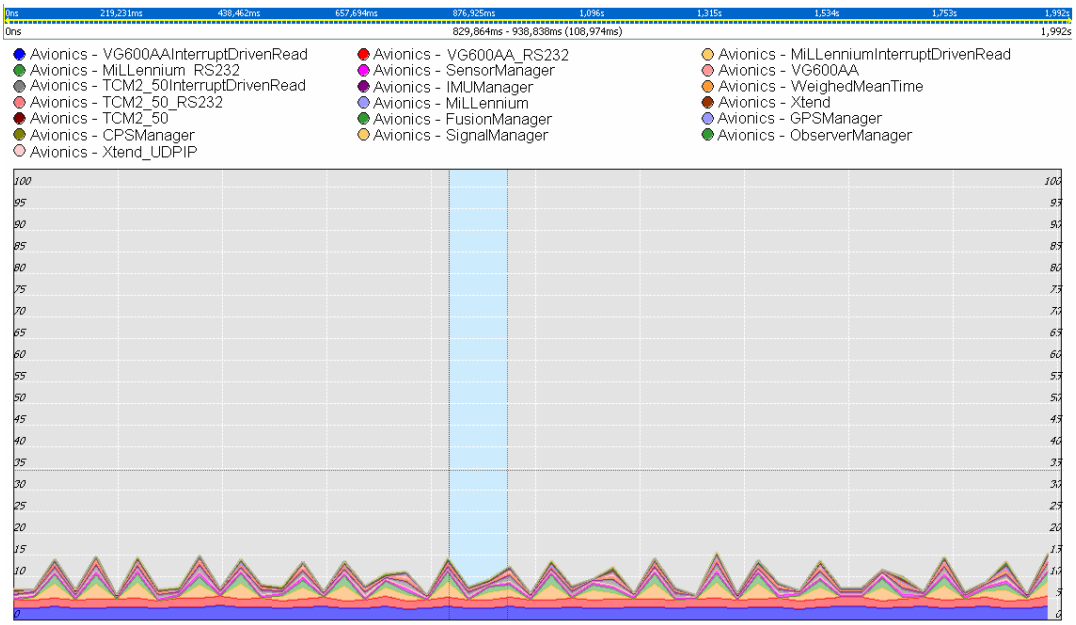

Figura 8-10: Distribuição de consumo de processamento da aplicação AvionicsRealTimeSoftware - Processamento significativo (>1us/s).

Na Figura 8-11 é apresentada a distribuição do consumo de processamento dos *drivers* da serial (devc – ser8250) e da rede (io-net) utilizados pela aplicação e que tem processamento significativo. O devc-ser8250 em azul é o *driver* utilizado pela VG600AA (IMU) e isso fica evidente dada à continuidade do processamento. Já o devc-ser8250 vermelho é o *driver* utilizado pelo MiLLennium (GPS) e isso é evidenciado pelo perfil em serra cuja freqüência é de aproximadamente 100ms.

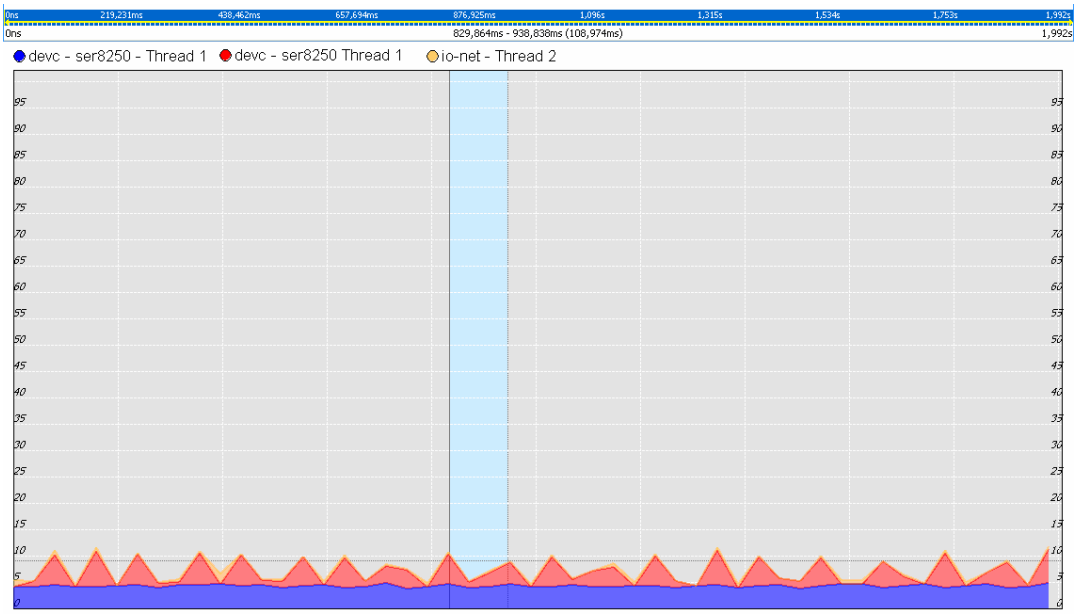

Figura 8-11: Distribuição de consumo de processamento dos drivers utilizados pela aplicação - Processamento significativo (>1us/s)

Na Figura 8-12 é apresentada a distribuição do consumo de processamento das interrupções (*interrupt*) utilizadas tanto pela aplicação como pelos *drivers* utilizados pela aplicação e que tem processamento significativo.

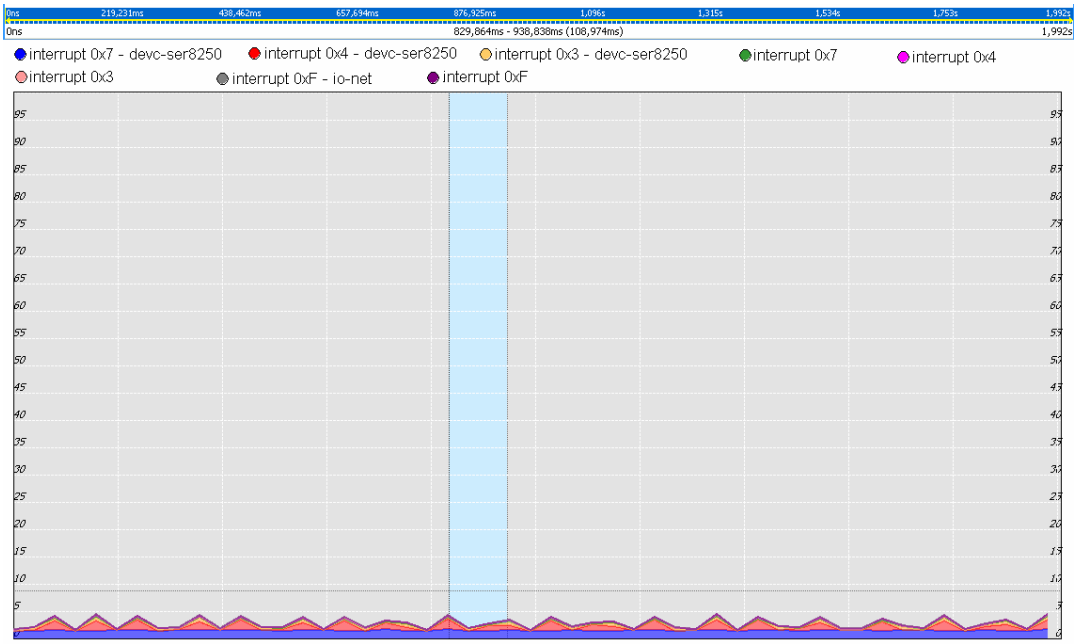

Figura 8-12: Distribuição de consumo de processamento das interrupções utilizadas pela aplicação - Processamento significativo (>1us/s)

Na Figura 8-13 é apresentada a distribuição do consumo de processamento de todas as tarefas que juntas somam 95% do processamento da aplicação, que no caso, inclui a própria aplicação, bem como *drivers* e interrupções utilizados pela mesma.

Nesta figura verifica-se uma informação importante. Os principais consumidores de processamento são em seqüência:

1. *Driver* da serial da VG600AA (IMU);
2. *Thread* de leitura da serial da VG600AA (IMU);
3. *Driver* da serial do MiLLennium (GPS);
4. Interrupção do *driver* da serial da VG600AA (IMU);
5. *Thread* de montagem da mensagem da serial da VG600AA (IMU);
6. *Thread* de leitura da serial do MiLLennium (GPS);
7. *Thread* de montagem da mensagem da serial do MiLLennium (GPS);
8. Interrupção do *driver* da serial do MiLLennium (GPS);
9. Agente **SensorManager**;
10. Agente **VG600AA**.

Desta informação, pode-se inferir duas conclusões. A primeira mostra que se for necessário otimizar o processamento na aplicação, deve-se focar nas *threads* de leitura e montagem de mensagens seriais, pois se assume que seja improvável alcançar uma melhoria de desempenho nos *drivers* uma vez que estes foram desenvolvidos pela QNX. A segunda conclusão indica que, embora se esperasse uma sobrecarga de processamento na comunicação entre agentes, pois a arquitetura exige grande quantidade de comunicação, verificou-se que o processamento relacionado a esta comunicação é desprezível. Isto invalida a justificativa de que a principal desvantagem de arquiteturas hierárquicas é o elevado custo com processamento na passagem de mensagens.

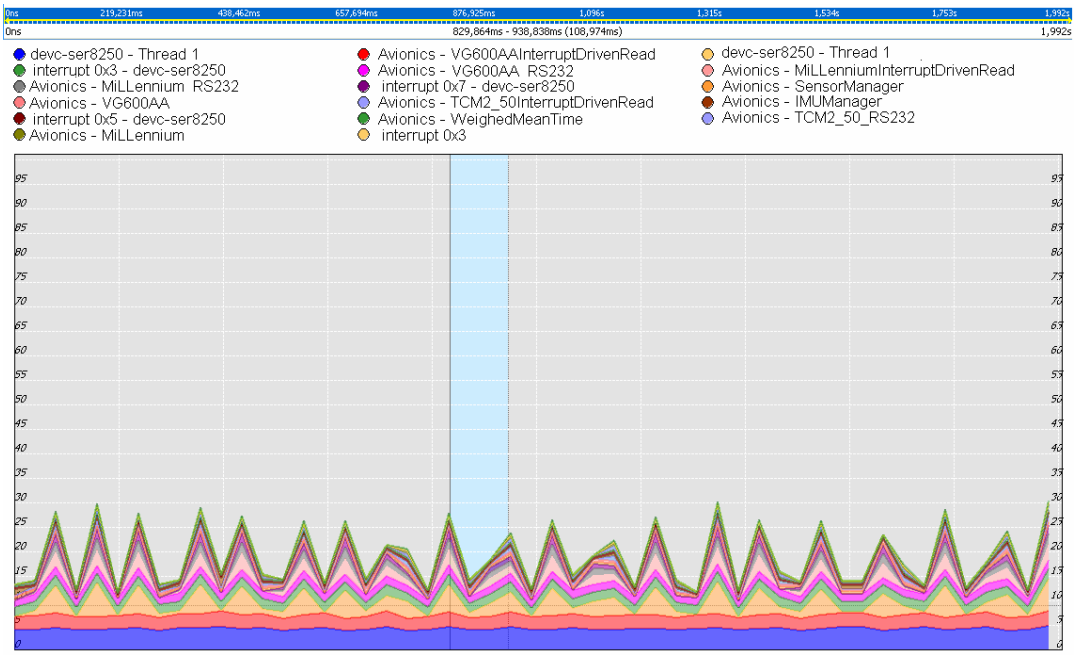

Figura 8-13: Distribuição de consumo de processamento de 95% do processamento de aplicação.

Na Figura 8-14, tem-se o diagrama de linha do tempo apresentado uma visão geral em 2s de instrumentação do *kernel*. Este tipo de diagrama apresenta interrupções, eventos de *kernel*, estados dos processos e *threads* bem como troca de mensagens entre processos e muitas outras informações, como pode ser observado na Figura 8-15 e Figura 8-16.

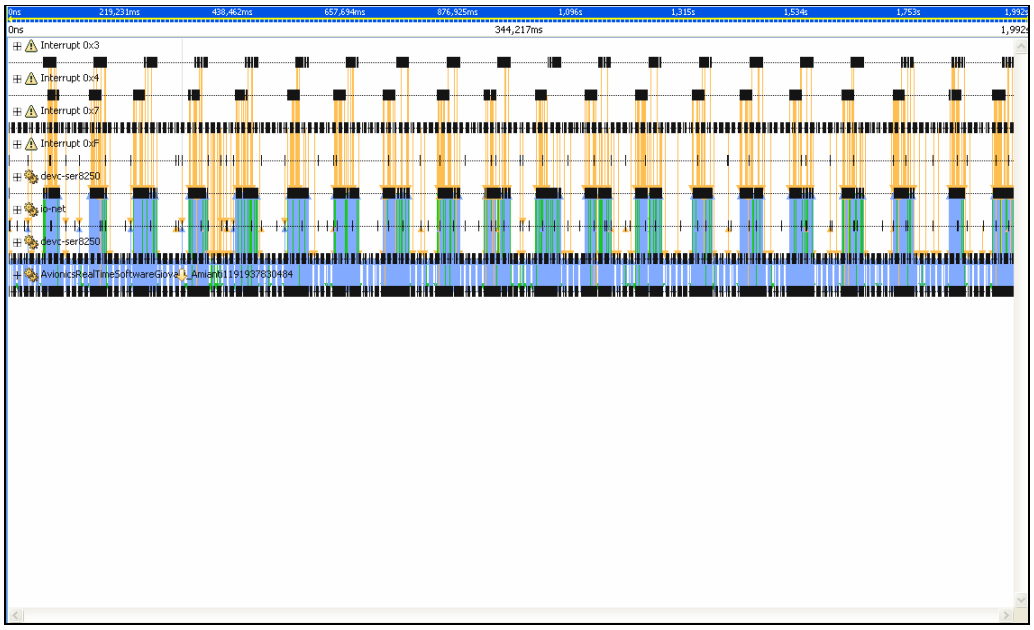

Figura 8-14: Diagrama de linha do tempo da aplicação e derivados em tempo de instrumentação de 2s.

Name	State Line Color
Dead	20,20,20
Running	0,255,0
Ready	50,150,50
Stopped	20,20,20
Send	255,0,0
Receive	0,0,255
Reply	0,0,255
Stack	255,150,150
WaitThread	255,150,150
WaitPage	255,150,150
SigSuspend	254,230,90
SigWaitInfo	254,230,90
NanoSleep	128,0,128
Mutex	255,0,255
Condvar	254,150,70
Join	254,230,90
Interrupt	254,230,90
Semaphore	254,150,70
WaitCtx	255,150,150
NetSend	255,0,0
NetReply	0,0,255

Figura 8-15: Cores dos estados no diagrama de linha do tempo.

Names	Colors
Background	255,255,255
Text	20,20,20
Timeline	20,20,20
Event	20,20,20
Element Selection	200,200,200
Range Selection	225,225,225
Range Markers	175,175,175
Search Markers	255,0,0
IPC Send	130,170,255
IPC Pulse	254,190,80
IPC Signal	254,150,70
IPC Reply	25,200,25
IPC Error	255,150,150
Event Labels	237,237,237
State Labels	170,210,255
Priority Labels	200,240,255
Extra	170,210,255

Figura 8-16: Cores dos eventos e comunicações no diagrama de linha do tempo.

Na Figura 8-17, tem-se um diagrama de linha do tempo com filtro de apresentação para o *driver* de rede e aplicação. Neste é possível observar a peridiciosidade do envio de mensagens da aplicação para a rede. Estas são as mensagens de envio de telemetria para a estação base.

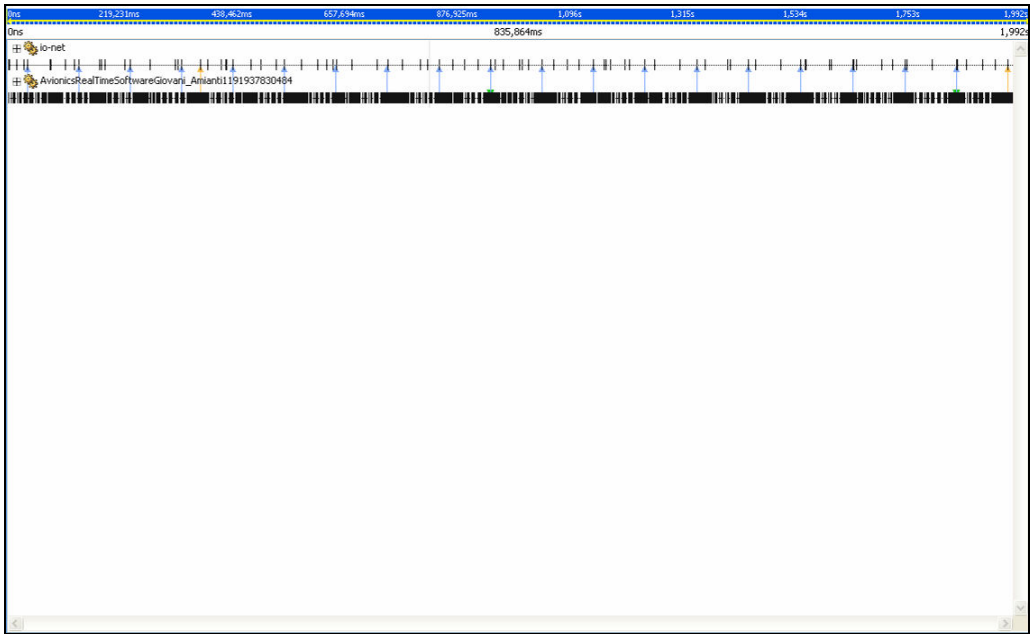

Figura 8-17: Diagrama de linha do tempo da aplicação com foco no envio de telemetria pela rede em tempo de instrumentação de 2s.

Na Figura 8-18, tem-se uma ampliação da linha do tempo que representa a comunicação da aplicação com o *driver* de rede com medição do período entre envios de mensagens. Como a *thread* de comunicação com a rede *ethernet* possui a menor prioridade do

sistema e esta está atendendo ao requisito RT1, pode-se inferir que todo o sistema está atendendo ao RT1.

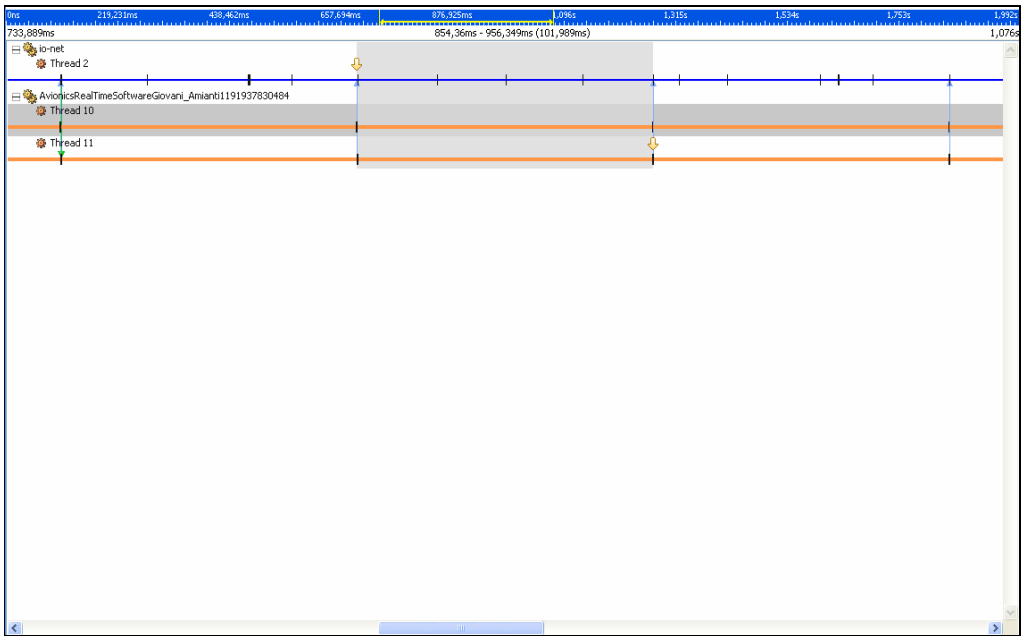

Figura 8-18: Ampliação do diagrama de linha do tempo da aplicação com foco no envio de telemetria pela rede em tempo de instrumentação de 300ms.

Na Figura 8-19, verifica-se detalhes de interrupções e comunicação entre processos que ocorrem em um período de operação, ou seja, 100ms. Na primeira linha, observa-se a interrupção gerada pela serial da TCM2-50 (CPS), que possui um *baud rate* de 9600bps e freqüência de 10 Hz, na segunda linha observa-se a interrupção gerada pela serial do MiLLennium (GPS) que possui um *baud rate* de 57600bps e freqüência de 10 Hz, e na terceira linha observa-se a interrupção gerada pela serial da VG600AA (IMU) que possui *baud rate* de 38400bps e freqüência de 85 Hz. Já na quarta linha observa-se a interrupção da rede que é compartilhada pela instrumentação do *kernel* e a aplicação assim, apresentando mais interrupções que o esperado. Na quinta, sexta e sétima linhas têm-se respectivamente o *driver* de serial da TCM2-50 e MiLLennium, o *driver* da rede, e o *driver* serial da VG600AA. Finalmente na ultima linha tem-se a aplicação.

Embora seja possível ampliar mais ainda o diagrama da Figura 8-19, com possibilidade de representação de estados, sinais, semáforos, exclusão mútua, eventos de *kernel* entre outras informações; esta ampliação revela-se impraticável de apresentar textualmente, sendo praticável para o desenvolvedor. E, de fato, tais informações diversas vezes foram imprescindíveis para verificar a consistência de funcionamento do sistema.

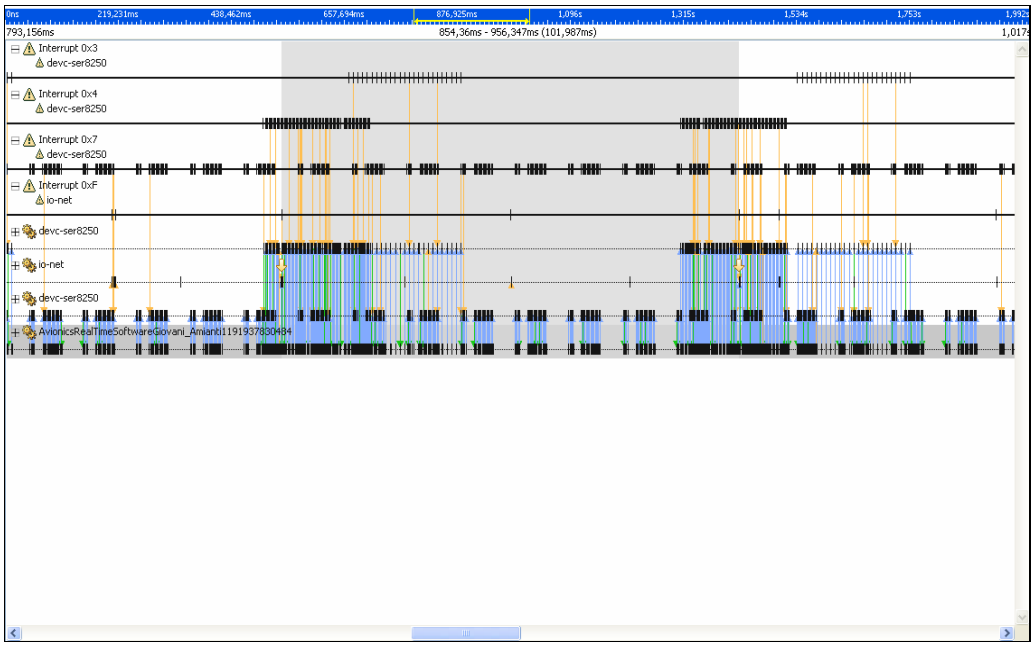

Figura 8-19: Ampliação do diagrama de linha do tempo da aplicação e derivados em tempo de instrumentação de 100ms.

8.3 Resultados na configuração de operação Remota com observações EKF

O foco desta etapa concentra-se na obtenção e análise de informações relacionadas ao desempenho do ponto de vista computacional e não da estimação dos estados. Os próximos resultados também foram obtidos através de debug de código, no qual o *kernel* instrumentado envia informações do sistema e da aplicação à IDE QNX Momentics.

Na Figura 8-20, pode-se observar o resumo da distribuição de processamento em dois segundos de teste. Em um computador embarcado, cuja configuração de processamento era de 1,2GHz e memória de 128MBytes, verificou-se o consumo de processamento total de 53,9%, incluindo aplicação de 38,3%, sistema de 8,7% e interrupções de 6,9%.

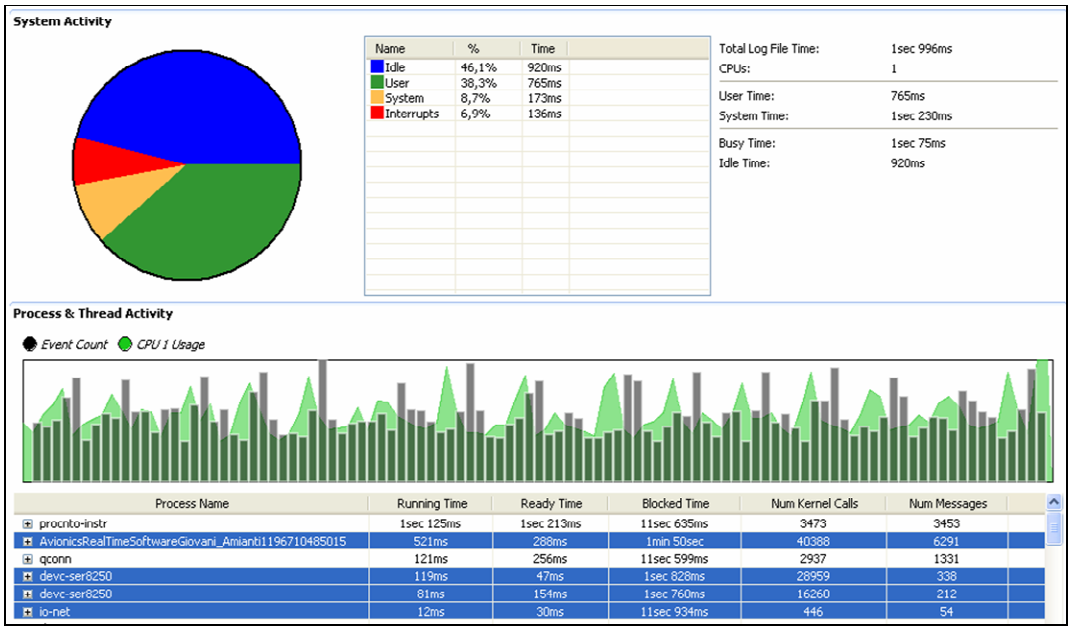

Figura 8-20: Resumo da distribuição de consumo de processamento e de eventos.

Na Figura 8-21 é apresentada a distribuição do consumo de processamento das *threads* da aplicação com processamento significativo. Pode-se observar a *thread* 28 que executa os cálculos do filtro de Kalman estendido e, conforme o esperado consome uma quantidade de processamento significativo. Nesta configuração de operação, esta foi a *thread* que consumiu a maior parcela de processamento.

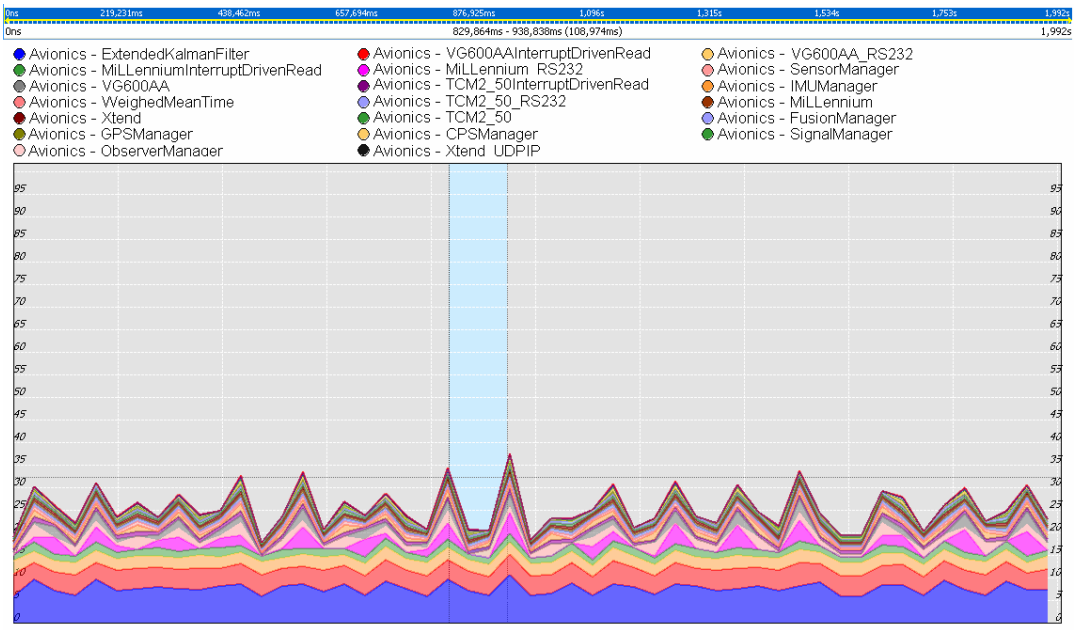

Figura 8-21: Distribuição de consumo de processamento da aplicação AvionicsRealTimeSoftware - Processamento significativo (>1us/s).

Na Figura 8-22 é apresentado a distribuição do consumo de processamento de todas as tarefas que juntas somam 95% do processamento da aplicação que no caso inclui a própria aplicação bem como *drivers* e interrupções utilizados pela mesma.

Nesta figura verifica-se uma informação importante. Os principais consumidores de processamento são, em ordem decrescente:

1. Filtro de Kalman Estendido
2. *Driver* da serial da VG600AA (IMU);
3. *Thread* de leitura da serial da VG600AA (IMU);
4. *Driver* da serial do MiLLennium (GPS);
5. Interrupção do *driver* da serial da VG600AA (IMU);
6. *Thread* de montagem da mensagem da serial da VG600AA (IMU);
7. *Thread* de leitura da serial do MiLLennium (GPS);
8. *Thread* de montagem da mensagem da serial do MiLLennium (GPS);
9. Interrupção do *driver* da serial do MiLLennium (GPS);
10. Agente **SensorManager**;

Assim, caso seja necessário otimizar o desempenho de processamento do sistema, a implementação do filtro de Kalman deve ter o foco. Já as *threads* seguintes, correspondem à leitura da porta serial bem como a montagem da mensagem que provem desta porta, que, na configuração do item 8.3, eram as tarefas que consumiam o maior processamento.

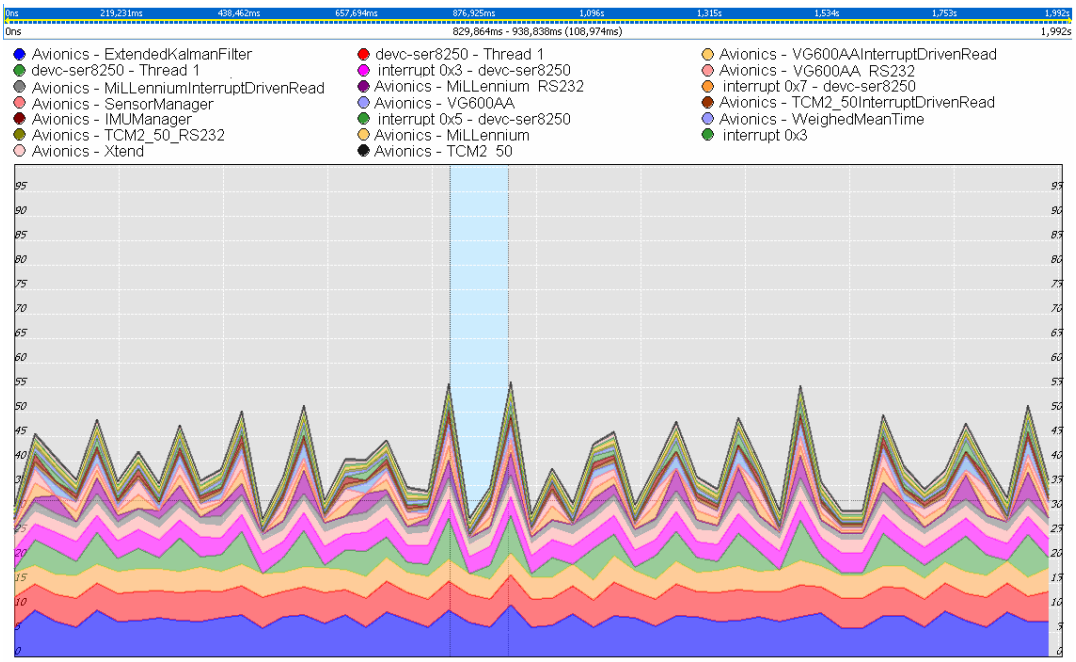

Figura 8-22: Distribuição de consumo de processamento de 95% do processamento de aplicação.

Na Figura 8-23, tem-se um diagrama de linha do tempo com filtro de apresentação para o *driver* de rede e aplicação. Neste, é possível ratificar a peridiciosidade do envio de mensagens da aplicação para a rede. Estas são as mensagens de envio de telemetria para a estação base.

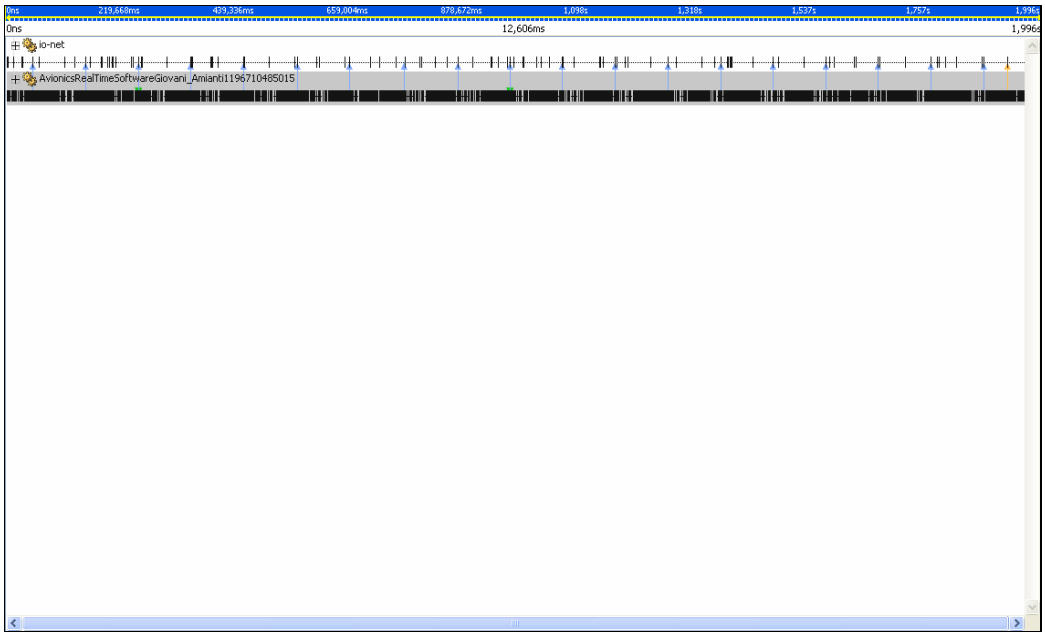

Figura 8-23: Diagrama de linha do tempo da aplicação com foco no envio de telemetria pela rede em tempo de instrumentação de 2s.

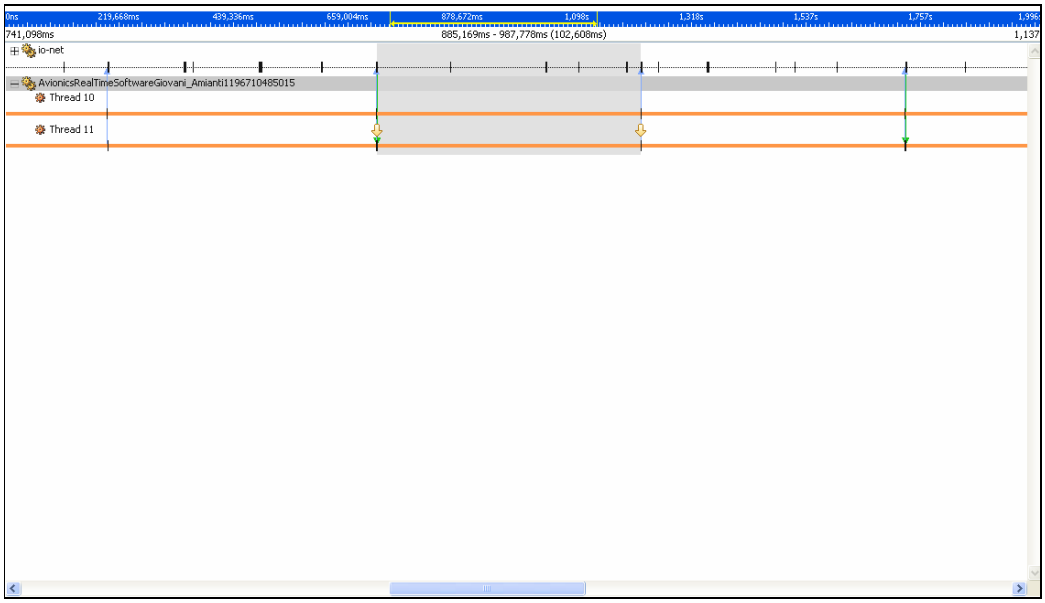

Figura 8-24: Ampliação do diagrama de linha do tempo da aplicação com foco no envio de telemetria pela rede em tempo de instrumentação de 300ms.

Na Figura 8-24, tem-se uma ampliação da linha do tempo que representa a comunicação da aplicação com o *driver* de rede com medição do período entre envios de mensagens. Como a *thread* de comunicação com a rede *ethernet* possui a menor prioridade do sistema e esta está atendendo ao requisito RT1, pode-se inferir que todo o sistema está atendendo ao RT1.

Na Figura 8-25, verifica-se detalhes de interrupções e comunicação entre processos que ocorrem em um período de operação, ou seja, 100ms. Na primeira linha, observa-se a interrupção gerada pela serial da VG600AA (IMU) que possui um *baud rate* de 38400bps e freqüência de 85 Hz, na segunda linha observa-se a interrupção gerada pela serial da TCM2-50 (CPS) que possui um *baud rate* de 9600bps e freqüência de 10 Hz, e na terceira linha observa-se a interrupção gerada pela serial do MiLLennium (GPS) que possui *baud rate* de 54600bps e freqüência de 10 Hz. Já na quarta linha observa-se a interrupção da rede que é compartilhada pela instrumentação do *kernel* e a aplicação assim, apresentando mais interrupções que o esperado. Nas quinta, sexta e sétima linhas têm-se respectivamente o *driver* de serial da VG600AA, o *driver* da rede, e o *driver* serial da TCM2-50 e MiLLennium. Finalmente na ultima linha tem-se o processo da aplicação.

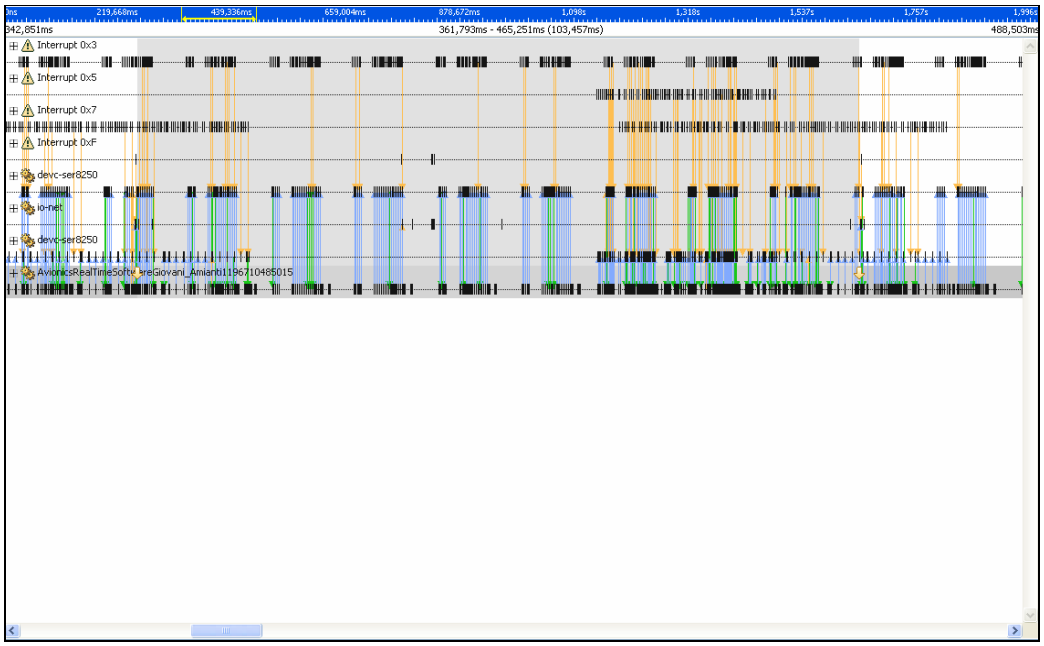

Figura 8-25: Ampliação do diagrama de linha do tempo da aplicação e derivados em tempo de instrumentação de 100ms.

8.4 Considerações finais deste capítulo

Este capítulo apresentou os resultados de parte dos testes de integração realizados nesta pesquisa, nas configurações de operação WMT e EKF.

Inicialmente foi definido o plano de testes de integração incluindo o hardware de testes, o ambiente de implantação de testes, testes funcionais e testes de desempenho.

Os requisitos funcionais foram mapeados pelos testes funcionais. Somente os casos de uso “Escrever dados e *log* na caixa preta”, “Configurar sensores VG600AA (IMU), MiLLennium (GPS) e TCM2-50 (CPS)”, “Receber correção DGPS” e “Fundir dados sensoriais pelo método WMT e EKF” não foram aprovados completamente pelos testes e a solução destes problemas serão propostas para trabalhos futuros. Por exemplo, no caso de uso “Fundir

dados sensoriais pelo método WMT e EKF", os requisitos funcionais do ponto de vista computacional foram completamente atendidos, que era um dos objetivos desta pesquisa, já os requisitos de qualidade e estabilidade da estimação, que depende do modelo, não foram atendidos completamente.

Já os requisitos temporais foram avaliados em testes de desempenho compreendendo testes de corretude, de processamento e de eventos. Por meio da instrumentação de aplicação, no diagrama de seqüência, foi aprovada a corretude do funcionamento do sistema. Por meio da instrumentação do *kernel*, foram aprovados os requisitos temporais, principalmente o *deadline* de 100ms que foi atendido com um consumo de processamento de 17,5% na configuração WMT e 38,3% na configuração EKF.

9 CONCLUSÕES

9.1 Considerações técnicas

Esta dissertação apresentou um trabalho que focou no desenvolvimento do sistema aviônico de um VANT com requisitos de homologação. Este trabalho compreendeu tanto a definição de uma metodologia direcionada à homologação como um ciclo de projeto completo de parte sistema aviônico do VANT Apoena I, mais especificamente o sistema de navegação e comunicação.

No capítulo 2, comprovou-se a carência de estudos ou de divulgação técnica na área de aviônicos para VANTS com requisitos de homologação. Mesmo em projetos que se destacam pela qualidade do desenvolvimento de software observou-se a deficiência de diversas análises imprescindíveis para a homologação que por sua vez foram detalhadas nesta dissertação.

O capítulo 3 apresentou um resumo do projeto BR-UAV detalhando a plataforma, aviônica e sistema controle autônomo; que forneceram a base conceitual para o projeto do software aviônico embarcado.

No capítulo 4, foi proposta uma metodologia de desenvolvimento de sistemas aviônicos de VANTs com requisitos de homologação. Esta metodologia foi seguida em todo o desenvolvimento revelando-se muito eficiente, pois reduziu significantemente o tempo de desenvolvimento e ainda aprimorou a qualidade do produto final, se comparado ao projeto piloto, que foi uma versão de validação do conceito da aviônica proposto implementada por métodos convencionais (codificação manual).

A norma de homologação aviônica DO-178B, neste trabalho, restringiu-se a um elemento de orientação na definição do processo de desenvolvimento bem como das ferramentas. Obviamente ainda não se atingiu um software passível de homologação, pois o processo de homologação é muito superior ao escopo da presente pesquisa. Entretanto formou-se uma base sólida para a continuidade do trabalho no caminho da homologação.

O processo de desenvolvimento proposto na metodologia revelou-se adequado, pois, engloba todo o ciclo de desenvolvimento, desde a captura dos requisitos, projeto de sistema, projeto de software, implementação e testes.

Das ferramentas escolhidas somente a integração entre o Rhapsody e o MatLab Simulink e o gerador de código MatLab RealTimeWorkShop apresentaram deficiências, que serão tratadas na próxima seção.

No capítulo 5, foram capturados com sucesso os requisitos do SANT (composto por operador, plataforma, hardware e software), do sistema aviônico (composto por hardware e software) e, por sua vez, do software embarcado. O atendimento aos requisitos do software embarcado foi validado pelos testes do software embarcado resultante deste trabalho.

No capítulo 6, foi apresentado o projeto do software aviônico embarcado com foco no projeto e realização de uma arquitetura de software genérica e extensível que possa ser utilizada não somente no projeto BR-UAV, mas também em outros projetos de robôs móveis futuros.

Na GESAM, foi adotado o paradigma de agentes com características de autonomia, comportamento reativo, comportamento deliberativo, concorrência, encapsulamento e interface. Das arquiteturas confiáveis e seguras apresentadas também no capítulo 6, foi adotado na GESAM um híbrido dos padrões HRP, DRP e SEP.

Embora a generalidade e extensibilidade não tenham sido efetivamente avaliadas na GESAM, pois esta não foi implementada em outras plataformas, avaliações preliminares durante o desenvolvimento apontaram fatores indicativos da presença destas características na GESAM.

O primeiro sensor (que pode ser encarado como um módulo) implementado foi o CPS que custou 8 semanas de desenvolvimento. Já o segundo sensor implementado, o GPS, custou 1 semana de desenvolvimento. Finalmente o último sensor implementado nesta versão custou somente 3 dias de desenvolvimento. Esta redução de tempo evidencia a melhoria da capacidade da GESAM adaptar-se à inclusão de novos módulos, indicando extensibilidade.

Por outro lado, os módulos (no caso os agentes) desenvolvidos na GESAM, não se aderem somente ao VANT Apoena I, mas a diversas outras plataformas. No caso, por exemplo, do sistema de navegação desenvolvido, o mesmo pode ser aplicado tanto a VANTs (aviões, helicópteros, dirigíveis) como também a UGVs e USV sem qualquer modificação. Já em UUVs o sistema de navegação pode ser aplicado com limitações. Em AUVs o módulo de GPS funcionaria somente quando a plataforma emergisse. No ROV, o módulo de GPS poderia ser retirado, pois não tem efeito nesta classe de plataforma. Assim, fica evidente que o sistema de navegação da GESAM é altamente genérico, cabendo a trabalhos futuros verificar a generalidade em outros sistemas.

No capítulo 7, foi apresentado o projeto de detalhes da arquitetura GESAM, mais especificamente, os padrões de estruturação de dados, abstrações e implementação de classes.

Finalmente, no capítulo 8, foram apresentados os resultados de parte dos testes de integração realizados na GESAM.

Todos os requisitos funcionais foram mapeados por testes funcionais. Somente os casos de uso “Escrever dados e *log* na caixa preta”, “Configurar sensores VG600AA (IMU), MiLLennium (GPS) e TCM2-50 (CPS)”, “Receber correção DGPS” e “Fundir dados sensoriais pelo método WMT e EKF” não foram aprovados completamente pelos testes e a solução destes problemas serão propostas para trabalhos futuros. Já os requisitos temporais foram avaliados em testes de desempenho nas configurações de operação remota com observações WMT e de operação remota com observações EKF. Por meio da instrumentação de aplicação,

no diagrama de seqüência, foi aprovada a corretude do funcionamento do sistema. Por meio da instrumentação do *kernel*, foram aprovados os requisitos temporais, principalmente o *deadline* de 100ms que foi atendido com um consumo de processamento de 17,5% na configuração WMT e 38,3% na configuração EKF.

Neste contexto, pode-se inferir que os dois objetivos da pesquisa: - **propor uma metodologia de projeto do sistema aviônico de VANTs; - projetar, implementar e testar uma arquitetura de software aviônico embarcado de VANT;** foram alcançados com contribuições nas áreas de: metodologia de desenvolvimento de software aviônico homologável, sistemas de tempo real crítico, captura de requisitos de sistema e software, arquitetura de software, navegação e comunicação.

9.2 Histórico do desenvolvimento, dificuldades e facilidades

Esta pesquisa herdou um software aviônico desenvolvido pelo autor em seu projeto de formatura realizado em 2005. Este software aviônico foi desenvolvido no QNX4 (antiga versão do QNX) cujas funcionalidades eram aquisição dos sensores VG600AA (IMU), MiLLennium (GPS) e TCM2-50 (CPS) e armazenamento das informações em um *BlackBox*. Entretanto, não houve compatibilidade significativa entre o código implementado em QNX4 e o QNX Neutrino. Assim, para evitar tal problema de compatibilidade em projetos futuros, e não perder mais desenvolvimentos, foi adotado, no projeto de mestrado, o padrão POSIX.

Assim, o primeiro passo da pesquisa foi instalar o QNX Neutrino no PC104 e o QNX Momentics no computador de desenvolvimento. Nesta etapa, foram encontradas sérias dificuldades na instalação do QNX Neutrino no PC104, principalmente, devido à incompatibilidade com o barramento SCSI, que foi resolvido com o "*Verbose*" da inicialização do QNX, tomando 3 meses da pesquisa. Depois de resolvido este problema, o conjunto de desenvolvimento QNX Neutrino/Momentics apresentou ótimo funcionamento. O aprendizado do QNX Neutrino, dado o conhecimento do QNX4 foi rápido e fácil, principalmente pela qualidade dos manuais. O fato de o QNX Momentics ser uma IDE baseada no eclipse acelerou o aprendizado. Suas ferramentas de: *debug*, *system profile*, *application profile*, *memory analisys*, *code coverage* foram de fácil aprendizado principalmente pela qualidade dos tutoriais fornecidos pela QNX.

Com o conjunto de desenvolvimento em QNX totalmente operante, foram desenvolvidas aplicações de comunicação serial, comunicação *ethernet*, criação e gerenciamento de *threads*, passagem de mensagens entre *threads* e escrita em arquivo que foram base para aprendizado e testes. A utilização do padrão POSIX facilitou muito tal trabalho, pois, encontra-se uma grande quantidade de referências pelo fato deste ser o padrão utilizado pelo Linux que por sua vez possui uma grande comunidade de desenvolvedores. As referências POSIX em Linux formaram uma base de exemplos que foram significativamente compatíveis com o QNX Neutrino, exceto pequenos fatores de adaptação como nomes de *drivers* entre outros. De posse desse conjunto de aplicações testes, desenvolveu-se a leitura

serial da VG600AA (IMU), MiLLennium (GPS) e TCM2-50 (CPS), e o envio destas informações via escrita na *ethernet* para a estação base. Assim obteve-se uma versão conceitual do software aviônico, provando sua viabilidade, entretanto codificado a mão.

O próximo passo foi a utilização do Rhapsody em SysML na fase de engenharia de sistemas. A instalação do Rhapsody foi fácil, entretanto sua utilização apresentou sérias barreiras de aprendizado. O tutorial de SysML do Rhapsody é completo (extenso), porém confuso, não ensinando muito bem as potencialidades da ferramenta. Por outro lado, a empresa Telelogic disponibiliza seminários *on-line* que são extremamente eficientes no ensino da ferramenta.

O próximo passo foi a utilização do Rhapsody em C++ na fase de engenharia de software. A instalação do Rhapsody em C++ apresentou graves problemas com a conexão entre o QNX Momentics e o Rhapsody. Embora a promessa seja de uma conexão "*plug and play*", mesmo com a ajuda dos fornecedores brasileiros do Rhapsody, a empresa ANACOM, a efetivação da conexão somente foi realizada após 2 meses de árduo trabalho.

Já a utilização do Rhapsody C++, assim como no SysML, apresentou sérias barreiras de aprendizado. O tutorial do Rhapsody em C++ também é completo (grande), porém também é confuso, não ensinando muito bem as potencialidades da ferramenta. E como no caso do SysML, a telelogic disponibiliza seminários on-line que são extremamente eficientes no ensino da ferramenta em C++.

A partir do Rhapsody em C++ operacional, no que se refere ao desenvolvimento do software no Rhapsody em C++, o desempenho foi melhor que o esperado sem dificuldades intransponíveis pois, todos os problemas podem ser encarados como problemas comuns no desenvolvimento de software embarcado.

A seguir, o EKF foi desenvolvido no Simulink do MatLab em parceria com o aluno de projeto de formatura Fábio Doro Zanoni. Obteve-se desempenho de estimação regular com cerca de 2 a 4 vezes a precisão desejada. Mesmo assim, caminhou-se para a sua implementação em tempo real, ou seja, geração de código pelo RealTimeWorkshop a partir dos modelos do Simulink, deixando a melhoria do desempenho de estimação para trabalhos futuros, pois o objetivo deste trabalho focava a computação.

Primeiramente, deve-se entender o conceito da integração Rhapsody & Simulink. No modelo feito no Rhapsody é inserido um objeto com o estereotipo Simulink. Este objeto conecta-se ao modelo do Simulink e ao respectivo código gerado pelo RealTimeWorkshop. Para efetivar esta conexão, são passados os parâmetros do modelo em Simulink e do código do RealTimeWorkshop ao Rhapsody. O Rhapsody automaticamente representa as portas do Simulink como portas de fluxo no objeto e o preenche com o código gerado pelo RealTimeWorkshop. Para a finalização da conexão, deve-se definir um parâmetro tempo de amostragem que especifica a taxa que o Rhapsody amostra o modelo Simulink. Aqui que se verifica um grande problema na integração Rhapsody & Simulink, pois o EKF em questão era

aperiódico, pois, os passos de propagação e atualização ocorrem conforme a chegada de dados dos sensores e não em um período pré-definido, o que inviabiliza a utilização da integração Rhapsody & Simulink.

Assim a integração entre o Rhapsody e o MatLab foi deixada para trabalhos futuros, pois isto não prejudicaria significativamente nem desenvolvimento ou a filosofia de utilização da geração automática de código. Isso porque todas as matrizes resultantes das simulações do Simulink tiveram seus códigos gerados automaticamente através da ferramenta de manipulação simbólica do MatLab, o que evita os principais erros matemáticos tratados na seção 4.5. Para tanto, somente foram implementadas manualmente, pelo aluno de projeto de formatura Fábio Doro Zanoni, as operações matemáticas como soma, subtração, multiplicação, transposição e inversão de matrizes, bem como o algoritmo de Kalman, sendo estas operações passiveis de testes extensivos.

O algoritmo mais complicado de se tratar foi à inversão de matrizes, pois dependendo de como este foi implementado, pode-se gerar grandes distúrbios no EKF, devido a erros de precisão, truncamento e inversão de matrizes com determinantes pequenos. O algoritmo implementado para realizar a inversão neste trabalho foi o Gauss Jordan, devido sua facilidade de implementação (com relação aos outros métodos) e também da possibilidade de realizar a inversão de matrizes de qualquer tamanho com uma boa precisão.

Finalmente entrou-se no período de testes de integração. O debug de modelo auxiliou muito os testes unitários, onde somente uma pequena parte do código está instrumentado influenciando insignificantemente no processamento do computador tanto embarcado como de desenvolvimento. Entretanto, nos testes de integração, houve uma grande dificuldade de reduzir a instrumentação de forma que existisse capacidade computacional para observar, pela animação do modelo, o funcionamento da aplicação.

Além disso, verificou-se uma grande dificuldade de visualização das *threads*, devido ao seu grande número e pelo fato do Rhapsody não gerar código de com nome nas *threads* embora no modelo seja especificado este parâmetro. Esta deficiência vem do fato do Rhapsody implementar código seguindo o padrão POSIX que, por sua vez, não tem nenhuma função de nomeação de *threads*. Em sistemas *multithreads*, fica difícil identificá-las nos testes. Entretanto, o QNX Neutrino tem uma função não POSIX, cuja sintaxe é pthread_setname_np(hThread, name) onde o sufixo _np significa non-POSIX. Sendo assim, foi implementada uma modificação da OXF que nomeia *threads*, entretanto esta ainda apresenta problemas que precisam ser depurados o que ficará para trabalhos futuros.

Os resultados do desempenho de estimação dos testes de bancada foram satisfatórios, pois o filtro se estabilizou rapidamente em torno da posição estática, tendo uma oscilação inferior a 2 graus da atitude e uma oscilação inferior a 1m na latitude e longitude e a 1,7m na altitude, enquanto que o GPS fornecia um erro de aproximadamente 2.3m na latitude e longitude e 3,7 m na altitude. Já nos testes dinâmicos, a estimação simplesmente não

funcionou, pois, o filtro diverge alguns segundos após o início da estimação. Assim, deve se ter um esforço de sintonização do filtro em ambientes dinâmicos.

No contexto apresentado neste histórico, verifica-se que o grande esforço da pesquisa (aproximadamente 30% do tempo) se concentrou em tarefas que são geralmente consideradas fáceis e que não trazem reconhecimento como instalação e integração, conforme se pode observar na Figura 9-1. Entretanto, após atravessar essas barreiras, tarefas consideradas difíceis, foram facilmente transpostas com o auxílio das ferramentas selecionadas.

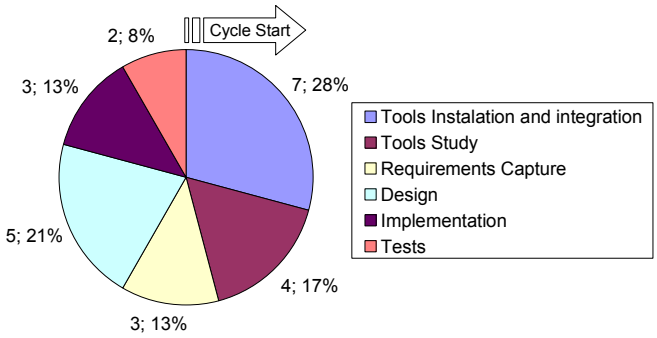

Figura 9-1: Distribuição do trabalho desta pesquisa.

9.3 Sugestões para trabalhos futuros

As sugestões para trabalhos futuros, propostas nesta seção, compreendem desde solução de pequenos problemas encontrados no desenvolvimento do software embarcado, passando por propostas de projetos completos de sistemas, de software e de navegação..

9.3.1 Solução de problemas encontrados na pesquisa

- Desenvolver uma forma de integração Rhapsody & Simulink

Esta integração deve ser aperiódica com *trigger* gerado por evento no Rhapsody.

- Comunicação partindo de uma porta em direção a várias portas

O Rhapsody implementa esta comunicação, entretanto, somente uma das portas recebe a mensagem.

- Implementar nome nas *threads*

Melhorar a modificação da OXF proposta no capítulo 17 (ANEXO – Modificações na Rhapsody OXF) que nomeia as *threads*, mas que ainda apresenta problemas.

- Resolver problema de sincronização com a mensagem da VG600AA (IMU).

- Implementar uma rotina de segurança contra sobre armazenamento do agente **BlackBox**.

- Desenvolver as rotinas de tratamento de erros dos sensores

Deve-se considerar que as rotinas de detecção de erros já foram implementadas.

- Solucionar problemas de estabelecimento de conexão no sistema DGPS.

9.3.2 Aperfeiçoamento do software

- Implementação da GESAM em outros veículos não tripulados

Esta implementação tem o objetivo de avaliar capacidade de generalidade e extensibilidade da GESAM. Obviamente com o desenvolvimento em um único veículo, como apresentado neste trabalho, não é suficiente para levantar todas as necessidades da arquitetura para classificá-la com genérica e extensível, sendo assim, na realidade o principal resultado da aplicação da GESAM em outros veículos será seu aperfeiçoamento. Na Figura 9-2 tem-se uma proposta dos veículos nos quais a GESAM pode ser implementada.

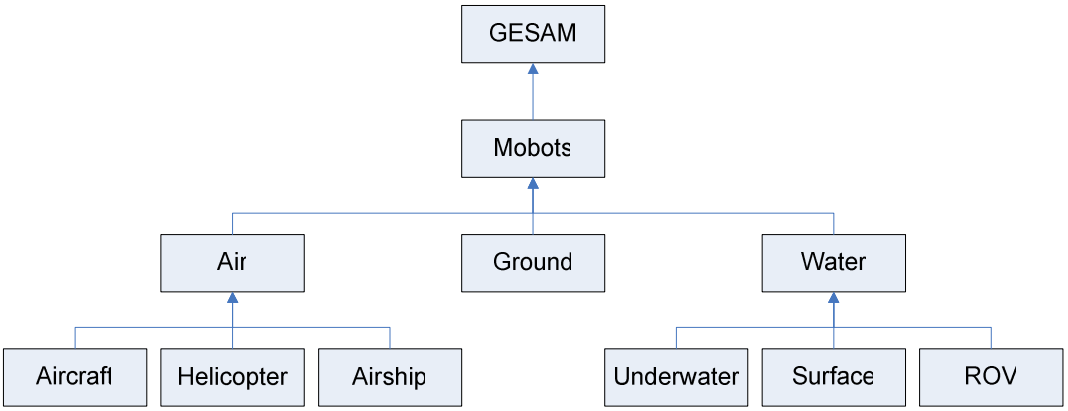

Figura 9-2: Veículos não tripulados.

- Desenvolver o agente **Manager**

Este agente deve ser capaz de receber e gerenciar uma missão estabelecida pelo operador na estação base e enviada pelo *data-link*. Quando em operação, este agente deve ser capaz de receber uma reconfiguração de missão. O agente **Manager** é um campo fecundo para a aplicação de técnicas de inteligência artificial, principalmente aprendizado e adaptação.

- Desenvolver o agente **Autopilot**

Este agente deve integrar as funções de controle, guiamento, planejamento e anti-colisão segundo a proposta de utilização do MatLab que é um campo a ser mais bem desenvolvido.

- Desenvolver o agente **Actuator**

Este agente deve realizar a interface entre os outros agentes e os módulos de atuação bem como ser uma camada gerenciadora de alto nível.

- Implementar, no agente **Communicator,** outras formas de comunicação

As formas de comunicação podem ser:

- com a estação base através de GPRS e Satélite;

- com outros robôs através de comunicação de coordenação;

- e com o sistema de controle de tráfego aéreo através de *transponder*.

- Implementar, no agente **Observer,** métodos de fusão no nível de ponto, característica e símbolo.

- Reiterar o desenvolvimento do software aviônico com foco na homologação

O processo de homologação é composto por uma série de auditorias das autoridades de homologação por meio de documentação ou de reuniões presenciais. Entretanto, no contexto acadêmico, como o objetivo não é obter um produto homologado e sim melhorar substancialmente a qualidade do software embarcado em robôs móveis, o processo de homologação poderia ser simulado pelos próprios pesquisadores seguindo as respectivas normas disponíveis na RTCA e FAA. As auditorias de processo de desenvolvimento de software seguem, em geral, o *Job Aid* (FAA-DA, 2007) que prevê 4 momentos de envolvimento: - SOI 1 (documentação): é realizada para a revisão dos planos solicitados pela DO-178B, no caso PSAC, SDP, SVP, SCMP, SQAP; - SOI 2 (presencial): é realizada já com o processo em andamento, próximo da fase de projeto do software. O objetivo desta fase é verificar a aderência do processo aos planos aprovados; - SOI 3 (presencial): é realizada quando as atividades de verificação já estão substancialmente efetivadas. O objetivo desta fase também é verificar a aderência do processo aos planos aprovados; - SOI 4 (documentação): é realizada para dar fecho ao processo. Em geral ela verifica se não há pendências abertas antes da aprovação final da autoridade.

9.3.3 Aperfeiçoamento do Sistema

- Desenvolvimento do *firmware* do Módulo de Atuação

Na primeira versão considerada básica, este *firmware* deve comunicar-se com a rede embarcada, de onde recebe o valor das referências de atuação, e gerar os respectivos sinais PWM para os servos e os respectivos sinais digitais para controles de luzes e seleção de câmeras de navegação.

Em versões posteriores devem-se impor os seguintes requisitos:

- o módulo de atuação deve receber sinais do transmissor RC, traduzi-los e envia-los à ECU central que por sua vez armazenará esta informação que será útil para experimentos de identificação dinâmica;

- o módulo de atuação deve deliberar sobre o controle da plataforma, ou seja, se as referências para atuação serão oriundas do RC ou da rede (controle remoto ou controle autônomo), ou ainda em caso de falha do sistema, deve acionar o procedimento de emergência que inclui a abertura de pára-quedas;

- no caso de sistema redundante, os módulos de atuação redundantes devem coordenar as responsabilidades de atuação e deliberação de controle para a garantia de segurança.

- Desenvolvimento do *firmware* do Módulo de *Data-link*

Na primeira versão considerada básica, a rede sem fio será ponto a ponto e o *firmware* do *data-link* deve ter duas funções. 1) quando um *data-link* recebe uma mensagem de outro *data-link*, o receptor realiza um *broadcast* na rede. 2) quando um *data-link* recebe uma mensagem da rede, envia esta mensagem para o outro *data-link*.

Em versões posteriores devem-se impor os seguintes requisitos:

- no caso de sistema redundante, a rede sem fio será ponto a multiponto e os *data-link*s redundantes devem coordenar as responsabilidades de comunicação;

- o *firmware* da versão básica, deve ser utilizado somente em condições degradadas, pois o objetivo é aperfeiçoar do esquema de comunicação em *broadcast* para *multicast*. Neste esquema de *multicast* o *firmware* do *data-link* deve reconhecer o tipo de mensagem que está recebendo pela rede sem fio e realizar um *multicast* para o grupo de módulos interessados em receber a informação. Por exemplo, mensagens do sistema de atuação devem ser enviadas para todos os módulos de atuação e mensagens de informação de missão ou de coordenação entre veículos devem ser enviadas para todas as ECUs centrais.

- Desenvolvimento dos módulos AD/DD-Ethernet para ADB, TD e ED

Embora tenha sido desenvolvido (implementado e testado) o *driver* da placa de aquisição analógica do PC104 em QNX Neutrino, verifica-se que a distribuição dos sensores na rede embarcada apresenta diversos benefícios, entre eles: - desenvolvimento do *firmware* de acesso a hardware em micro-controladores é muito mais fácil que desenvolvimento de *drivers* em QNX Neutrino; - módulos micro-controladores com conversão AD e comunicação *ethernet* que possuem especificações semelhantes de uma placa de aquisição AD/DD com barramento PC104, apresentam custo 75% menor; - melhoria da confiabilidade do sistema, pois caso a ECU Central falhe, os módulos sensores conseguem enviar a telemetria diretamente ao *data-link*, com qualidade da informação degradada, mas que ainda mantém o mínimo de segurança do sistema. - potencializa o desenvolvimento de sistema HILS, conforme será visto nos próximos itens. Na primeira versão considerada básica, o *firmware* dos módulos AD/DD-Ethernet devem aquisitar os sinais analógicos e digitais e realizar um *broadcast* na rede.

Em versões posteriores devem-se impor os seguintes requisitos:

- no caso de sistema redundante, a operação continua a mesma;

- o *firmware* da versão básica deve ser utilizado somente em condições degradadas, pois o objetivo é aperfeiçoar do esquema de comunicação em *broadcast* para *multicast*, assim como no *data-link*. Neste esquema de *multicast* o *firmware* dos módulos AD/DD-ethernet deve realizar um *multicast* para o grupo de módulos interessados em receber a informação. Por exemplo, em operação normal, deve ser realizado um *multicast* para todas as ECU Centrais e em caso de operação degradada, deve realizado um *multicast* para todos os *data-link*s.

- Desenvolvimento dos módulos Serial-Ethernet para IMU, GPS e CPS

Verifica-se que a distribuição dos sensores seriais na rede embarcada apresenta menos benefícios se comparados aos do módulo AD/DD-Ethernet, entretanto os benefícios dos módulos Serial-Ethernet ainda são atraentes: - melhoria da confiabilidade do sistema, pois caso a ECU Central falhe, os módulos sensores conseguem enviar a telemetria diretamente ao *data-link*, com qualidade da informação degradada, mas que ainda mantém o mínimo de segurança do sistema. - potencializa do desenvolvimento de sistema HILS, conforme será visto nos próximos itens.

- Desenvolvimento de um HILS

Com os desenvolvimentos dos itens anteriores, são abolidos da arquitetura de hardware a comunicação serial, analógica ou digital entre módulos aviônicos. Isto permite que um HILS com comunicação *ethernet* seja facilmente integrado à rede aviônica. Na primeira

versão, os sensores devem ser retirados da rede embarcada e o HILS será composto somente por software que simule além do ambiente e plataforma, os sensores e atuadores.

Em uma versão posterior devem-se impor os seguintes requisitos:

- o HILS deve ser composto também por mecanismos mecânicos de atitude e deslocamento, permitindo a introdução dos sensores e atuadores reais.

9.3.4 Propostas de solução dos problemas da Navegação

Neste ponto vale lembrar que o requisito da navegação é obter estimações com erro da ordem de 0,2m e 0,5º para se desenvolver o sistema de decolagem e pouso automáticos.

- Implementar formas computacionalmente mais estáveis do filtro de Kalman

Observa-se em testes estáticos que após 1 hora de funcionamento, o EKF original diverge. Já o EKF com modificações na propagação da covariância, conforme proposto por Joseph, (Brown, 1997) diverge em 1,5 horas. Este comportamento somente é observado em testes estáticos, pois em testes dinâmicos o filtro diverge após alguns segundos depois de inicializado e considera-se que isto esteja ligado à falhas na modelagem e não por problemas computacionais. Deve-se considerar que o investimento em técnicas computacionalmente estáveis vale a pena, pois as mesmas são aplicáveis tanto no filtro de Kalman linear como o estendido.

- Melhorar o modelo de estimação

O primeiro passo do projeto do sistema de navegação foi desenvolver o filtro de Kalman linear com uma modelagem muito simplificada. Embora a baixa qualidade da modelagem, a qualidade de estimação simulada era razoável (1m e 1º). Em seguida, com o objetivo de obter estimações melhores, desenvolveu-se um modelo cinemático livre de simplificações com o EKF e como resultado obteve-se uma qualidade de estimação simulada semelhante à linear (pouco menor que 1m e 1º). Ou seja, aumentou-se significativamente a complexidade computacional e não se obteve a correspondente melhoria da qualidade de estimação simulada. Mas, para comprovar esta assertiva, deve-se comparar o desempenho dos dois filtros no mundo real. Assim, recomenda-se a implementação on-line do filtro linear, principalmente com o objetivo de capacitação do pesquisador em sintonizar o filtro, pois o filtro linear é muito mais simples e consequentemente mais fácil de detectar os efeitos da sintonização. Em seguida, deve-se reiterar o projeto do EKF e então trabalhar na sua sintonização. Neste contexto, serão dados passos consistentes para alcançar o requisito de qualidade de estimação do sistema de navegação. Caso estes requisitos não sejam atendidos com os equipamentos de baixo custo disponíveis, ainda existe uma última proposta antes de utilizar sensores de melhor qualidade. A proposta é a inserção do modelo dinâmico da

plataforma no processo de estimação conforme Koifman (1999). Na Figura 9-3, pode-se observar a redução do erro de posição com o sistema assistido pela dinâmica.

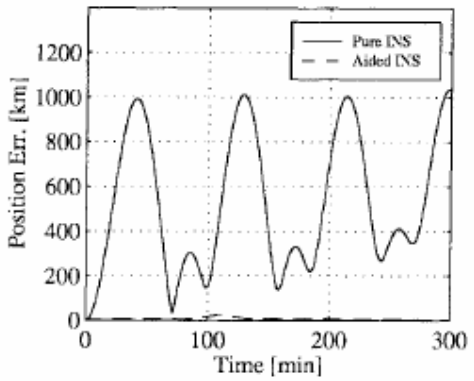

Figura 9-3: Erro de posição de navegação inercial pura e assistida pela dinâmica (Koifman 1999).

10 REFERÊNCIAS

AMIANTI, G; BARROS E. A. **Desenvolvimento da Aviônica de um VANT – Veículo Aéreo Não Tripulado.** In: Congresso Brasileiro de Automática, 16., 2006, Anais do XVI Congresso Brasileiro de Automática, Salvador, 2006.

AMIANTI, G.; MECCHI, A. C. L. **Projeto aviônico de um VANT.** 2005, p. 165. Projeto de Formatura – Escola Politécnica, Universidade de São Paulo, São Paulo, 2005.

AMIANTI, G.; DANTAS, J. L. D. **Projeto de desempenho, estabilidade e controlabilidade do VANT Apoena I.** Relatório parcial de projeto de desempenho, estabilidade e controlabilidade. Relatório interno Xmobots, 2006.

ANEEL. **Agência Nacional de Energia Elétrica.** Disponível em: < http://www.aneel.gov.br/ >. Acesso em: 20, mai, 2006.

ATKINS, E. M.; et al. Solus: **An Autonomous Aircraft for Flight Control and Trajectory Planning Research.** In: American Control Conference, 1998, Proceedings of the American Control Conference, Philadelphia, p.689-693, Jun. 1998.

AVIONICS, **Europe's Answer: UAVs in Controlled Airspace.** Avionics Magazine. Disponível em: <http://www.aviationtoday.com/av/categories/military/1022.html> Acessado em: 11 nov. 2003.

AWARD, M.; KUUSELA, J.; ZIGLER, J. **Object-Oriented Technology for Real-Time Systems.** Prentice Hall, NJ, 1996.

BAUER, B.; MÜLLER, J. P.; ODELL, J. **Agent UML: A formalism specifying multiagent interaction.** P. Ciancarini e M. Wooldridge (eds), Agent-Oriented Software Engineering, Springer-Verlag, Berlim, Germany, p.91-103, 2001.

BICHLER, L.; RADERMACHER, A.; SCHÜRR, **A. Evaluating UML Extensions for Modeling Real-time Systems.** In: International Workshop on Object-Oriented Real-Time Dependable Systems, 7, 2002, Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, p.271-278, 2002.

BRYSON, M.; SUKKARIEH S. **Building a Robust Implementation of Bearing-only Inertial SLAM for a UAV.** Journal of Field Robotics, v.24, n.1-2, p.113–143, Fev., 2007.

BROWN, R. G.; HWANG, P. Y. C. **Introduction to Random Signals and Applied Kalman Filtering.** 3.ed., New York, John Wiley & Sons , 1997.

BUENO S. S.; et al. **Robotic Airships for Exploration of Planetary Bodies with an Atmosphere: Autonomy Challenges.** Autonomous Robots, n.14, p.147–164, Mar.-Mai., 2003.

BURNS A., WELLINGS A. **Real-time and Programming Languages.** Terceira edição, Addison-Wesley, 2001.

CAMPBELL, M.; et al. **Experimental Demonstrations of Semi-Autonomous Control.** In; Proceedings of the 2004 American Control Conference, Boston, Jun.–Jul. 2004, n.6, p.5338-5343, 2004.

CASA - Civil Aviation Safety Authority (Austrália) (2006). **Improvements and amendments to CASR Part 101 – Unmanned aircraft and rocket operations.** Disponível em: < http://www.casa.gov.au/rules/1998casr/101/index.htm>. Acessado em junho de 2007.

COGNITIVELABS. **Simple Reaction Time Test.** Disponível em: <http://cognitivelabs.com/mydna_speedtestno.htm>. Acessado em: 10 jul. 2007.

DEDICATED SYSTEMS EXPERTS. **RTOS Evaluation Project.** Disponível em: < http://www.omimo.be/encyc/buyersguide/rtos/rtosmenu.htm >. Acesso em: 11 maio 2006.

DO-178B. **Software Considerations in Airborne Systems and Equipment Certification**. Radio Technical ComMission for Aeronautics, Washington DC, 1992.

DO-254. **Design Assurance Guidance for Airborne Electronic Hardware**. Radio Technical ComMission for Aeronautics, Washington DC, 2000.

DO-304. **Guidance Material and Considerations for Unmanned Aircraft System**. Radio Technical ComMission for Aeronautics, Washington DC, 2007.

DOUGLASS, B. P. **Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns.** Addison Wesley, p.749, 1999.

DOUGLASS, B. P. **Real-Time UML Workshop for embedded Systems.** Newes, Oxford, p.408, 2007.

ELFES, A. **Incorporating Spatial Representations at Multiple Levels of Abstraction in a Replicated Multilayered Architecture for Robot Control.** Intelligent Robots: Sensing, Modelling, and Planning, New York, 1997. World Scientific Publishers. Invited paper, 1996 International Dagstuhl Seminar on Intelligent Robots, Schloß Dagstuhl, Germany.

ELFES, A.; et al. **Modelling, Control and Perception for an Autonomous Robotic Airship.** Lecture Notes In Computer Science, v. 2238, p. 216–244, 2002.

EVANS, J.; ET AL. **DRAGONFLY A versatile UAV platform for the advancement of air1craft navigation and control.** In: Digital Avionics Systems Conference, 20., v.1, n.14-18, p.1C3_1-1C3_12, Oct 2001.

FAA-DA, FEDERAL AVIATION ADMINISTRATION DESIGN APPROVALS. **Guidance and Job Aids for Software and Complex Electronic Hardware**, USA Federal Aviation Administration.

Disponível em: < http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/guide_jobaid/>. Acessado em: 10 dez. 2007.

FAA-AC, FEDERAL AVIATION ADMINISTRATION ADVISORY CIRCULAR. **AC20-115B**, USA Federal Aviation Administration. Disponível em: < http://rgl.faa.gov/Regulatory_and_Guidance_Library/rgAdvisoryCircular.nsf/0/dcdb1d2031b19791862569ae007833e7/$FILE/AC20-115B.pdf >. Acessado em: 10 dez. 2007.

FAR, FEDERAL AVIATION REGULATION. **Part 23: Airworthiness Standards: Normal Category Airplanes, Subpart F – Equipment**, USA Federal Aviation Administration, 2006. Disponível em: <http://ecfr.gpoaccess.gov/cgi/t/text/text-idx?c=ecfr&sid=49ea04697e61a6c36d06352681217052&rgn=div5&view=text&node=14:1.0.1.3.10&idno=14>. Acessado em: 10 dez. 2007.

FABIANI, P.; ET AL. **Autonomous flight and navigation of VTOL UAVs: from autonomy demonstrations to out-of-sight flights.** Aerospace Science and Technology, v.11 n.2-3 p.183-193, Mar.-Apr. 2006.

FARINES, J.; FRAGA, J. DA S.; OLIVEIRA, R. S. **Sistemas de Tempo Real.** Escola de Computação, 2000.

FERREIRA, G. A. N. **Desenvolvimento de uma arquitetura de controle baseada em objetos para um robô móvel aquático.** 137p Dissertação (mestrado) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2003.

FRANKLIN, S.; GRAESSER, A. **Is it an agent, or just a program?: A taxonomy for autonomous agents.** Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Budapest, Hungary, 1996.

GÖKTOGAN, A. H.; et al. **The Real-Time Development and Deployment of a Cooperative Multi-UAV System.** In: INTERNATIONAL SYMPOSIUM ON COMPUTER AND INFORMATION SCIENCES, 2003, p. 576–583, 2003.

HALANG, W. A. **Real-Time Systems: Another perspective.** The Journal of Systems and Software, p.101-108, 1992.

HALL, JR. C. E. **A Real-Time Linux system for autonomous navigation and flight attitude control of an uninhabited aerial vehicle.** In: DIGITAL AVIONICS SYSTEMS CONFERENCE, 20., 2001, Daytona Beach, FL, v.1, p.1A1/1 - 1A1/9, IEEE, 2001.

HALL, JR. C. E. **On board flight computers for flight testing small uninhabited aerial vehicles.** In: MIDWEST SYMPOSIUM ON CIRCUITS AND SYSTEMS, 45., 2002.,v.2, p.II-139 - II-143, IEEE, 2002.

HALL, W. D.; ADAMS, M. B. **Autonomous Vehicle Software Taxonomy.** Proceedings of IEEE Symposium on Autonomous Underwater Vehicle Technology, Washington, DC, 1992.

HALLBERG, E.; KAMINER, I.; PASCOAL, A. **Development of a Flight Test System for Unmanned Air Vehicles.** IEEE Control Systems Magazine, v.19, n.1, p.55-65, Feb. 1999.

HB. **Human Benchmarck: Reaction Time.** Disponível em: <http://www.humanbenchmark.com/tests/reactiontime/stats.php>. Acesso em: 07/11/2007.

HIGHRELY. **DO-178B & DO-254 Questions & Answers.** Disponível em: <http://www.highrely.com/do178b_questions.php>. Acesso em: 12 nov 2007.

HIGHRELY, **Do178 Training.** 2006.

HIGHRELY. **DO-178B Cost And Benefits: what are the true DO-178B costs and benefits; a detailed analysis.** HighRely Whitepaper, 2005. Disponível em: <http://highrely.com/whitepapers.php>. Acesso em: 10 jun. 2007.

HODGE, G.; YE, J.; STUART W. **Multi-Target Modelling for Embedded Software Development for Automotive Applications.** In: 2004 SAE World Congress, Detroit, Michigan, mar. 2004. SAE Technical Papers Series, 2004.

HONG W. E.; et Al. **RT-Linux based Hard Real-Time Software Architecture for Unmanned Autonomous Helicopters.** In: International Conference on Embedded and Real-Time Computing Systems and Applications, n. 11, 2005.

ISO, INTERNATIONAL ORGANIZATION STANDARDIZATION. **Industrial automation systems and integration -- Product data representation and exchange -- Part 233: Systems engineering data representation**, 2008. Disponível em: < http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35465>. Acessado em: 10 jan. 2008.

JOHNSON, B.W. **Design & analysis of fault tolerant digital systems.** Addison-Wesley, Boston, USA, 1988.

JANG, S. J.; LICCARDO, D. **Automation of small UAVs using a low cost MEMS sensor and embedded computing platform.** In: DIGITAL AVIONICS SYSTEMS CONFERENCE, 25., 2006.

KIM, H. J.; SHIM, D. H. **A flight control system for aerial robots: algorithms and experiments.** Control Engineering Practice, n.11, 1389-1400, 2003.

KIM, H. J.; SHIM, D. H.; SASTRY, S. **Flying Robots: Modeling, Control and Decision Making.** In: INTERNATIONAL CONFERENCE ON ROBOTICS & AUTOMATION, Mai. 2002, Washington, DC, v.1, p.66-71, 2002.

KIM J.; SUKKARIEH S. **Autonomous Airborne Navigation in Unknown Terrain Environments.** IEEE Transactions on Aerospace and Electronic Systems, v.40, n.3, p.1031-1045, Jul. 2004.

KIM J.; SUKKARIEH S. **Real-time implementation of airborne inertial-SLAM.** Robotics and Autonomous Systems, v.55, n.1, p. 62–71, Jan. 2007.

KOIFMAN, M.; BAR-ITZHACK, I. Y. **Inertial Navigation System Aided by Aircraft Dynamics.** IEEE Transactions on Control Systems Techonology, v.7, n.4, p.487-493, Jul. 1999.

KRTEN, R. **Getting Started with QNX 4 – A Guide for Realtime Programmers.** Parse Software Devices, 1998.

KUMON, M.; et al. Flight **Path Control of Small Unmanned Air Vehicle**. Journal of Field Robotics, v.23, n.3-4, p 223–244, 2006.

LAPLANTE, P. A. **Real-time systems design and analysis : an engineer's handbook.** 3. ed, IEEE Press, Piscataway, NJ, 2004.

LI, Q.; YAO, C. **Real-Time Concepts for Embedded Systems.** CMP Books, San Francisco, CA, 2003

LOEGERING G.; et al. **The evolution of the Global Hawk & Mald avionics systems**. In: Digital Systems Conference, 18., v.2, p.6A1_1-6A1_8, IEEE, 1999.

LUO, R. C.; KAY, M. G. **Multisensor Integration and Fusion for Intelligent Machines and Systems**. 1995.

MELANSON, P.; TAFAZOLI, S. **A Selection Methodology for the RTOS Market.** In: Data SYSTEMS IN AEROSPACE CONFERENCE, 2003, Prague, 2003.

MÜLLER, J.P. **The design of intelligent agents: a layered approach.** Lecture notes in artificial intelligence, Springer Verlag, Berlin, Germany, 1996.

NELSON, R. C. **Flight Stability and Automatic Control.** McGraw-Hill Book Company, New York, NY, 1989.

ODELL, J.; PARUMAK, H; BAUER, B. **Extending UML for agents.** Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence, AOIS, Austin, TX, pp3-17.2000.

OGATA, K. **Discrete-time control systems.** 2ºEd. p.745, Prentice Hall, 1995.

OLIVEIRA, F. A. **CTA e o Projeto VANT.** In:SEMINÁRIO INTERNACIONAL DE VANT, 1.,2005, São José dos Campos. Apresentação. Disponível em: <http://www.aviacao-civil.ifi.cta.br/svant/Apresetacoes /02_CTA_e_o_Projeto_VANT_CTA_IAE.pdf>. Acesso em: 10 out. 2005.

OSD. Office of the Secretary of Defense. **Unmanned Aerial Vehicles Roadmap 2005 – 2030**, Department of Defense of USA, 2005.

POLIDO, M. F. **Um método de refinamento para desenvolvimento de software embarcado: Uma abordagem baseada em UML-RT e especificações formais.** Tese (Doutorado em Engenharia Mecânica) – Escola Politécnica da USP, p. 184, 2007.

POTTER, B. **Using MathWorks Tools to Develop DO-178B Certified Code.** Matlab Aerospace and Defense Digest, v.1, n.2, 2004. Disponível em: < http://www.mathworks.com/company/newsletters/aero_digest/aug04/Honeywells.pdf>, Acessado em:05 nov. 2007.

POTTER, B. **Achieving Six Sigma Software Quality Through the Use of Automatic Code Generation.** In: MathWorks International Aerospace and Defense Conference, 2005. Proceedings of MathWorks International Aerospace and Defense Conference, 2005.

PURVES, D.; ET AL. **Neuroscience.** 3. ed. Sinauer Associates, Inc., Massachusetts, 2007.

QNX. **The QNX 4 Real-time Operating System.** QNX Software Systems Ltd., 1998.

QNX. **QNX Neutrino Realtime Operating Systems – Building Embedded Systems.** QNX Software Systems Ltd., 2004.

QNX NEUTRINO. **QNX Real-time RTOS.** Disponível em: <http://www.qnx.com/products/neutrino_rtos/>, Acessado em: 11 nov. 2007.

QNX MOMENTICS. **QNX Momentics.** Disponível em: <http://www.qnx.com/products/tools/>, Acessado em: 11 nov. 2007.

RAYMER, D. P. **Aircraft Design: A Conceptual Approach.** 2.ed., AIAA Education Series - American Institute of Aeronautics and Astronautics, Washington, DC, 1992.

RAWASHDEH O. A., CHANDLER G. D.; LUMPP, JR. J. E. **A UAV Test and Development Environment Based on Dynamic System Reconfiguration.** In: WORKSHOP ON ARCHITECTING DEPENDABLE SYSTEMS, INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, St. Louis, Missouri USA, May, 2005. Proceedings of International Conference on Software Engineering, 2005.

REGULAMENTO BRASILEIRO DE HOMOLOGAÇÃO AERONÁUTICA. **RBHA-100. Operação de Veículos Aéreos Não Tripulados.** 2004, Disponível em: <http://www.anac.gov.br/biblioteca/rbha.asp>. Acesso em: 10 jun. 2007.

REGULAMENTO BRASILEIRO DE HOMOLOGAÇÃO AERONÁUTICA **RBHA**. Disponível em: <http://www.anac.gov.br/biblioteca/rbha.asp>. Acesso em: 10 jun. 2007.

ROSKAM, J. **Airplane Design Part I.** 1.ed. DAR Corporation, Lawrence, KA, 1997a.

ROSKAM, J. **Airplane Design Part II.** 1.ed. DAR Corporation, Lawrence, KA, 1997b.

ROSKAM, J. **Airplane Design Part III.** 1.ed.. DAR Corporation, Lawrence, KA, 1997c.

ROSKAM, J. **Airplane Design Part IV.** 1.ed. DAR Corporation, Lawrence, KA, 1997d.

ROSKAM, J. **Airplane Design Part V.** 1.ed. DAR Corporation, Lawrence, KA, 1997e.

ROSKAM, J. **Airplane Design Part VI.** 1.ed. DAR Corporation, Lawrence, KA, 1997f.

ROSKAM, J. **Airplane Design Part VII.** 1.ed. DAR Corporation, Lawrence, KA, 1997g.

ROSKAM, J. **Airplane Design Part VIII.** 1.ed. DAR Corporation, Lawrence, KA, 1997h.

ROSKAM, J. **Airplane Dynamics and Performance.** 1.ed. DAR Corporation, Lawrence, KA, 1997i.

ROSKAM, J. **Flight Dynamics and Automatic Control.** 1.ed. DAR Corporation, Lawrence, KA, 1997j.

SELIC, B. **Turning clockwise: Using UML in the real-time domain.** Communications of the ACM, v.42, n.10, p. 46-54., oct. 1999.

SELIC, B.; GULLEKSON, G.; WARD, P. T. **Real-Time Objected-Oriented Modeling.** John Wiley, NY, 1994.

SILBERSCHATZ, G. **Operating System Concepts.** Sexta edição, Addison-Wesley, 2001.

SIMMONS, R. **Structured control for autonomous robots.** IEEE Transactions on Robotics and Automation, 10(1), 1994.

SHIXIANJUN; JIAKUN, S.; HONGXING, L. **Hardware-in-the-loop Simulation Framework Design for a UAV Embedded Control System.** In: Chinese Control Conference, 25., Harbin, Heilongjiang, Aug. 2006. Proceedings of the 25th Chinese Control Conference, 2006.

STALLINGS, W. **Operating Systems.** Quarta edição, Prentice-Hall, 2001.

SUKKARIEH, S.; et al. **The ANSER Project: Data Fusion across Multiple Uninhabited Air Vehicles.** The International Journal of Robotics Research, v.22, n.7–8, p 505-539, Jul.–Aug. 2003.

TANENBAUM, A. **Modern Operating System.** Prentice-Hall, 2002.

TISDALE, J.; et al. **The Software Architecture of the Berkeley UAV Platform.** In:IEEE International Conference on Control Applications, 2006, Munich, Germany. p.1420-1425, 2006.

VASCONCELLOS, Y. **Inteligente e Sem Piloto.** Revista FAPESP. n.84, São Paulo, 2003.

VASCONCELLOS, Y. **Ajuda do Céu.** Revista FAPESP. n.123, São Paulo, 2006.

VIDOLOV B.; MIRAS J.; BONNET S. **AURYON – A Mechatronic UAV Project Focus on Control Experimentations.** In: International Conference on Computational Intelligence for Modelling, Control and Automation, and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, IEEE, 1072 - 1078, 2005.

WOOLDRIDGE, M.; JENNINGS, N.R. **Intelligent Agents: Theory and Practice.** In: Knowledge Engineering Review, Cambridge University Press, 1995.

ZANONI, F. D. **Desenvolvimento de filtro de KALMAN on-line com requisitos tempo-real para a navegação de VANTs.** 2006, p. 76. Projeto de Formatura – Escola Politécnica, Universidade de São Paulo, São Paulo, 20056

11 ANEXO - RESUMO DOS PROJETOS ESTADO DA ARTE

11.1 Projeto Kyosho

O projeto Kyosho (Figura 11-1) da universidade nacional de Kyungpook (HONG, 2005) concentra-se no estudo de arquiteturas de software para veículos aéreos autônomos.

Figura 11-1: VANT do projeto Kyosho.

Na Tabela 11-1 pode-se observar um resumo das características do sistema aviônico do projeto Kyosho.

Tabela 11-1: Resumo das características do sistema aviônico do projeto Kyosho.

Arquitetura de Hardware	Centralizada
Processamento	x86/ Transmeta Crusoe
Especificação do software	Diagramas de seqüência e escalonamento
Sistema Operacional	RT-Linux
Linguagem	C++
Arquitetura de Software	Hierárquica
Verificação dos requisitos de tempo-real	Deadline

11.1.1 Hardware

A arquitetura de hardware é composta por um computador central (*Main Board* da Figura 11-2) com processador x86 Transmeta Crusoe e por uma placa microcontrolada de aquisição e geração de sinais (*Onboard Computer* da Figura 11-2) onde se conectam tanto sensores como acelerômetros, giros, CPS, GPS, e de pressão; quanto atuadores e receptor RC.

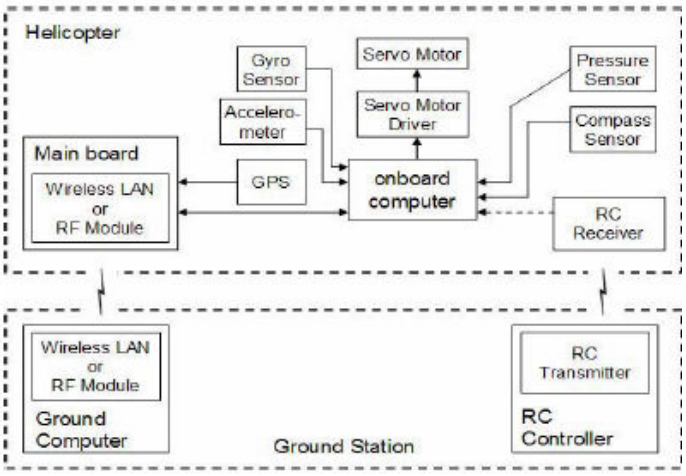

Figura 11-2: Arquitetura de hardware do projeto Kyosho.

11.1.2 Software

Hong (2005) propõe uma arquitetura de software hierárquica com o objetivo de suprir a deficiência de arquiteturas de software mais flexíveis capazes de suportar reconfiguração dinâmica de tarefas concorrentes sem a perda de qualquer *hard deadline* das tarefas de tempo real, pois, segundo ele, a maior parte dos pesquisadores considera somente problemas de controle, o que também foi evidenciado no capítulo 2.

Um esboço da arquitetura de software proposta por Hong (2005), para garantir a execução de tarefas em tempo real, pode ser observado na Figura 11-3. A arquitetura hierárquica possui quatro níveis: *Hardware*, *Execution*, *Service Agent* e *Remote User Interface*.

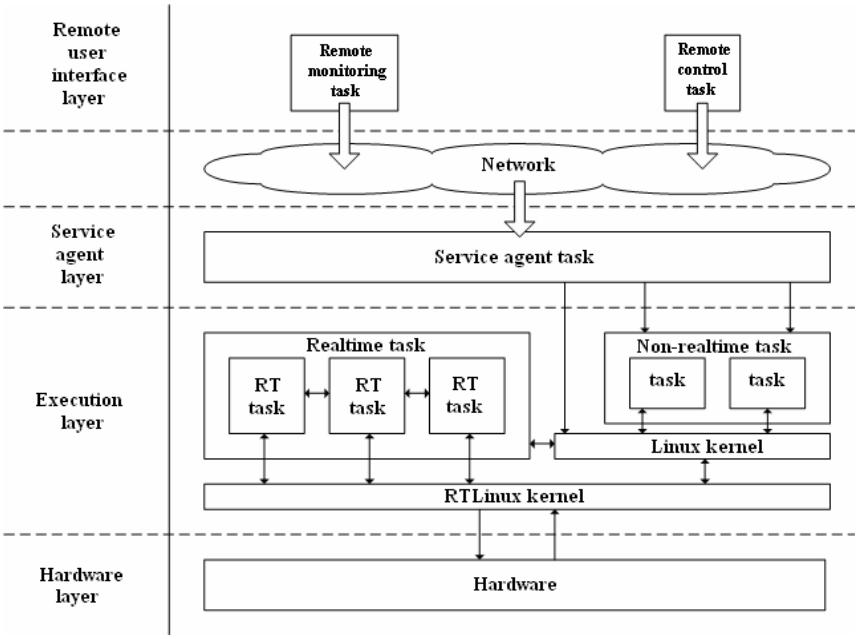

Figura 11-3: Arquitetura de software hierárquica do projeto Kyosho.

O nível de Hardware é composto por sensores, atuadores e computadores.

O nível de *Execution*, onde se alocam as tarefas de tempo real, é composto por tarefas de vôo autônomo sendo assim o nível de maior prioridade. Neste nível também podem ser alocadas tarefas que não são de tempo real, entretanto essas terão prioridade inferior às primeiras.

O nível de *Remote User Interface* é composto principalmente por tarefas de monitoramento e controle remoto, mas também, por comunicação do o nível *Service Agent* via rede.

O nível de *Service Agent* é composto pela tarefa servir o monitoramento remoto. A tarefa *Service Agent* recebe requisições de informações da tarefa de monitoramento remoto do nível *Remote User*, através da rede, e decide pelo estado do VANT.

A seqüência de eventos do sistema (Figura 11-4) consiste em obter os dados dos sensores e do controle remoto via a porta serial do *Onboard Computer*. Em seguida o sistema calcula a atitude usando o filtro de Kalman. Baseado na informação da IMU e CPS é calculado em uma tarefa periódica a velocidade angular e linear e atitude. Estas informações são enviadas à base para estar monitorar o vôo. Os dados do GPS também são coletados e enviados para a estação base periodicamente. Na Figura 11-5 tem-se a seqüência de operações.

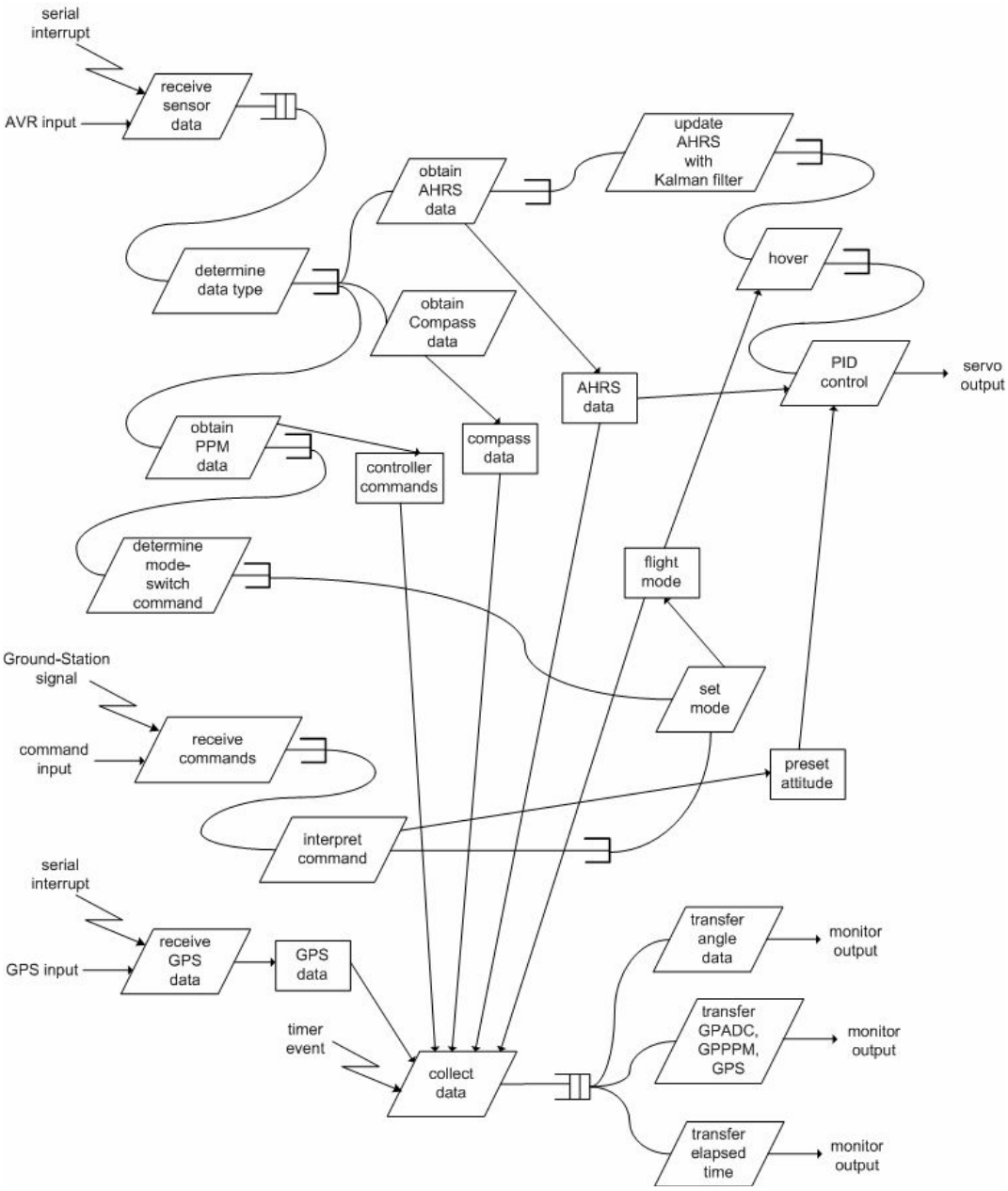

Figura 11-4: Diagrama de seqüência de eventos do projeto Kyosho.

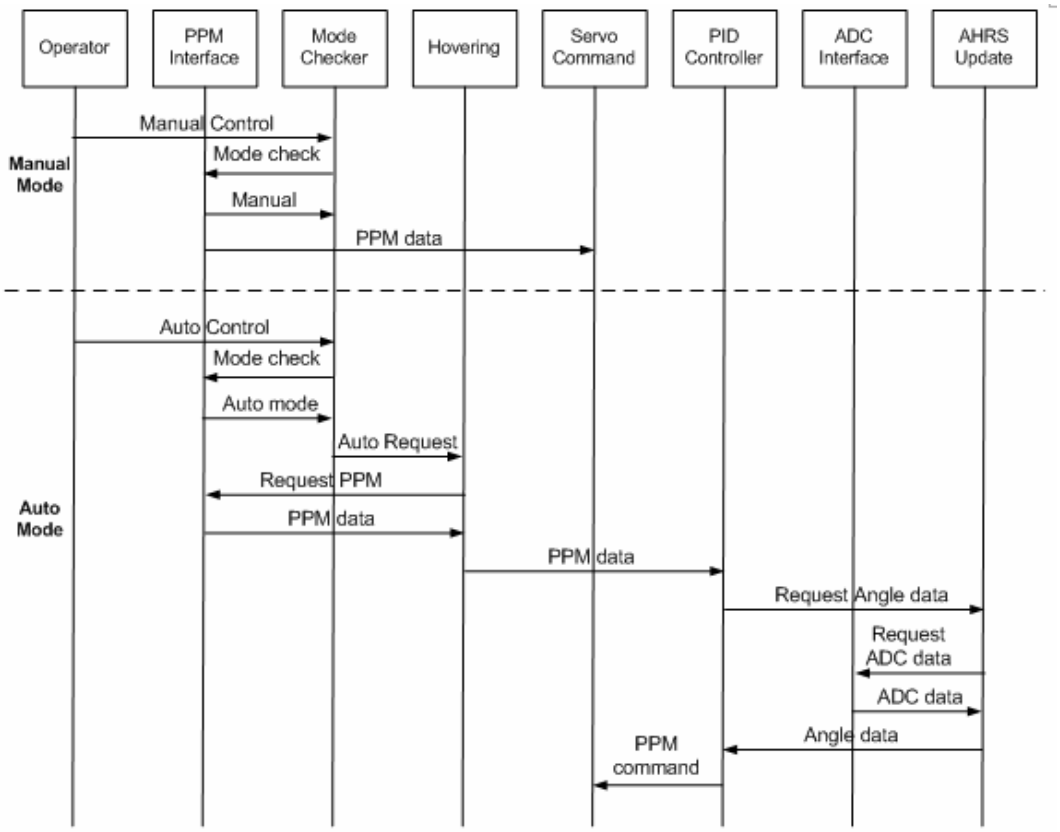

Figura 11-5: Diagrama de seqüência do projeto Kyosho.

11.1.3 Verificação dos requisitos de Tempo real

Como dito anteriormente, A *Main Board* executa tarefas em múltiplos níveis de prioridade. Estas tarefas são divididas em alta, média e baixa prioridade. As tarefas de maior prioridade incluem receber dados de sensores, atualizar o AHRS (kalman), realizar o controle PID e enviar dados requeridos pelo *Onboard Computer*. Na Figura 11-6 pode-se observar que o requisito temporal para processar tais tarefas é de 320 microssegundos e na Figura 11-7 pode-se observar que tal requisito é cumprido uma vez que o tempo de processamento das tarefas de tempo real foi de 258 microssegundos.

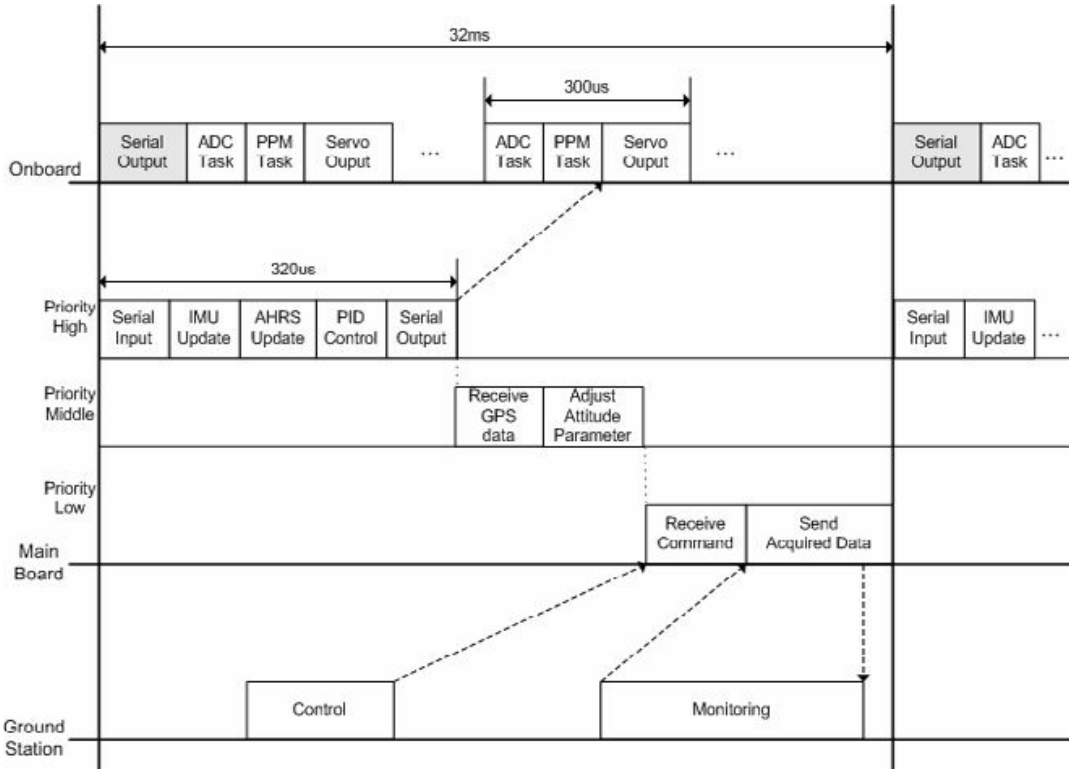

Figura 11-6: Diagrama de escalonamento do projeto Kyosho.

	Linux	RT-Linux
Average Value	32.242 ms	258 µs
Standard Deviation	1.669 ms	24.535 µs
Maximum Value	39.653ms	376 µs
Minimum Value	16.672ms	246 µs

Figura 11-7: Resultado do tempo de processamento das tarefas de tempo real.

11.2 Projeto Brumby

O projeto Brumby (Figura 11-8) da Universidade de Sydney (GÖKTOGAN, 2003; SUKKARIEH, 2003; KIM, 2004; KIM, 2007; BRYSON, 2007) tem o objetivo desenvolver uma plataforma para pesquisas de: aerodinâmica em túnel de vento, desempenho, estabilidade, sistemas de controle de vôo, otimização de trajetórias, determinação de atitude pelo receptor DGPS, determinação de atitude por imagens do horizonte, decolagem e pouso autônomos, SLAM entre outros.

	Mk I	Mk II	Mk III
MTOW	25kg	30kg	45kg
Wing Span	2.3m	2.8m	2.9m
Engine	74cc	80cc	150cc
Power	5Hp	5.5Hp	16Hp
Max Speed	100 Knots	100 Knots	100 Knots
			

Figura 11-8: Versões da aeronave Brumby.

Na Tabela 11-2 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-2: Resumo das características do sistema aviônico do projeto Brumby.

Arquitetura de Hardware	Distribuído
Processamento	x86/PC104
Especificação do software	não declara ou não implementa.
Sistema Operacional	QNX Neutrino/Linux/WinNT
Linguagem	C++
Arquitetura de Software	CommLibX *Framework*
Verificação dos requisitos de tempo-real	Deadlines

11.2.1 Hardware

Cada plataforma é equipada com um *Flight Control Computer* (PC104 da Figura 11-9) no qual são conectados os sensores comuns a todas as plataformas: GPS, IMU, CPS, Inclinômetros, barômetro, velocidade do ar, entre outros.

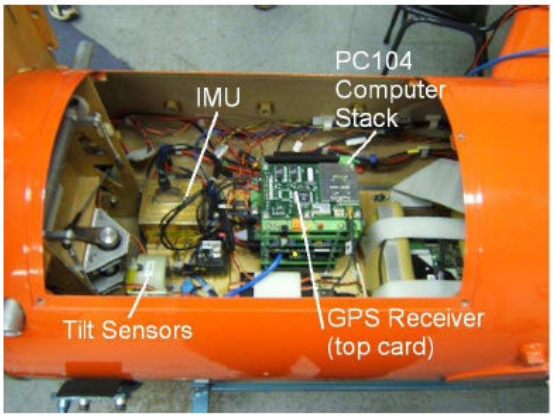

Figura 11-9: Flight Control Computer embarcado no Brumby.

Entretanto o VANT Brumby é utilizado em diversos tipos de pesquisas e por isso sua arquitetura de hardware deve permitir flexibilidade para uma grande variedade de configurações, principalmente do *payload* (Figura 11-10) e por isso desenvolveu-se uma arquitetura distribuída. Assim cada *payload* possui um computador de processamento que é ligado a uma rede *ethernet* pela qual troca informações com outros *payloads*, *Flight Control Computer (*Figura 11-11*),* bem como com outros veículos (Figura 11-12).

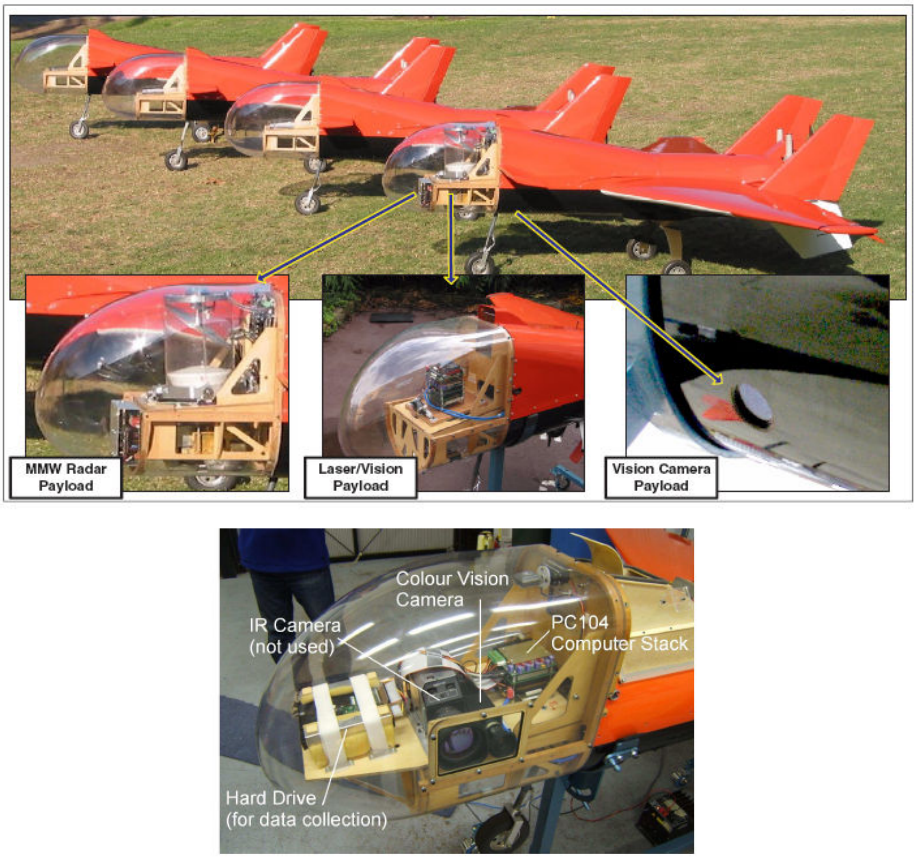

Figura 11-10: Diversos sensores do payload do Brumby.

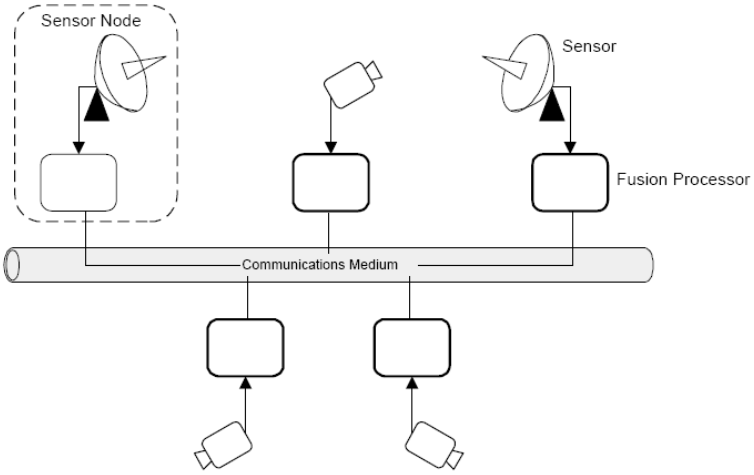

Figura 11-11: Rede de comunicação dentro de uma plataforma aérea Brumby.

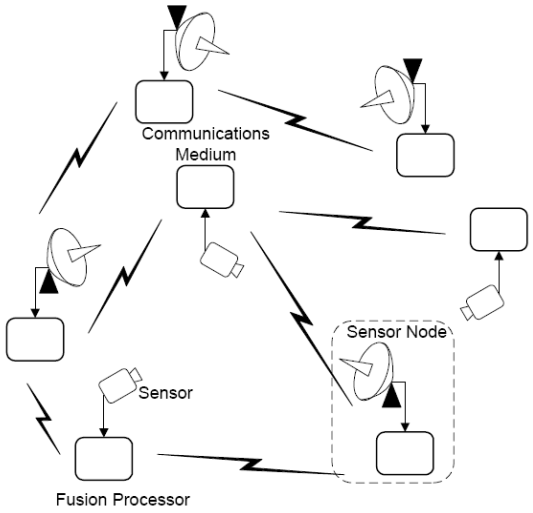

Figura 11-12: Rede de comunicação entre plataformas aéreas Brumby.

11.2.2 Software

Todo o sistema foi implementado em C++, entretanto o piloto automático (controle de vôo, guiamento e navegação) roda no RTOS QNXNeutrino no *Flight Control Computer* e os outros nós do sistema, *payloads* e estação base, rodam diferentes OS, como Linux e WinNT devido ao suporte limitado de *driver*. Assim a comunicação entre os nós passa a ser um problema e, como solução desenvolveu-se uma nova *framework* de comunicação chamada CommLibX. A CommLibX é desenvolvida para cada OS e abstrai a comunicação multi-canal em tempo real.

Neste contexto, não se verifica o emprego de metodologias de desenvolvimento formal de software, nem mesmo a escolha de uma arquitetura de software padrão do sistema exceto a CommLibX e por utilização da IDE Momentics para debug de código.

11.2.3 Verificação dos requisitos de Tempo real

As tarefas mais críticas do sistema, ou seja, as de controle de vôo, operam na freqüência de 50 Hz. A verificação dos requisitos de tempo real restringiu-se a medição do tempo de processamento de cada ciclo obtendo-se 16ms sendo aqui classificação como verificação por deadline.

11.3 Projeto Aurora

O projeto AURORA (Figura 11-13) está sendo desenvolvido pelo CTI (Centro Tecnológico de Informática) (ELFES, 2002; BUENO, 2003) em Campinas, através do Laboratório de Robótica e Visão Computacional, Instituto de Automação e a LTA Brasil Ltda. O objetivo do projeto AURORA é voltado diretamente para questões ambientais, biodiversidades e monitoração climática. A meta do projeto AURORA é o desenvolvimento de um robô dirigível com um nível de autonomia significativo durante todas as fases de sua operação. Isso inclui a

habilidade para realizar missões, navegação automática, diagnóstico de falhas, recuperação, avaliação em tempo real dos sensores e replanejamento de tarefas.

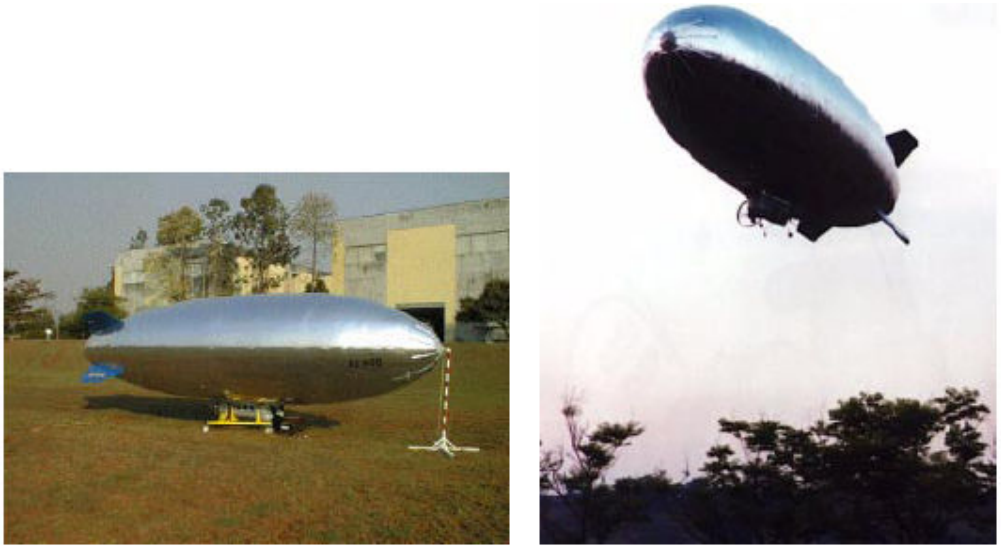

Figura 11-13: Dirigível Aurora I.

Na Tabela 11-3 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-3: Resumo das características do sistema aviônico do projeto Aurora.

Arquitetura de Hardware	Híbrida
Processamento	x86/PC104
Especificação do software	não declara ou não implementa.
Sistema Operacional	RTLinux
Linguagem	C++
Arquitetura de Software	3-*layer structure*/ *Task Control Architecture* / ATLAS
Verificação dos requisitos de tempo-real	não declara ou não implementa.

11.3.1 Hardware

O sistema embarcado inclui: CPU (PC104), sensores, atuadores e comunicação. Todos estes montados na gôndola do dirigível (Figura 11-14). Uma arquitetura de hardware híbrida (Figura 11-15) integra todos os equipamentos embarcados. Sensores de controle e navegação (CPS, IMU, GPS, velocidade do ar, barômetro e rotação do motor) bem como sensores de diagnóstico do veículo (sensores de superfície de controle, temperatura do motor, nível de combustível e bateria) são conectados ao sistema por meio de portas seriais (direto ao PC104) ou por meio uma rede CAN microcontrolada.

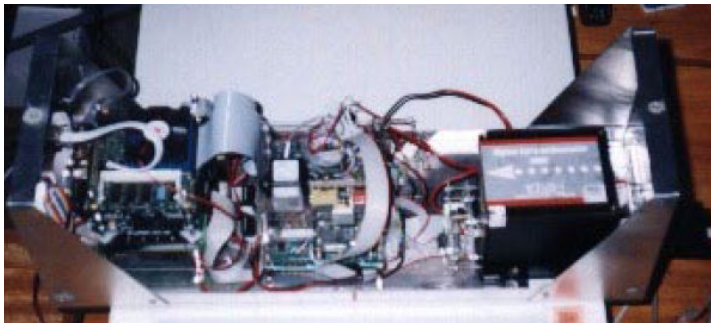

Figura 11-14: Hardware embarcado na gôndola do Aurora.

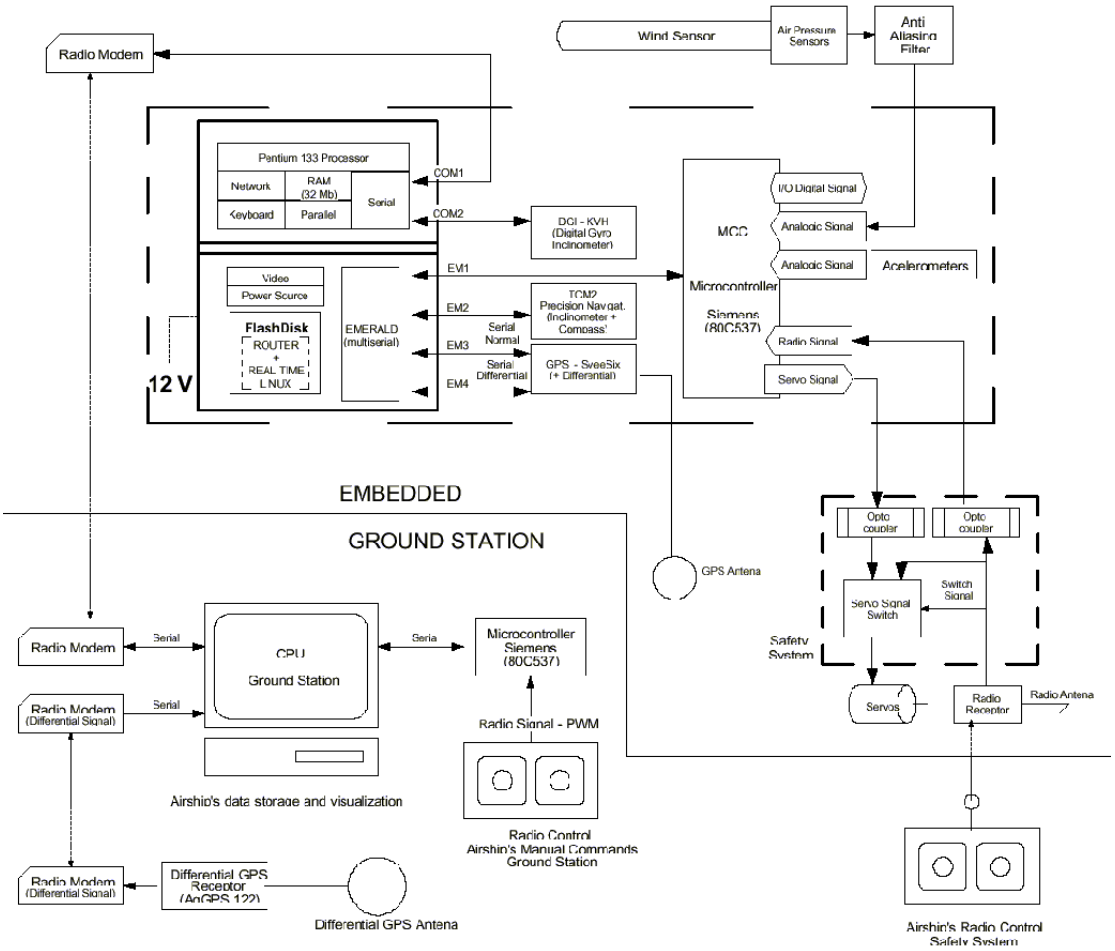

Figura 11-15: Arquitetura de hardware do Aurora.

11.3.2 Software

A arquitetura de software consiste de uma estrutura em 3 níveis, combinada com um método de programação de fluxo de dados em alto nível e um ambiente de desenvolvimento do sistema (Figura 11-16). A complexidade do sistema requere uma arquitetura de controle e comunicação deliberativo-reativa, onde diferentes subsistemas podem rodar

independentemente como *threads* separadas, sendo capaz de trocar informações e de inibir e habilitar umas as outras.

Para facilitar o gerenciamento de processos diferentes em um ambiente distribuído, a arquitetura baseia-se na *Task Control Architecture* (SIMMONS, 1994) que é uma série de primitivas que ajudam no processo de escalonamento e comunicação inter-processos.

A estrutura de controle foi desenvolvida em níveis e multi-rate similar com a arquitetura ATLAS (ELFES, 1997).

Segundo Bueno (2003) foi escolhido para o projeto aurora o RTOS RT-Linux, reconhecido pela confiabilidade e robustez permitindo sua utilização em ambientes com requisitos de tempo real com um espaço em memória relativamente pequeno.

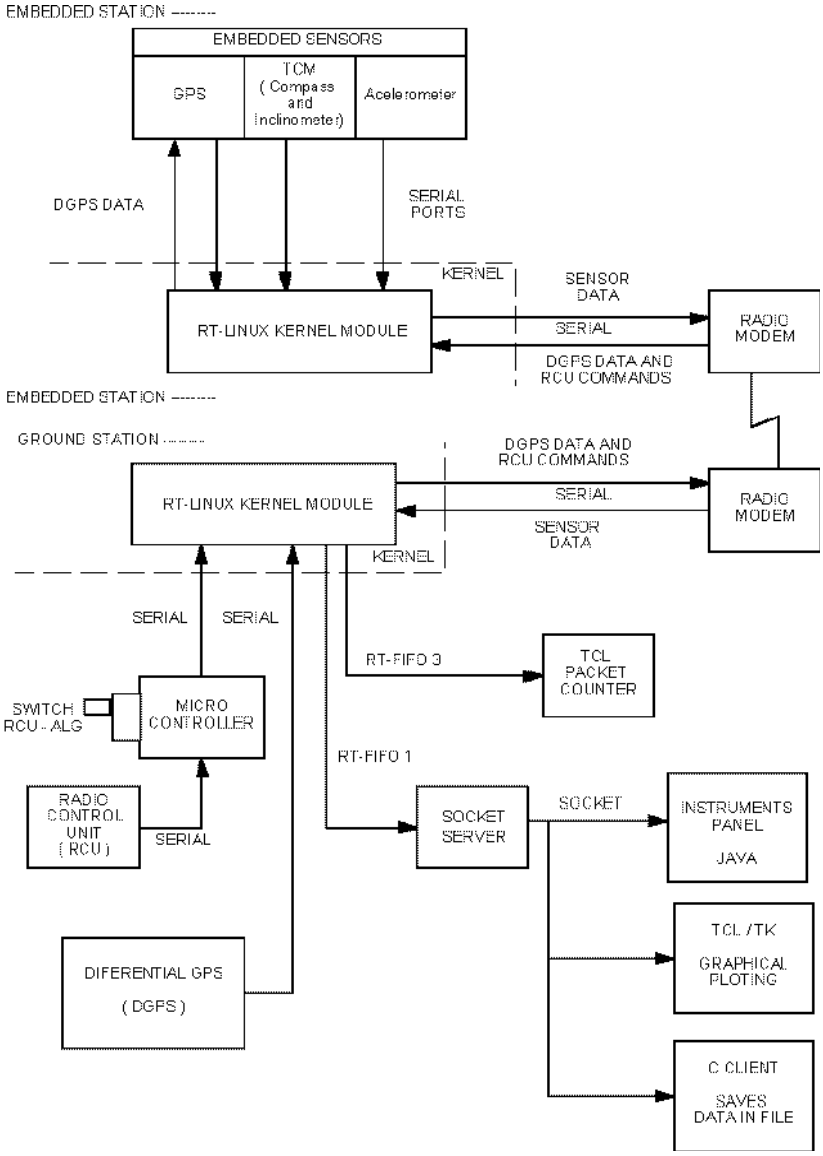

Figura 11-16: Arquitetura de software do Aurora.

11.3.3 Verificação dos requisitos de Tempo real

Os processos rodando no sistema embarcado lêem e enviam dados dos sensores para a estação base bem como executam as estratégias de controle de vôo autônomo, enviando comando aos atuadores. Estas tarefas são executadas a uma freqüência de 10 Hz, entretanto não foi obtido nenhum método que permite evidenciar que tal requisito está sendo cumprido.

11.4 Projeto XAV

O projeto XAV (Figura 11-17) da empresa Crossbow Technology, Inc. (JANG, 2006) foca o desenvolvimento de um VANT de baixo custo utilizando COTS, principalmente MEMs, com o objetivo de uma plataforma robótica *open-souce*.

Figura 11-17: VANT CrossBow XAV.

Na Tabela 11-4 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-4: Resumo das características do sistema aviônico do projeto Brumby.

Arquitetura de Hardware	Centralizada
Processamento	XScale
Especificação do software	Especificação das tarefas e comunicações.
Sistema Operacional	Linux
Linguagem	C++
Arquitetura de Software	Não declara
Verificação dos requisitos de tempo-real	Deadline

11.4.1 Hardware

O VANT (Figura 11-18) é equipado com MNAV *sensor suite* e Startgate CPU (Figura 11-19) que juntos provê toda a arquitetura de hardware centralizada (Figura 11-20).

Figura 11-18: Hardware embarcado no XAV.

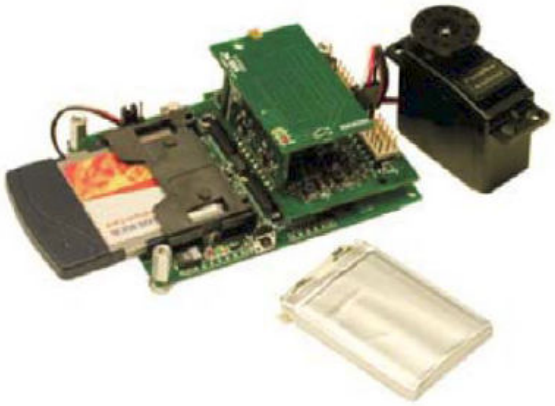

Figura 11-19: MNAV integrado ao Startgate.

O MNAV é um sistema de sensoriamento e um sistema de controle de servos projetado para operar em veículos de radio controle RC. O sistema de sensoriamento inclui acelerômetros, giroscópios, magnetômetros, pressão estática e dinâmica bem como um GPS. JÁ o sistema de controle de servos inclui uma interface para os servos bem como uma interface de recepção PPM. Isto provê controle por software dos servos e interpretação dos comandos do transmissor RC.

O Stargate é uma CPU XScale que se conecta ao MNAV via 51 pinos. No Stargate opera o Linux onde são processados o AHRS e INS usando um filtro de kalman em tempo real.

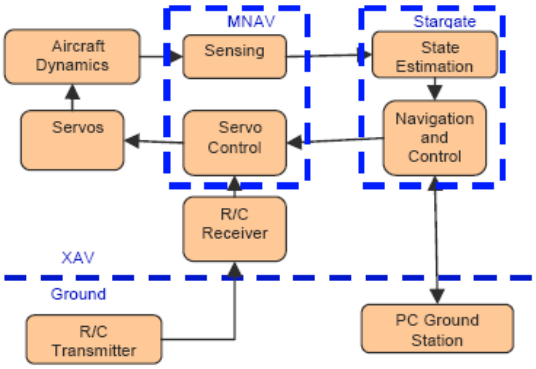

Figura 11-20: Arquitetura de hardware do XAV.

11.4.2 Software

Uma arquitetura de software de tempo real foi desenvolvida para suprir a deficiência de serviços geralmente encontrados em um RTOS, não oferecidos pelo Linux. Nesta arquitetura um único processo é implementado no Linux e este é composto por várias *threads* que são chamadas e escalonadas separadamente. A comunicação entre as *threads* é realizada por memória compartilhada e por *mutexs* (uma especificação POSIX de exclusão mútua). Recursos compartilhados como CPU e I/O são realizados por um algoritmo de escalonamento com passagem de mensagens e variáveis condicionais.

Na Figura 11-21 poder-se observar que a aplicação consiste de *threads* esporádico-periódicas e periódicas cujas prioridades são estáticas sendo as *threads* escalonadas pelo escalonador Round Robin (RR). *Threads* periódicas (INS e Wi-Fi-Out) rodam em uma freqüência fixa, já as esporádico-periódicas (*Data Aquisition*, Wi-Fi-IN), são chamadas por eventos externos. (AHRS, *Control*) *threads* são seguidas pelo *Data Aquisition thread* usando variáveis condicionais.

A prioridade das *threads* é fixada de acordo com o seu tempo de processamento. *Threads* mais rápidas possuem a maior prioridade.

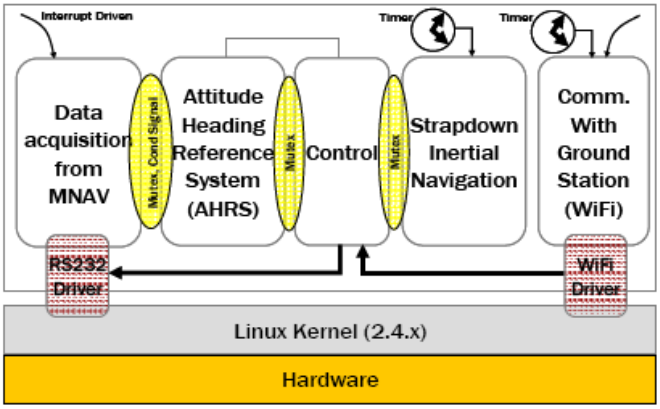

Figura 11-21: Arquitetura de software do XAV.

11.4.3 Verificação dos requisitos de Tempo real

A linha do tempo da implementação da arquitetura proposta pode ser observada na Figura 11-22. A negociação entre *threads*, com o objetivo de usar recursos do sistema é realizada segundo suas prioridade. Isto garante que tarefas críticas sempre completarão dentro de *deadlines* estabelecidos.

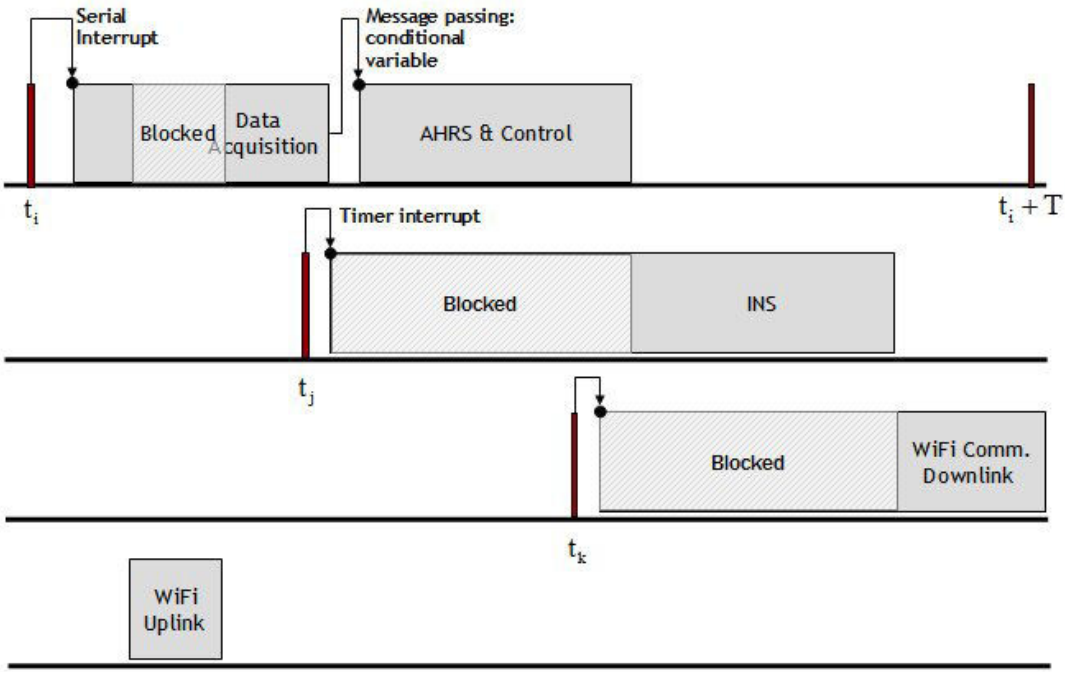

Figura 11-22: Linha do tempo de escalonamento.

11.5 Projeto Berkeley

O projeto Berkeley (Figura 11-23) da Universidade da Califórnia Berkeley (TISDALE, 2006) tem o objetivo de estudar navegação, sistema anti-colisão, e seguidor de alvos baseado em visão bem como cooperação entre vários VANTs.

Figura 11-23: Plataforma VANT Berkeley.

Na Tabela 11-5 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-5: Resumo das características do sistema aviônico do projeto Berkeley.

Arquitetura de Hardware	Centralizado
Processamento	x86/PC104
Especificação do software	Modelos genéricos
Sistema Operacional	QNX Neutrino 6.2
Linguagem	C
Arquitetura de Software	Hierárquica
Verificação dos requisitos de tempo-real	Deadlines

11.5.1 Hardware

A arquitetura de hardware é formada por duas unidades de processamento, o módulo aviônico Piccolo e o PC104.

O Piccolo, produzido pela Cloudcap Technologies, realiza o controle de baixo nível do VANT. Este inclui GPS, giros e Pitot para controle e navegação autônomos. O sistema Piccolo possui comunicação serial permitindo com que este se comunique com o PC104.

O PC104 realiza o planejamento de alto nível e gerencia a colaboração entre VANTs. É nele que são implementados os algoritmos de processamento de imagem.

11.5.2 Software

No projeto do software, implementado em C e no RTOS QNX, foi adotada uma arquitetura hierárquica onde estratégias de planejamento de alto nível foram isoladas das estratégias de controle de baixo nível. Adicionalmente, processos de planejamento de trajetória foram separados de processos de colaboração com objetivo de modularidade e independência dos processos, como pode ser observado na Figura 11-24. Neste caso a modularidade assegura a confiabilidade do sistema e a independência dos processos garante o desenvolvimento rápido e paralelo, ou seja, por vários membros da equipe.

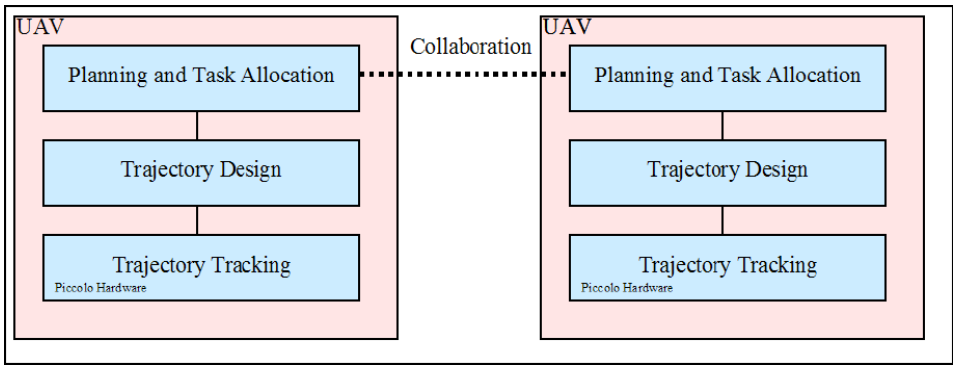

Figura 11-24: Arquitetura hierárquica do VANT Berkeley.

Processos de interface como o Piccolo Interface (que se comunica com o Piccolo), o *Payload* Interface (que se comunica com a estação base) e o A2A *Wireless* (que se comunica com outra aeronave) foram implementados para prover serviços de comunicação específicos.

Como cada comunicação tem seu próprio gerente, o desenvolvedor está isolado de muitos problemas de sincronização frequentemente associados a controle de tempo real.

O processo Supervisor gerencia a colaboração em alto nível, o processo Sensor gerencia sensores de visão e o processo *Controller* realiza os algoritmos seguidores de estruturas lineares e de órbita em ponto específico.

A comunicação entre processos utiliza um esquema *publish and subscribe* PS implementado usando memória compartilhada database chamada de Datahub. O esquema de comunicação OS é frequentemente chamado *generative approach,* pois os dados residem fora do processo que cria e lê o dado. Desde que as operações de leitura e escrita são desacopladas do tempo, não há necessidade de sincronizar as aplicações produtoras e consumidoras. Assim cada aplicação é desacoplada das demais, facilitando a codificação, distribuição e atualizações, pois uma aplicação pode substituir a outra que produz o mesmo dado.

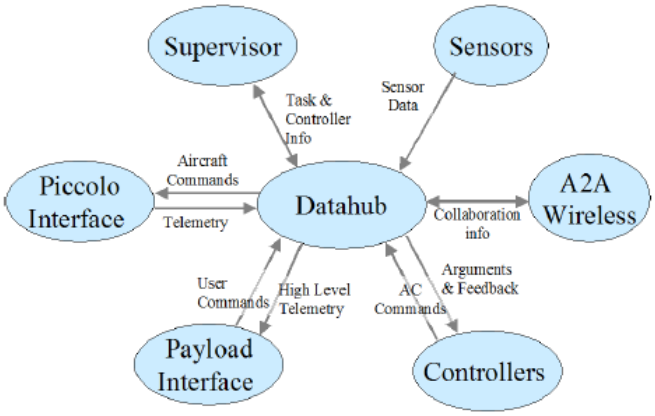

Figura 11-25: Arquitetura de software hierárquica de alto nível do VANT Berkeley.

11.5.3 Verificação dos requisitos de Tempo real

Experimentos foram realizados para determinar se o escalonador do QNX era efetivo em permitir que todos os processos atendam suas *deadlines*. Na Figura 11-26 pode se observar os períodos dos processo Piccolo Interface, *Payload Interface* e *Controller*. Os períodos desejados eram 30, 30 e 75ms, respectivamente. Para um experimento de 7 minutos, a amostra revela que o escalonador do QNX foi efetivo em deixar todos os processos atender aos seus deadlines com pequeno desvio do período desejado.

O processo *Controller* tem restrições temporais mais brandas e também é o processo menos critico e por isso possui a menor prioridade. Devido a esta baixa prioridade, o desvio padrão do período do processo é maior que os demais.

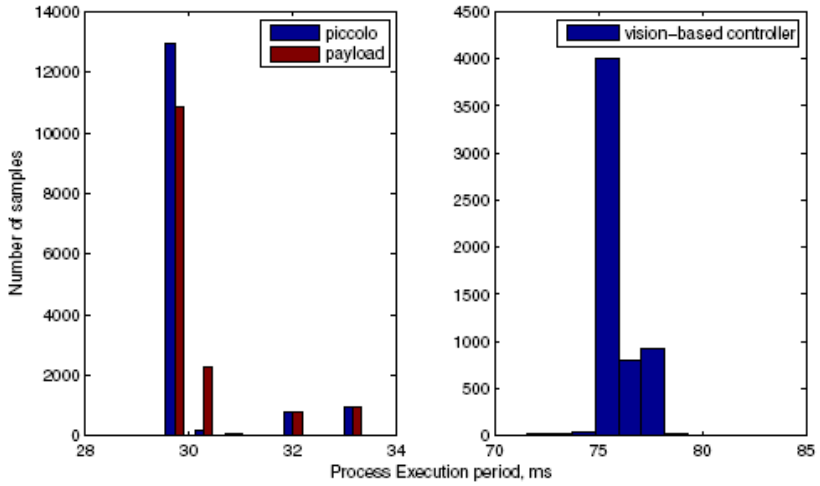

Figura 11-26: Períodos de execução dos processos Piccolo Interface, Payload Interface e Controller.

11.6 Projeto Stingray

O projeto Stingray (Figura 11-27) da Universidade da Carolina do Norte (HALL, 2001; HALL, 2002) tem o objetivo de desenvolver uma plataforma de vôo para testes aerodinâmicos e de controle.

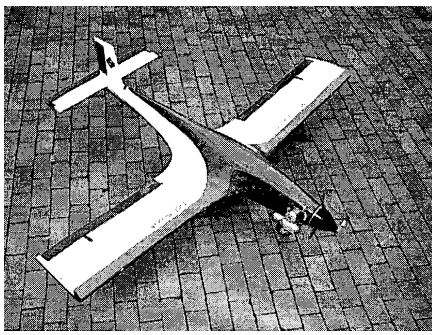

Figura 11-27: Projeto Stingray.

Na Tabela 11-6 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-6: Resumo das características do sistema aviônico do projeto Stingray.

Arquitetura de Hardware	Centralizada
Processamento	PC104
Especificação do software	não declara ou não implementa.
Sistema Operacional	RT-Linux
Linguagem	C
Arquitetura de Software	não declara ou não implementa.
Verificação dos requisitos de tempo-real	Deadlines

11.6.1 Hardware

O uso de RTOS como o RT-Linux, permitiu a utilização de COTS para a arquitetura de hardware centralizada formada por um único PC104, como pode ser observado na Figura 11-28.

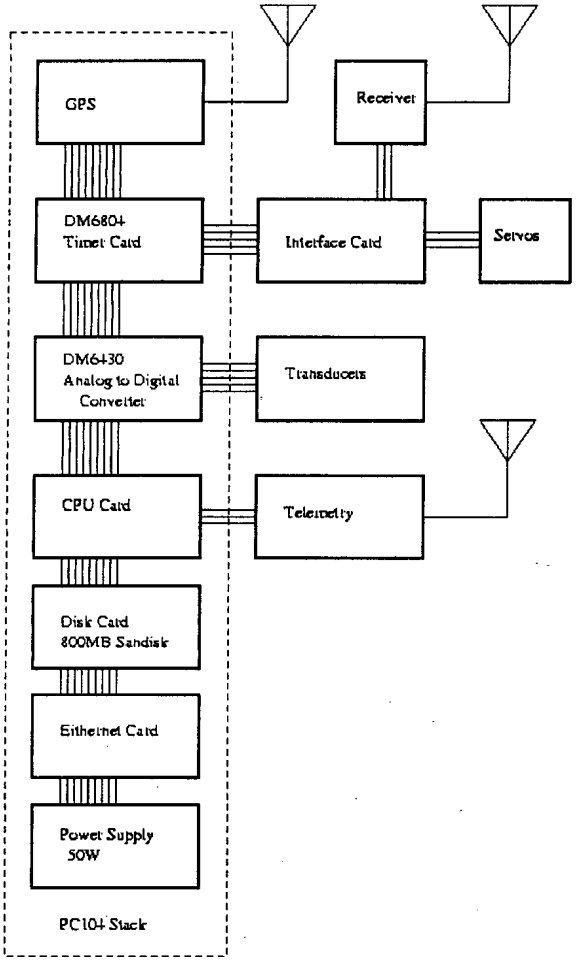

Figura 11-28: Arquitetura de hardware do Projeto Stingray.

11.6.2 Software

O sistema de controle de atitude e de navegação foi implementado em C no RT-Linux. As tarefas rápidas e temporalmente críticas como o controle de atitude foram implementadas como tarefas de tempo real no espaço do *kernel* (Figura 11-29). Já as tarefas mais lentas, como navegação, foram implementadas como tarefas de usuário provendo referências para o controle de atitude.

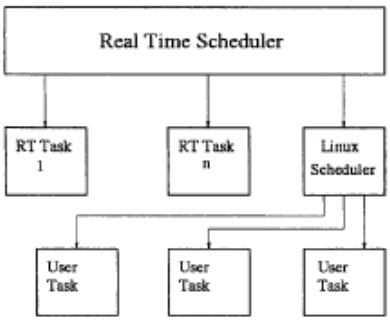

Figura 11-29: Escalonamento utilizando no Projeto Stingray.

11.6.3 Verificação dos requisitos de Tempo real

Foram realizados testes em uma tarefa de tempo real periódica (período de 10ms) para determinar o período médio bem como os desvios e seus resultados podem ser observados na Tabela 11-7.

Tabela 11-7: Resultados de testes de Jitter do RT-Linux.

Mean	10.000 ms
Standard Deviation	4.25 μs
Maximum	10.042 ms
Minimum	9.957 ms

11.7 Projeto Dragonfly

O projeto Dragonfly (Figura 11-30) da Universidade de Stanford (EVANS, 2001) é uma bancada de testes de suporte a novas pesquisas de determinação de atitude por GPS, navegação, controle tolerante a falhas e coordenação multiveículo.

Figura 11-30: Aeronave do Projeto Dragonlfy.

Na Tabela 11-8 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-8: Resumo das características do sistema aviônico do projeto Dragonfly.

Arquitetura de Hardware	Centralizada

Processamento	x86/PC104+
Especificação do software	não declara ou não implementa.
Sistema Operacional	QNX Neutrino
Linguagem	C
Arquitetura de Software	Cliente-Servidor
Verificação dos requisitos de tempo-real	não declara ou não implementa.

11.7.1 Hardware

A arquitetura de hardware centralizada consiste de um x86 PC104+ no qual são conectados GPS e IMU.

11.7.2 Software

A arquitetura de software, conforme pode ser observada na Figura 11-31, é baseada no sistema Cliente-Servidor implementado em C no RTOX QNX Neutrino. Os módulos servidores interfaceiam com os sensores realizando o pré-processamento de dados para entregá-los aos clientes interessados. Já os módulos clientes são as aplicações como *data loggers*, sistemas de controle e navegação.

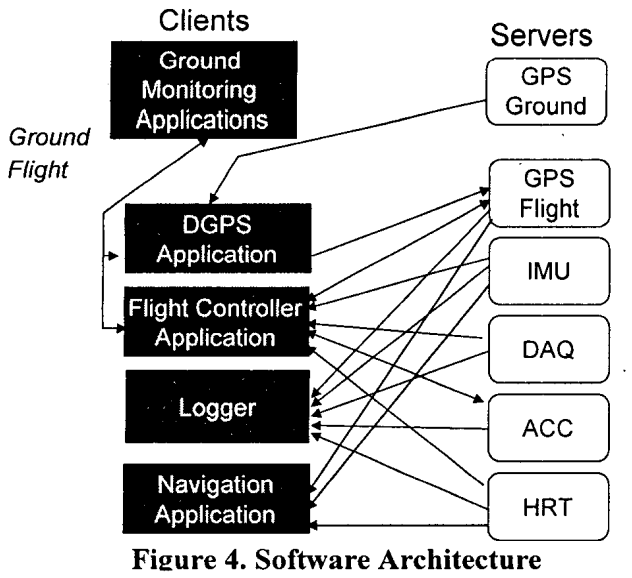

Figura 11-31: Arquitetura de software do Projeto Dragonfly

11.8 Projeto BEAR

O projeto BErkeley AeRobot (BEAR) (Figura 11-32 e Figura 11-33) da Universidade da Califórnia Berkeley (KIM, 2002; KIM, 2003) tem o objetivo de desenvolver múltiplos agentes autônomos integrados e inteligentes com complexidade de cognição e de controle reduzidos, tolerância à falhas, adaptatividade a mudanças em tarefas e ambiente, modularidade, escalabilidade para desenvolver missões complexas eficientemente.

Figura 11-32: Helicóptero do projeto BEAR.

Figura 11-33: Berkeley RUAV (Rotorcraft UAV) em um vôo autônomo com robôs de terra.

Na Tabela 11-9 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-9: Resumo das características do sistema aviônico do projeto BEAR.

Arquitetura de Hardware	Centralizada
Processamento	x86/PC104
Especificação do software	não declara ou não implementa.
Sistema Operacional	QNX Neutrino
Linguagem	C++
Arquitetura de Software	não declara ou não implementa
Verificação dos requisitos de tempo-real	não declara ou não implementa

11.8.1 Hardware

Os componentes embarcados (Figura 11-34) são caracterizados como segue: *flight control computer*, *navigational sensors*, *communication module* e *onboard power system*. O *flight control computer* é realiza as tarefas de navegação, guiamento e controle de vôo.

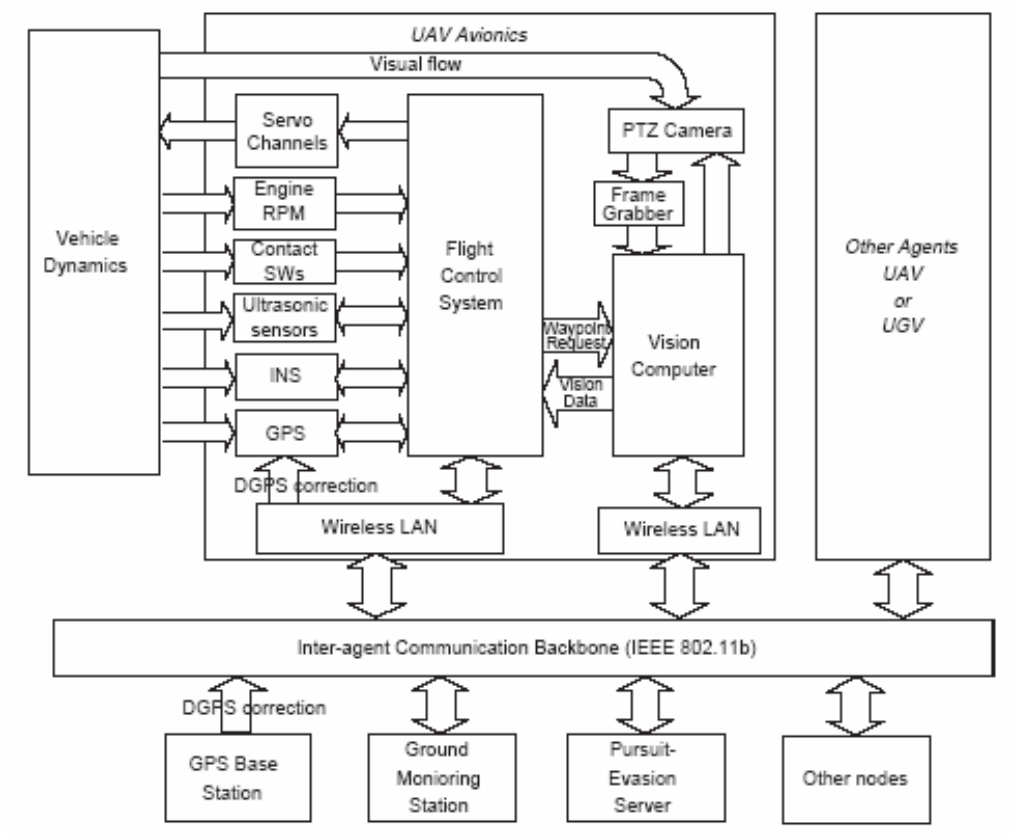

Figura 11-34: Arquitetura de hardware do projeto BEAR.

11.8.2 Software

O software de gerenciamento de vôo foi implementado C no RTOS QNX executando as funções integração com os sensores, piloto automático em tempo real e comunicação entre agentes físicos.

11.9 Projeto Solus

O projeto Solus (Figura 11-35) da universidade de Michigan (ATKINS, 1998) objetiva o desenvolvimento de técnicas de controle de vôo e planejamento de trajetória inteligentes focando em detecção e recuperação de falhas automatizadas.

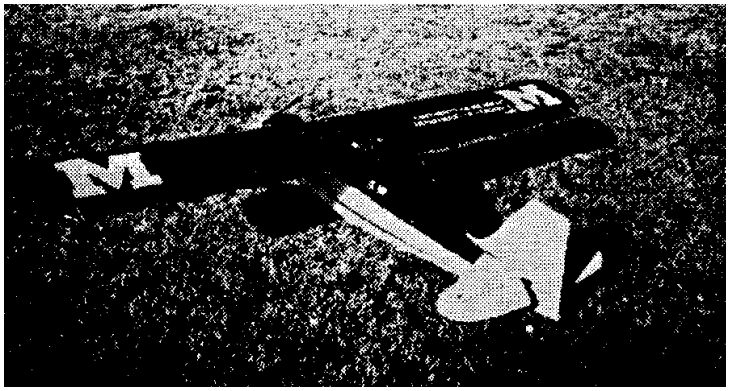

Figura 11-35: VANT do projeto Solus.

Na Tabela 11-14 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-10: Resumo das características do sistema aviônico do projeto Solus.

Arquitetura de Hardware	Centralizado com 2 núcleos de processamento
Processamento	x86/PC104
Especificação do software	não declara ou não implementa.
Sistema Operacional	QNX
Linguagem	C++
Arquitetura de Software	não declara ou não implementa.
Verificação dos requisitos de tempo-real	não declara ou não implementa.

11.9.1 Hardware

A arquitetura de hardware centralizada (Figura 11-36) é composta por uma pilha PC104 composta por uma CPU 486 que realiza a interface com sensores, estimação e controle; uma CPU Pentium que realiza o guiamento e a comunicação com a base e com o DGPS; uma placa de portas I/O para leituras analógicas e digitais; e uma placa customizada (AIB) que lê e interpreta os comandos enviados pelo transmissor RC e gera os sinais de referência para os servos.

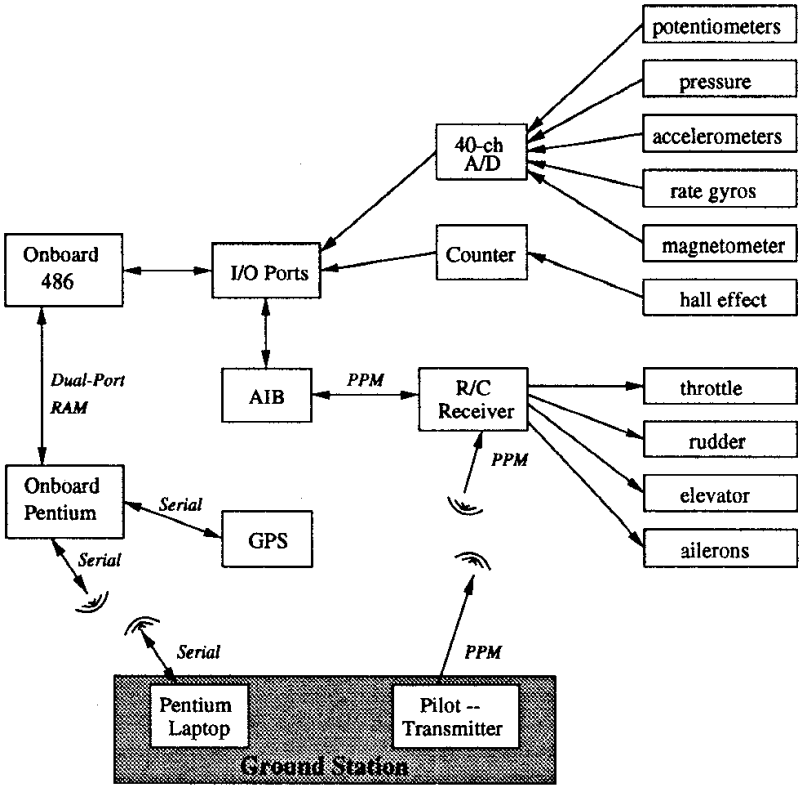

Figura 11-36: Arquitetura de hardware do projeto Solus.

11.9.2 Software

O sistema foi implementado em C++ no RTOS QNX. Não houve preocupações com o projeto de software e nem como identificar uma arquitetura utilizada. Na Figura 11-37 pode-se as tarefas executadas pelos núcleos do computador embarcado.

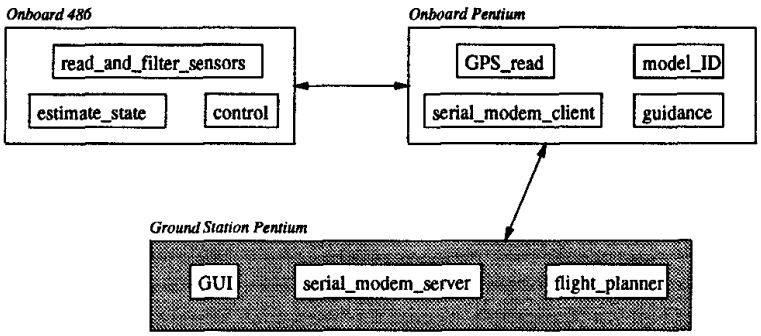

Figura 11-37: Arquitetura de software do projeto Solus.

11.10 Projeto AURYON

O projeto Auryon (Figura 11-38) da Universidade de Tecnologia de Compiègne (VIDOLOV, 2005) foca o desenvolvimento de Micro VANTs para testes de controle para estabilização, pouso e decolagem automáticos, SLAM e anti-colisão.

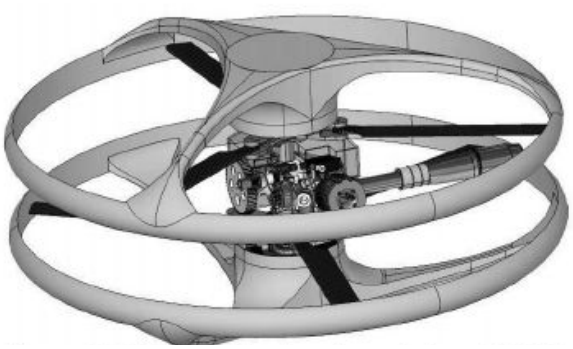

Figura 11-38: Modelo em CAD do protótipo Rigel do projeto Aurion.

Na Tabela 11-11 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-11: Resumo das características do sistema aviônico do projeto Auryon.

Arquitetura de Hardware	Centralizado
Processamento	8 bits Rabbit
Especificação do software	não declara ou não implementa.
Sistema Operacional	uC
Linguagem	C
Arquitetura de Software	não declara ou não implementa
Verificação dos requisitos de tempo-real	não declara ou não implementa

11.10.1 Hardware

A arquitetura de hardware centralizada consiste de um microprocessador de 8bits Rabbit, conforme pode ser observado na Figura 11-39. Os equipamentos embarcados incluem INS, GPS e transmissores *wireless*.

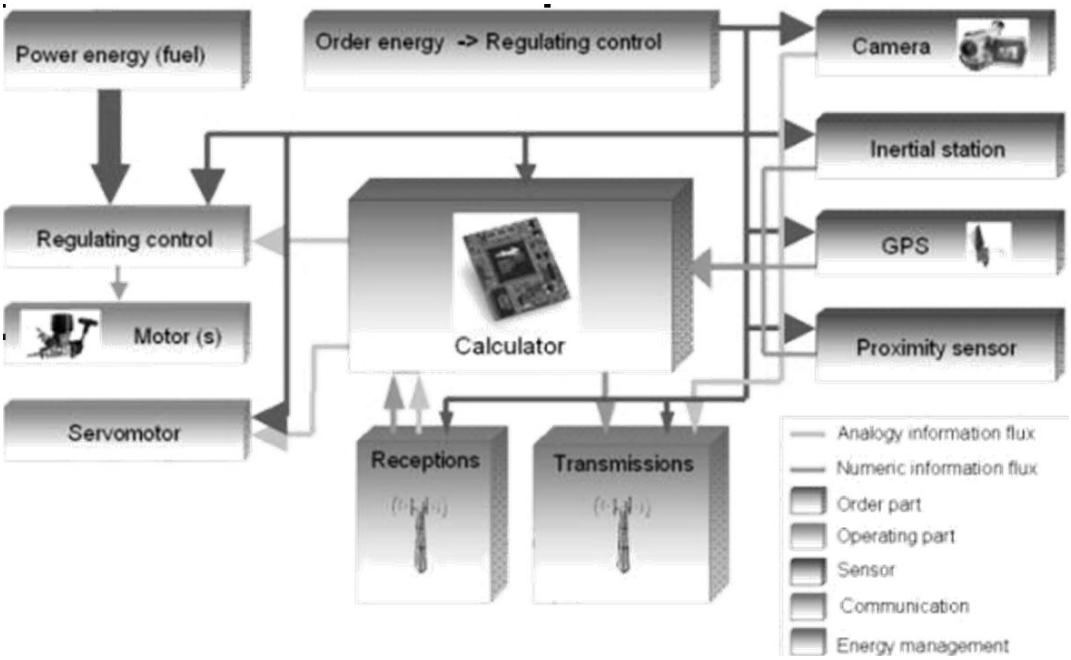

Figura 11-39: Arquitetura de harwdare do projeto Auryon.

11.10.2 Software

O Software foi implementado em C no *kernel* uC da Rabbit.

11.11 Projeto Kiteplane

O projeto Kiteplane (Figura 11-40) da Universidade de Kumamoto (KUMON, 2006) foca o estudo de algoritmos de controle autônomo para aeronaves do tipo *hang glider*. Segundo Kumon (2006) estas aeronaves são altamente eficientes para aplicações de monitoramento vulcânico.

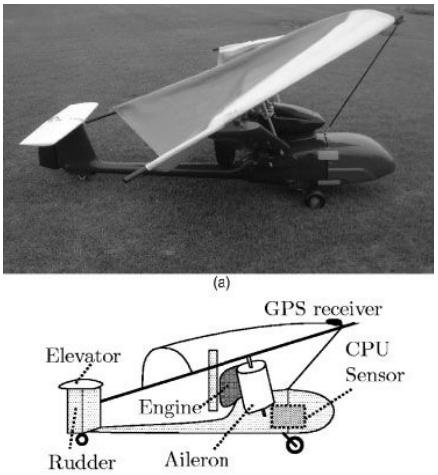

Figura 11-40: Aeronave do projeto Kiteplane.

Na Tabela 11-12 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-12: Resumo das características do sistema aviônico do projeto Kiteplane.

Arquitetura de Hardware	Centralizado
Processamento	x86/PC104
Especificação do software	não declara ou não implementa.
Sistema Operacional	ART-Linux
Linguagem	C++
Arquitetura de Software	não declara ou não implementa.
Verificação dos requisitos de tempo-real	não declara ou não implementa.

11.11.1 Hardware e Software

A arquitetura de hardware centralizada tem como ECU central um PC104 no qual são conectados: uma unidade analógico/digital e uma FPGA que gera os sinais para os servos, conforme pode ser observado na Figura 11-41.

O software foi implementado em ART-Linux, um RTOS baseado em Linux, e o sistema de controle foi implementado como uma *Real Time Task*.

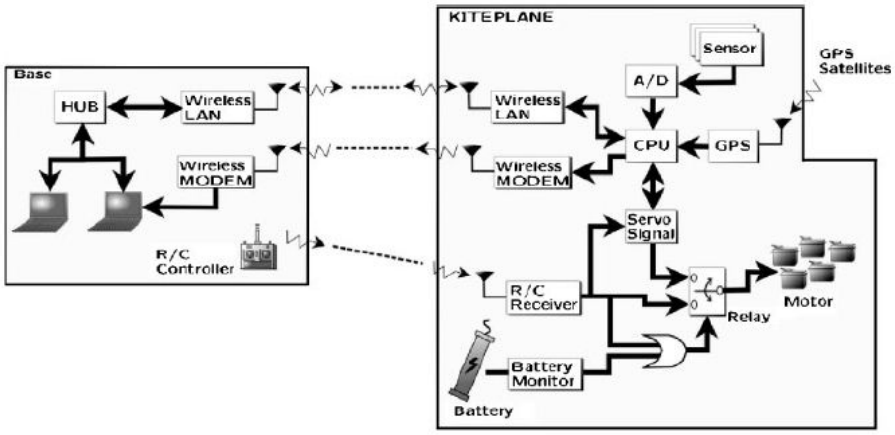

Figura 11-41: Arquitetura de hardware do projeto Kiteplane.

11.12 Projeto Frog

O projeto Frog (Figura 11-42) do *United States Naval Postgraduate School* (HALLBERG, 1999) foca o estudo de novos algoritmos de guiamento e controle bem como identificação da dinâmica.

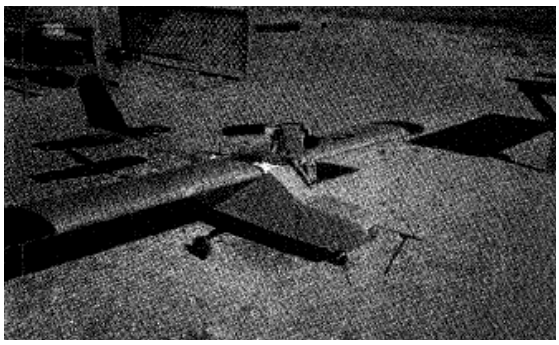

Figura 11-42: VANT do Projeto Frog.

Na Tabela 11-13 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-13: Resumo das características do sistema aviônico do projeto Frog.

Arquitetura de Hardware	Distribuída com processamento dos algoritmos de controle na base.
Processamento	não declara ou não implementa.
Especificação do software	não declara ou não implementa.
Sistema Operacional	não declara ou não implementa.
Linguagem	não declara ou não implementa.
Arquitetura de Software	não declara ou não implementa.
Verificação dos requisitos de tempo-real	não declara ou não implementa.

11.12.1 Hardware e Software

O objetivo do projeto era testar algoritmos em alto nível, assim, priorizou-se a utilização de equipamentos COTS para minimizar o desenvolvimento de hardware e software. Segundo a proposta, não houve desenvolvimento de hardware (Figura 11-43) e o desenvolvimento de software restringiu-se a utilização de ferramentas que interfaceiam com o hardware e geram uma abstração de alto nível de programação.

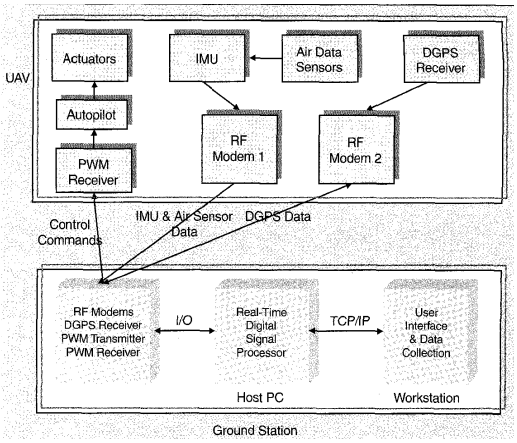

Figura 11-43: Arquitetura de hardware do Projeto Frog.

11.13 Projeto Global Hawk

O projeto Global Hawk (Figura 11-44) do grupo Northrop Grumman Ryan (LOEGERING, 1999) foi desenvolvido para a força aérea norte americana para aplicação de monitoramento de fronteira.

Figura 11-44: Global Hawk.

Na Tabela 11-14 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-14: Resumo das características do sistema aviônico do projeto Global Hawk.

Arquitetura de Hardware	Centralizado com dupla redundância
Processamento	IMMC
Especificação do software	não declara
Sistema Operacional	não declara
Linguagem	não declara
Arquitetura de Software	não declara
Verificação dos requisitos de tempo-real	não declara

11.13.1 Hardware

Para o desenvolvimento do sistema aviônico, foram estabelecidos os seguintes requisitos:

- O VANT deve ser desenvolvido com projeção para futuro crescimento;
- O VANT deve ter não mais que 1 perda em 200 vôos;
- O VANT deve ser capaz de taxiar, decolar e pousar autonomamente, retornar ao hangar autonomamente e também navegar por *waypoints* e aceitar mudanças em qualquer momento na missão.

- A confiabilidade de vôo do VANT deve ser consistente com a operação segura sobre áreas habitadas, operar dentro do espaço aéreo sobre IFR.
- O VANT deve prover comunicação com o controle de espaço aéreo via voz;
- A exatidão do sistema de navegação deve ser inferior a 20m;
- O VANT deve ter um *transponder* capaz de operar no modo 4;

Para atender a todos estes requisitos foram feitas diversas análises principalmente no campo da confiabilidade para atender ao requisito de 1 falha em 200 vôos. Foi verificado que os modos de falhas mais críticas para o sistema aviônico são:

1-Perda da IMU;

2-Perda dos atuadores de aileron, profundor e leme;

3-Perda do computador de controle de vôo;

4-Perda do ADB;

5-Perda do GPS por mais de 30min.

Então foram feitas as análises de confiabilidade que indicariam a necessidade de redundância. Foi verificado que um sistema com dupla redundância para os componentes críticos (IMU, Atuadores de aileron, profundor e leme, Computador de vôo, ADB e GPS) garantiria a confiabilidade requisitada. Na Figura 11-45 pode-se observar a distribuição da aviônica no VANT e na Figura 11-46 pode-se observar a arquitetura do sistema aviônico. Verificou-se também que um sistema em delta aumentaria a muito o custo por um pequeno aumento na confiabilidade.

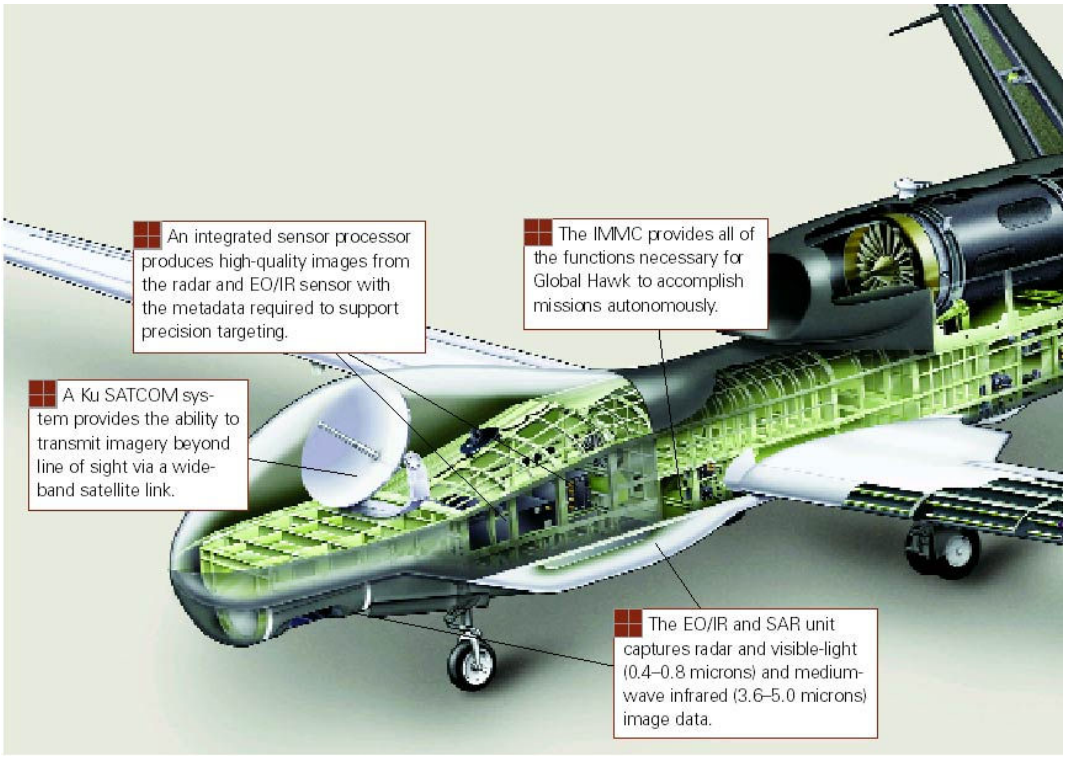

Figura 11-45: Sistema aviônico embarcado no Global Hawk.

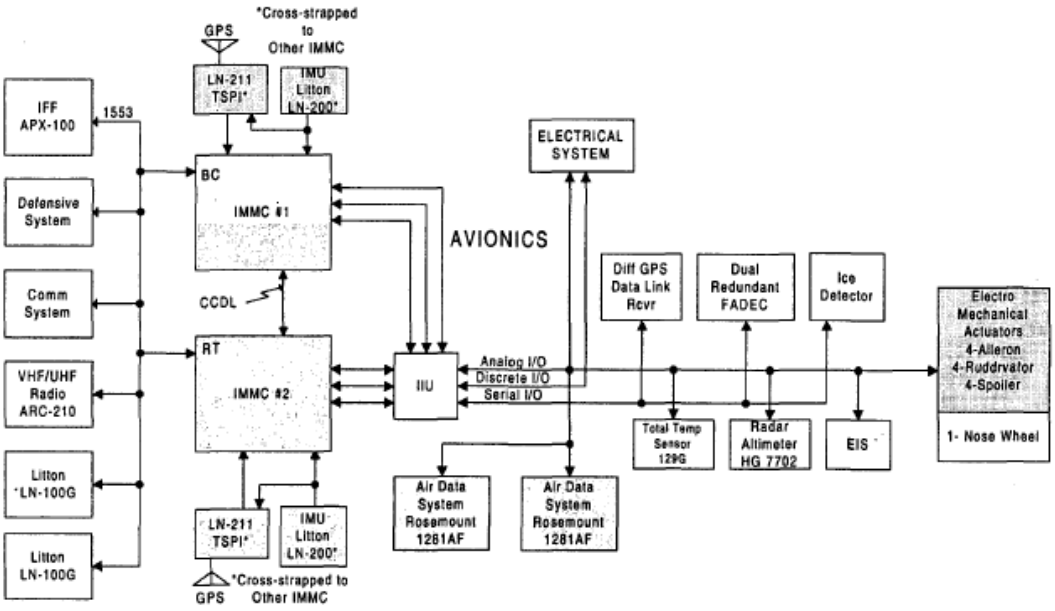

Figura 11-46: Sistema aviônico embarcado no Global Hawk.

11.14 Projeto SeaSCAN

O projeto SeaSCAN (Figura 11-47) da Insitu Group (CAMPBELL, 2004) tem o objetivo de estudar controle semi-autônomo cooperativo, ou seja, operações onde um ou vários

operadores controlam um ou vários veículos não tripulados. Na Figura 11-48 pode-se observar uma proposta de arquitetura de controle cooperativo hierárquico.

Figura 11-47: Aeronave do projeto SeaSCAN.

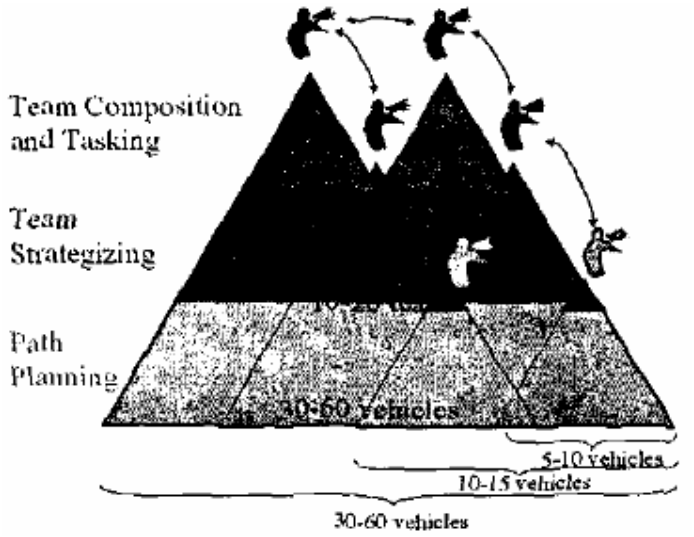

Figura 11-48: Arquietura de controle semi-autônomo hierárquico de sistemas complexos com vários veículos não tripulados.

Na Tabela 11-15 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-15: Resumo das características do sistema aviônico do projeto SeaSCAN.

Arquitetura de Hardware	Centralizado - SE555
Processamento	não declara
Especificação do software	não declara ou não implementa.
Sistema Operacional	não declara
Linguagem	não declara
Arquitetura de Software	não declara ou não implementa.
Verificação dos requisitos de tempo-real	não declara ou não implementa.

11.14.1 Hardware

O hardware consiste de um módulo (SE555) especialmente desenvolvido para o SeaScan (Figura 11-49) que provê gerenciamento de dados e o laço de controle interno (atitude e velocidade).

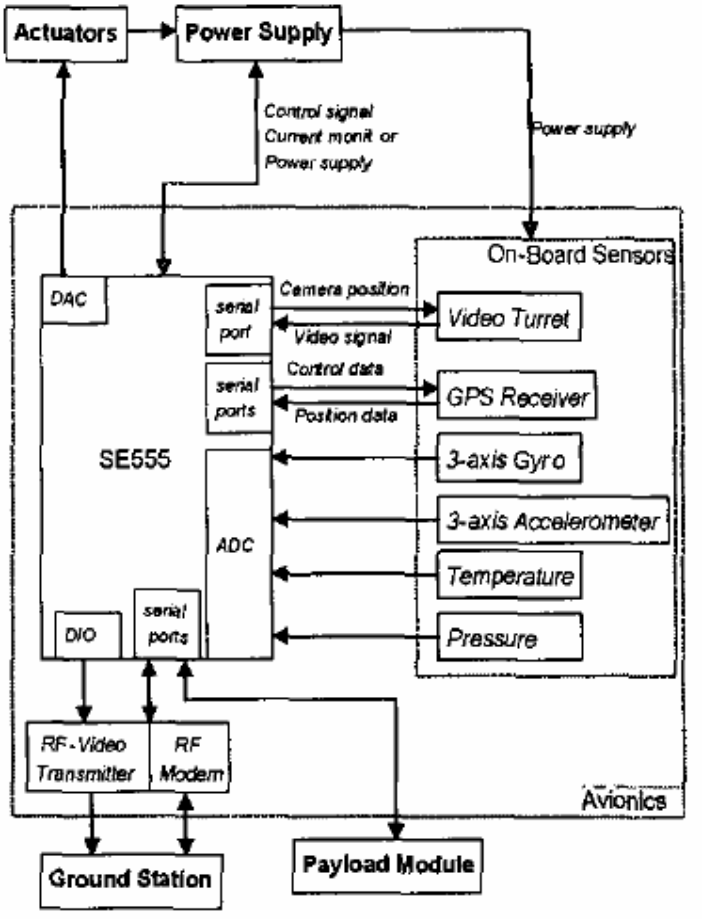

Figura 11-49: Arquietura de hardware do projeto SeaSCAN.

11.15 Projeto ReSSAC

O projeto ReSSAC (Figura 11-50) do departamento de dinâmica e controle de vôo do centro de pesquisa ONERA-Tolousse (FABIANI, 2006) foca o estudo de algoritmos de decisão autônoma e processamento de informação para missões de busca e salvamento.

Figura 11-50: RMax do Projeto ReSSAC.

Na Tabela 11-16 pode-se observar um resumo das características do sistema aviônico.

Tabela 11-16: Resumo das características do sistema aviônico do projeto ReSSAC.

Arquitetura de Hardware	2 ECU Centrais
Processamento	x86/PC104
Especificação do software	não declara ou não implementa.
Sistema Operacional	VXWorks
Linguagem	C++
Arquitetura de Software	não declara ou não implementa.
Verificação dos requisitos de tempo-real	não declara ou não implementa.

11.15.1 Hardware

A arquitetura de hardware embarcado é baseada em duas unidades centrais PC104 ligadas por uma memória compartilhada.

12 ANEXO - ESTRUTURA DE DADOS

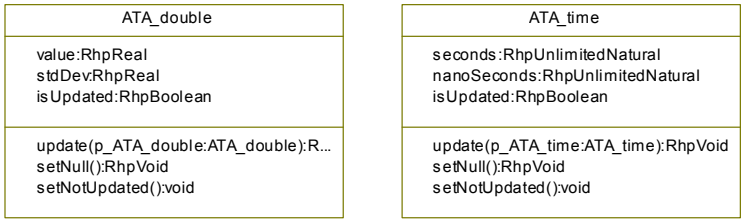

Figura 12-1: UML Object Model - Unidade básica de dados.

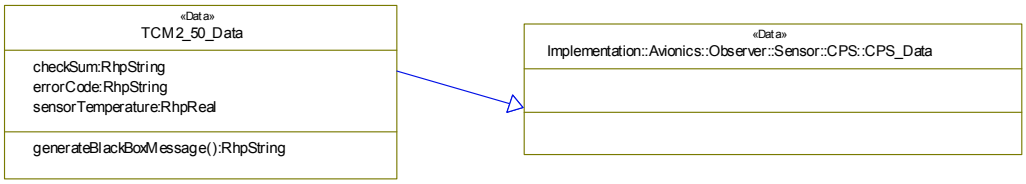

Figura 12-2: UML Object Model - Objetos de dados do sensor device TCM2-50.

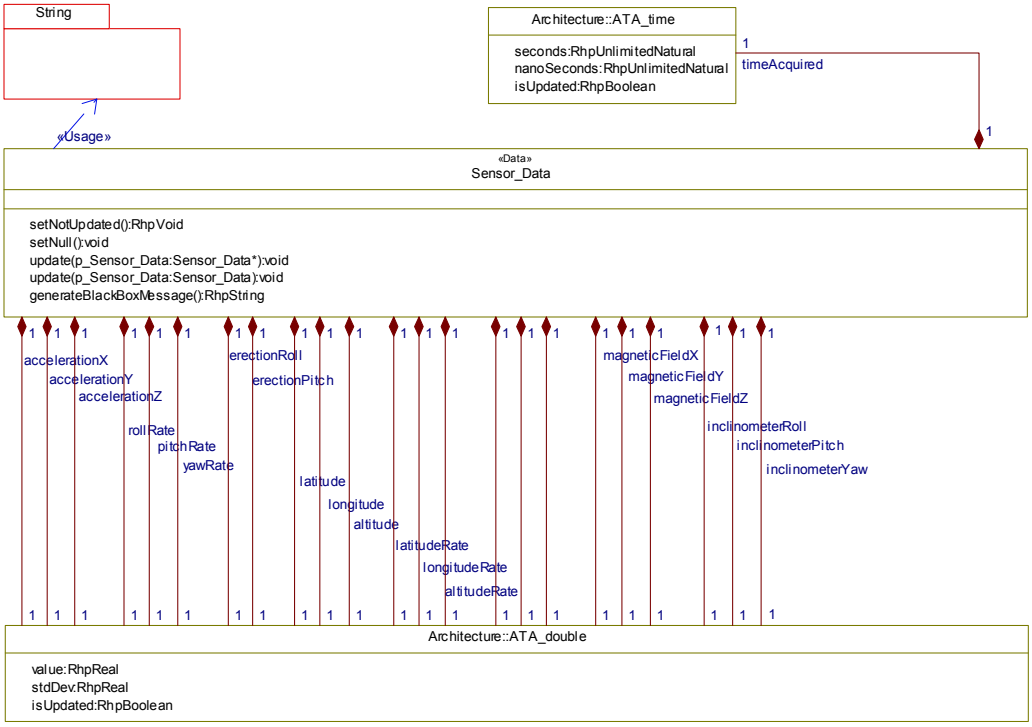

Figura 12-3: UML Object Model - Estrutura de dados de sensor.

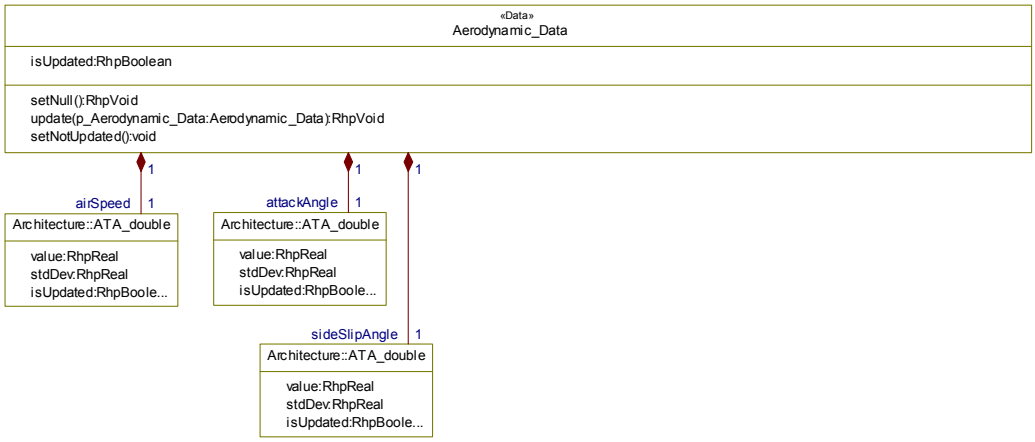

Figura 12-4: UML Object Model - Estrutura de dados Aerodinâmicos.

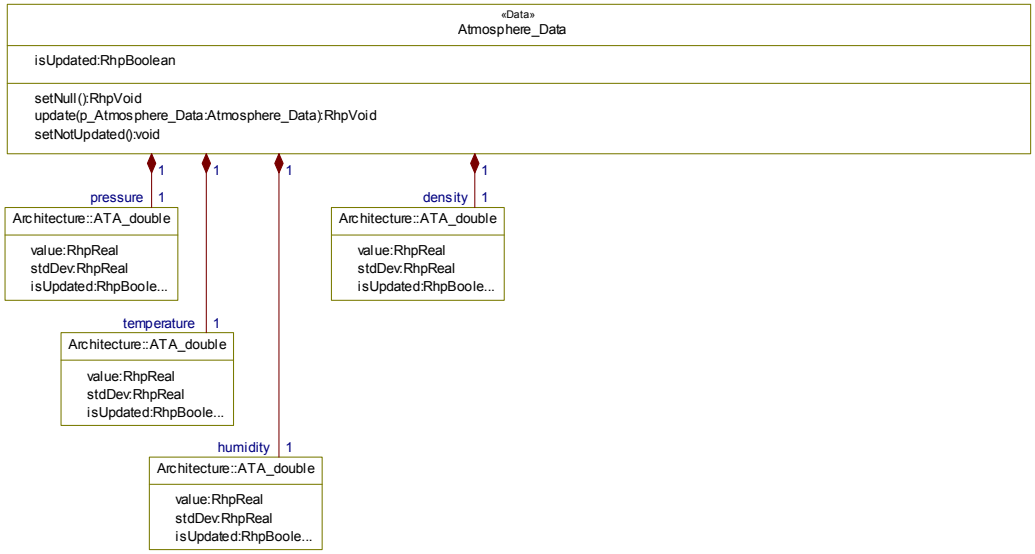

Figura 12-5: UML Object Model - Estrutura de dados da Atmosfera.

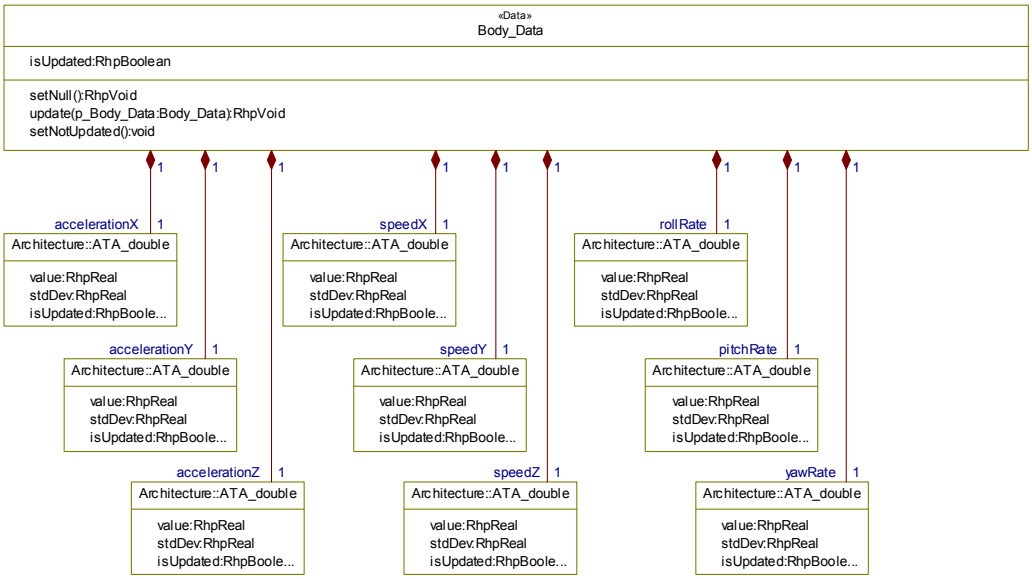

Figura 12-6: UML Object Model - Estrutura de dados do corpo.

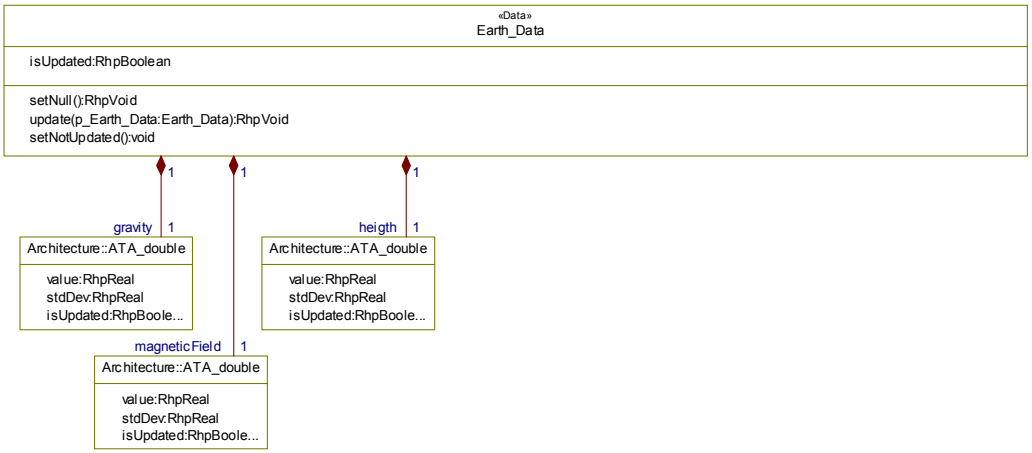

Figura 12-7: UML Object Model - Estrutura de dados da Terra.

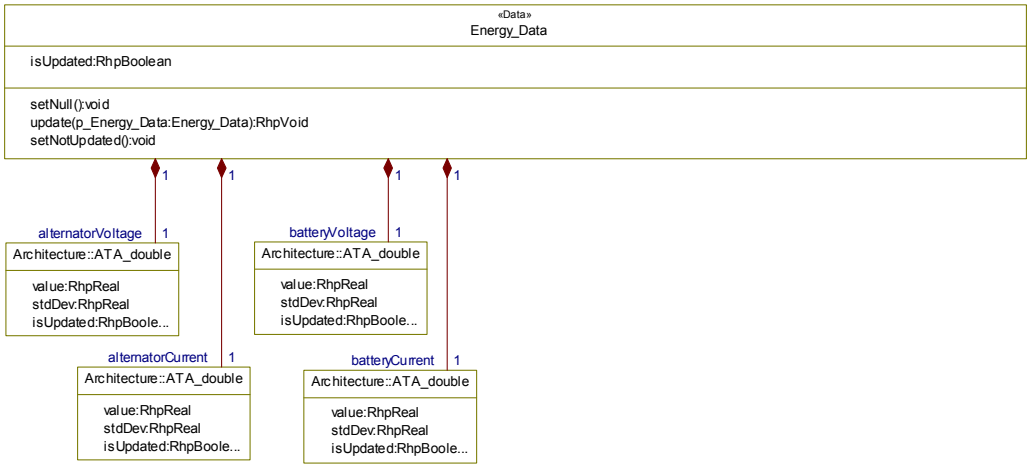

Figura 12-8: UML Object Model - Estrutura de dados do Sistema de Energia.

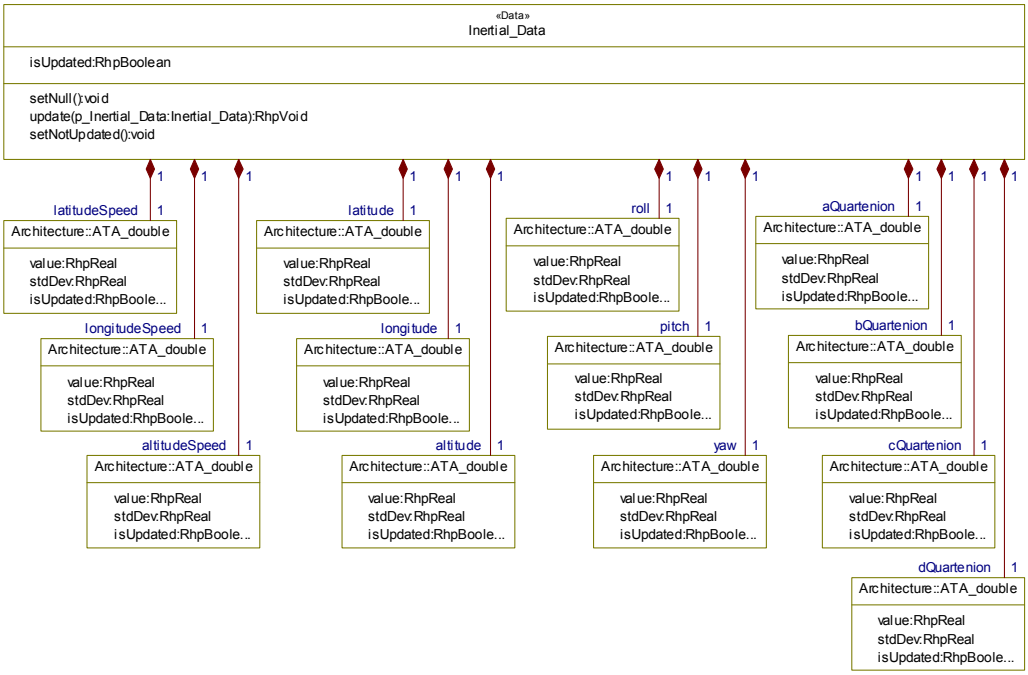

Figura 12-9: UML Object Model - Estrutura de dados Inerciais.

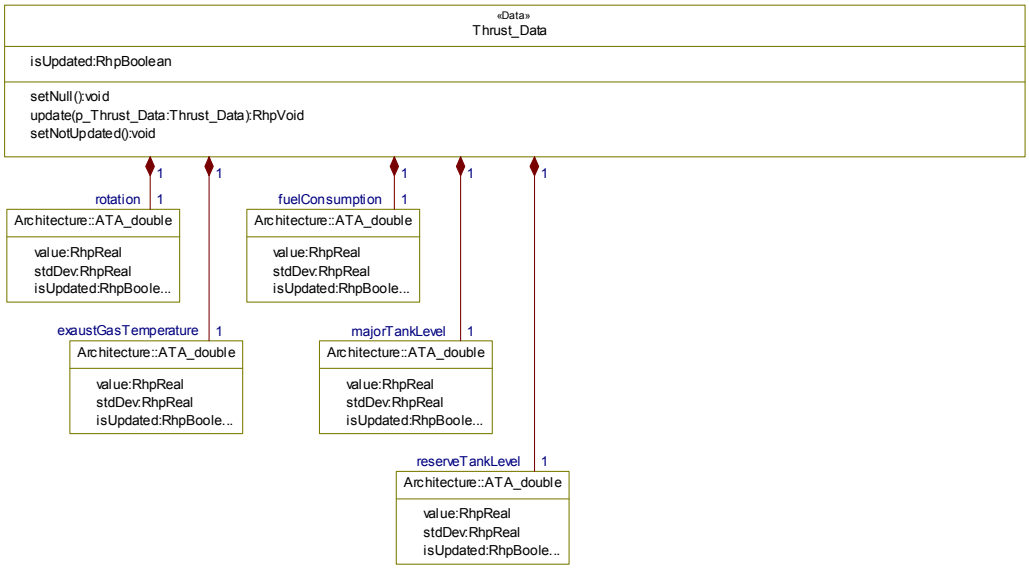

Figura 12-10: UML Object Model - Estrutura de dados do Sistema de Propulsão.

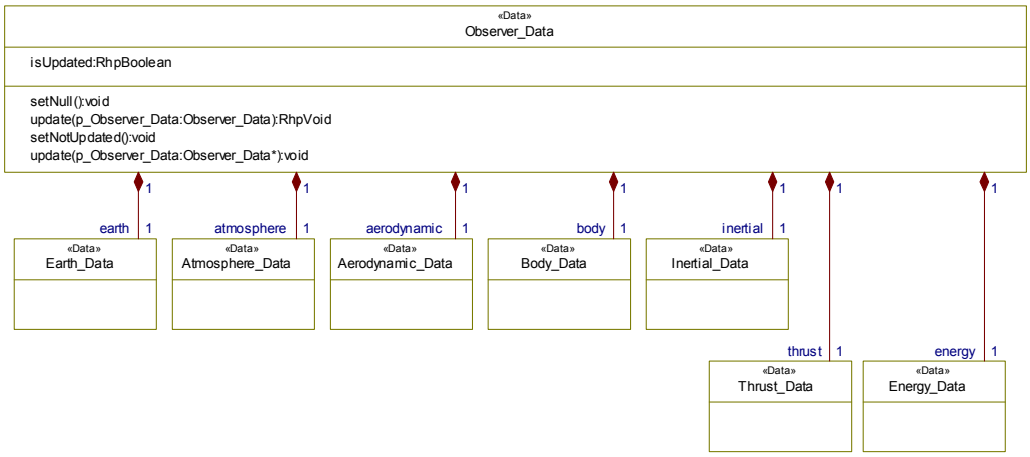

Figura 12-11: UML Object Model - Estrutura de dados do Sistema de Observação.

13 ANEXO - IMPLEMENTAÇÃO

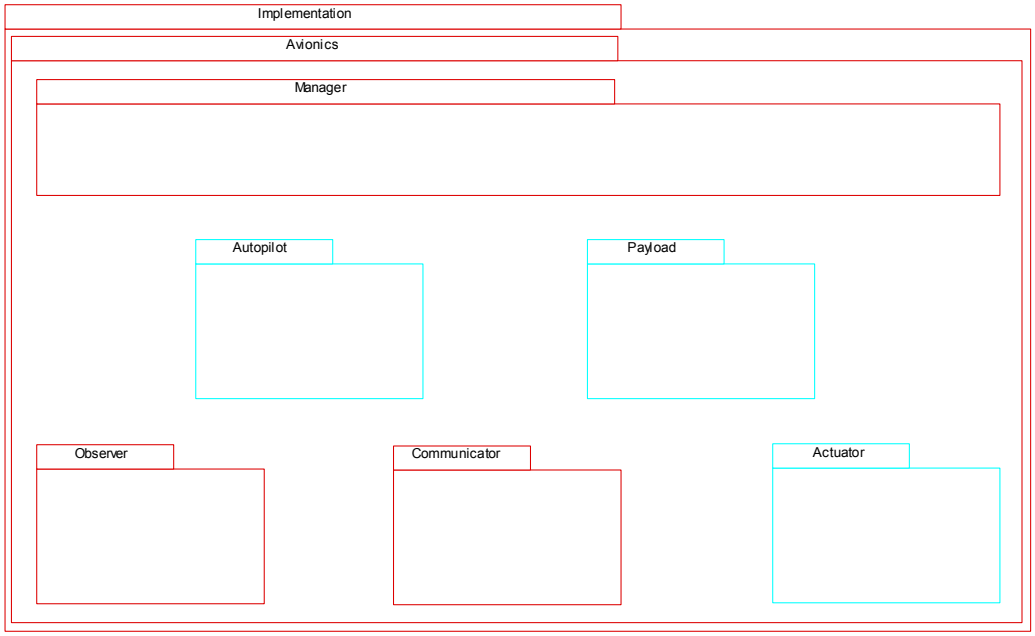

Figura 13-1: UML Object Model - Organização da implementação.

13.1 Package Manager

Figura 13-2: UML Object Model – Pacote Manager

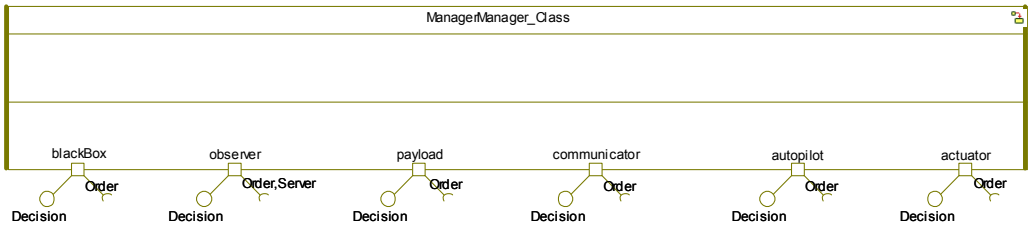

Figura 13-3: UML Object Model - Implementação da classe do agente Manager.

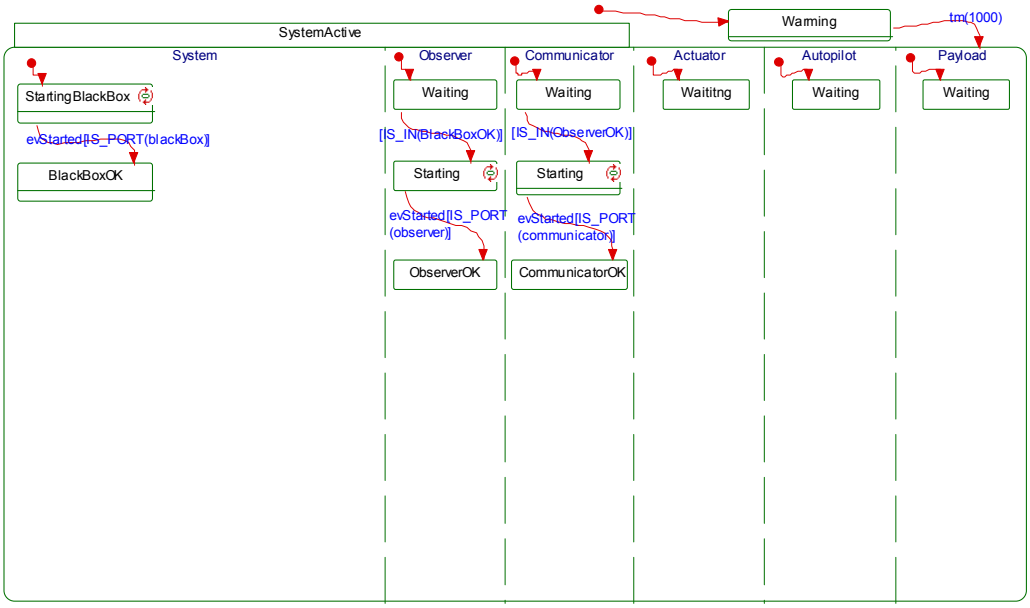

Figura 13-4: UML Statechart - Comportamento do agente Manager.

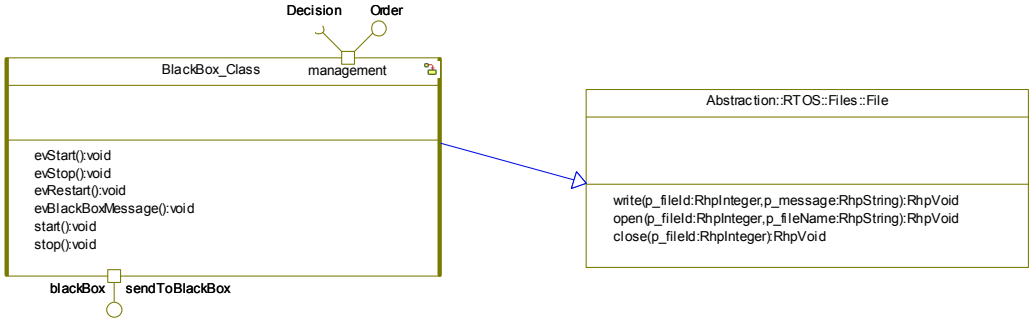

Figura 13-5: UML Object Model - Implementação da classe do agente BlackBox.

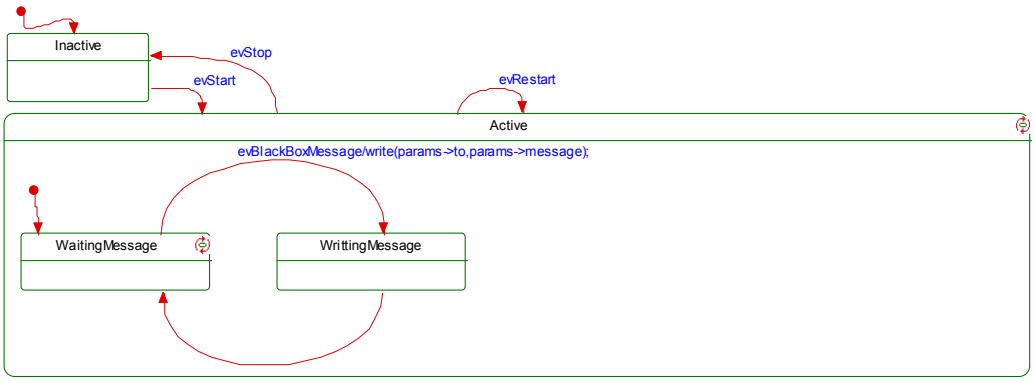

Figura 13-6: UML Statechart - Comportamento do agente Black Box.

13.2 Package Observer

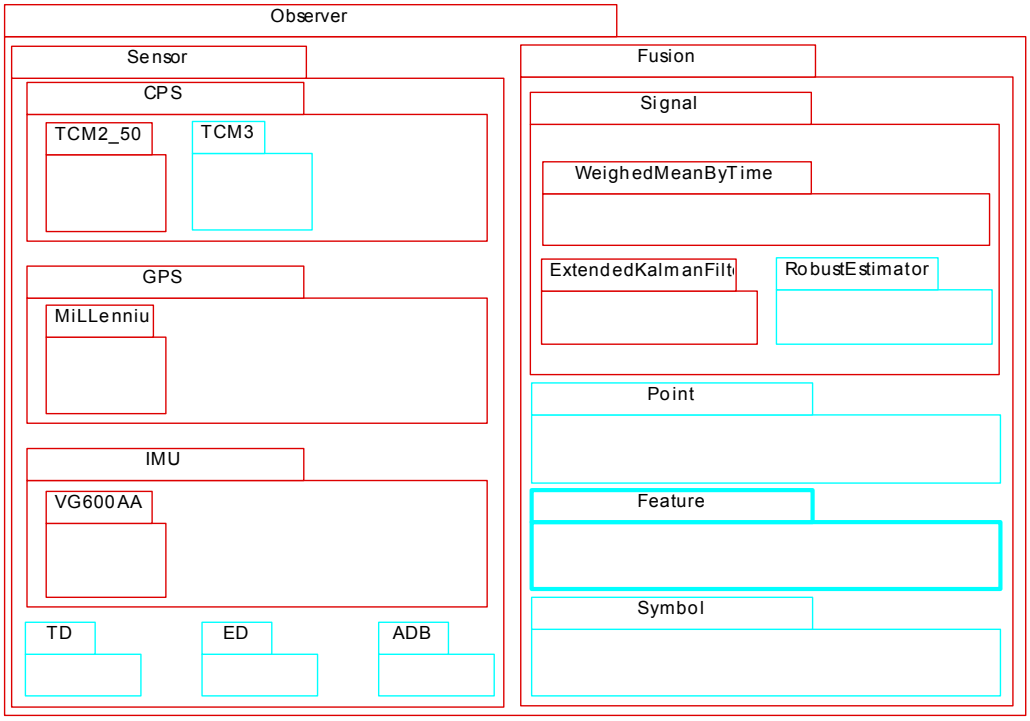

Figura 13-7: UML Object Model - Pacote Observer.

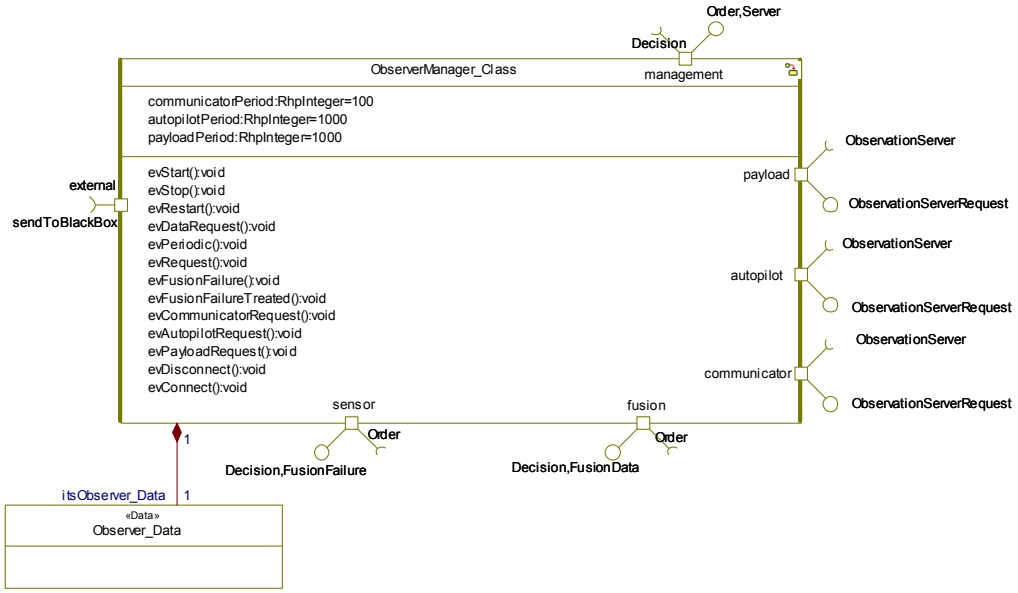

Figura 13-8: UML Object Model - Implementação do agente Observer.

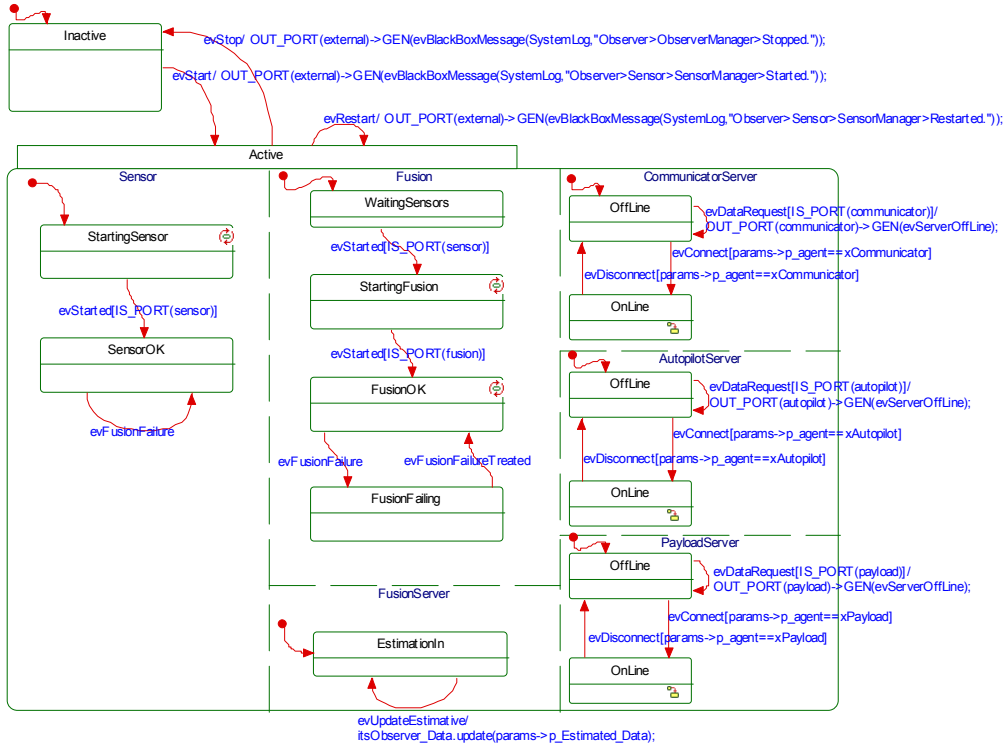

Figura 13-9: UML Statechart - Comportamento do agente Observer.

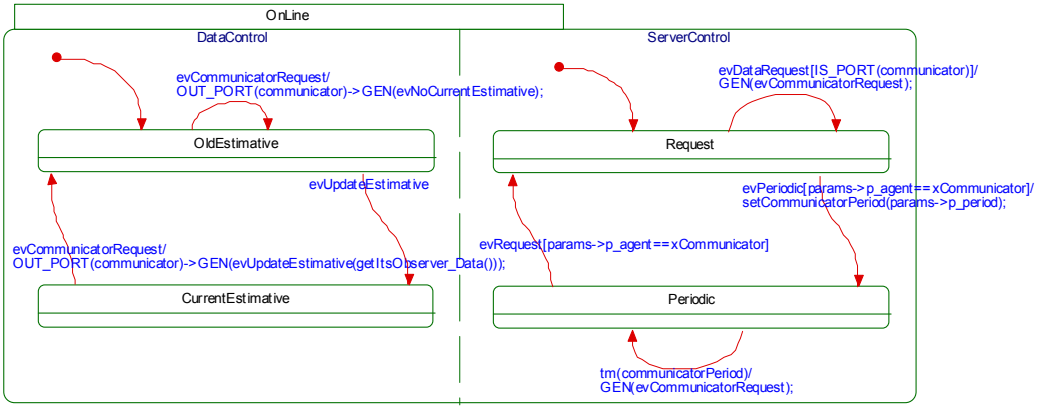

Figura 13-10: UML Statechart - Comportamento do estado ONLINE.

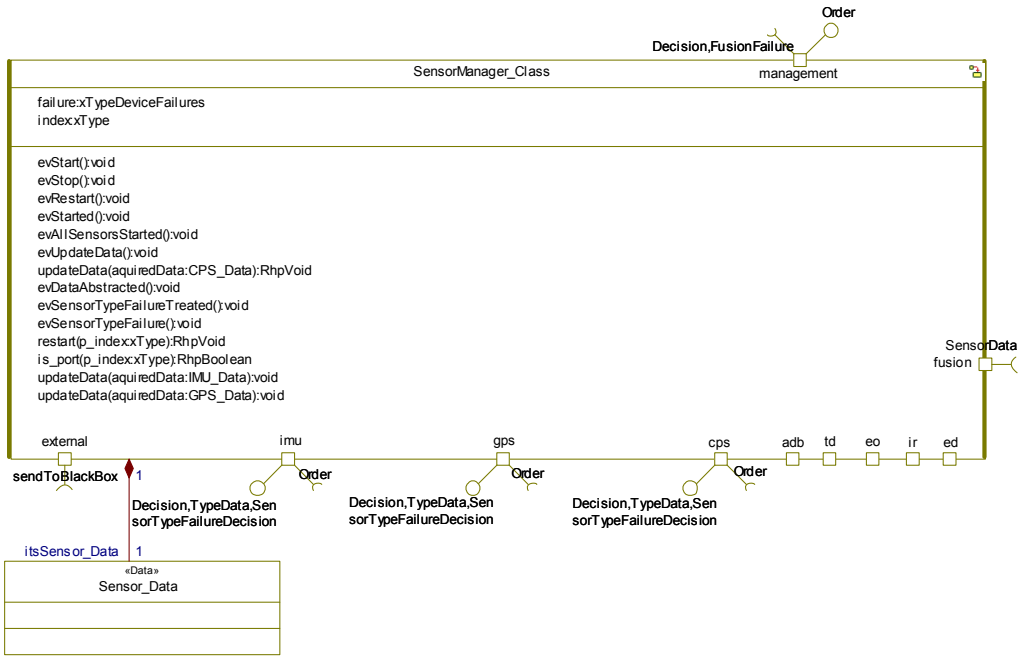

Figura 13-11: UML Object Model - Implementação do agente Sensor.

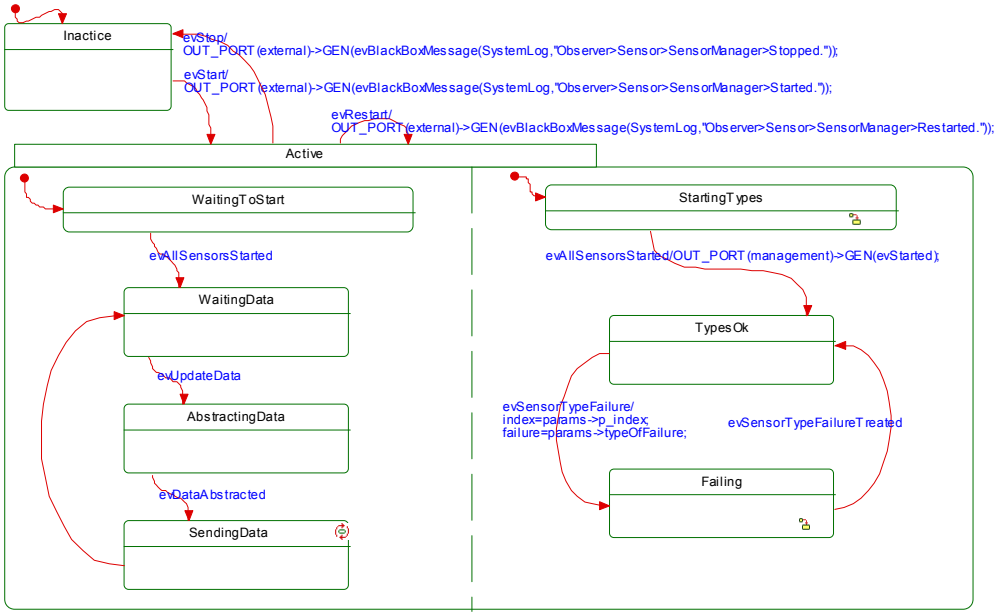

Figura 13-12: UML Statechart - Comportamento do agente Sensor.

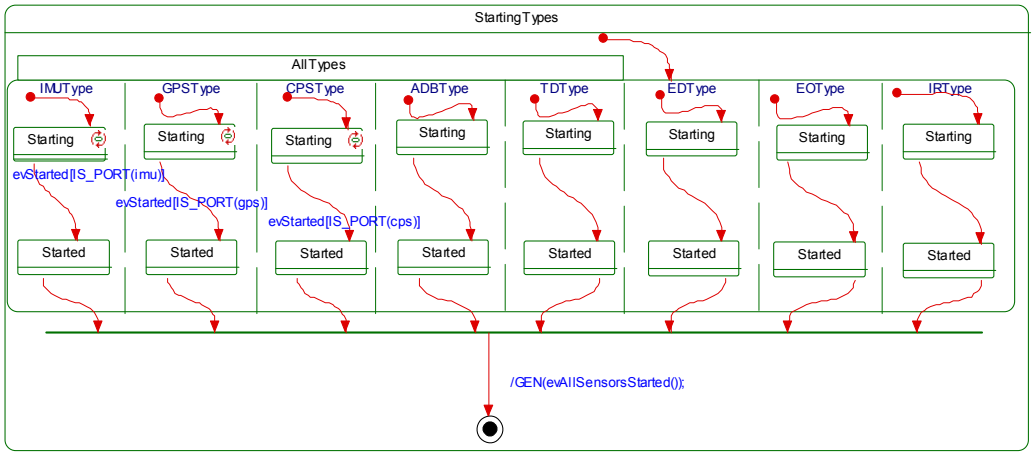

Figura 13-13: UML Statechart - Comportamento do estado Starting Types do agente Sensor.

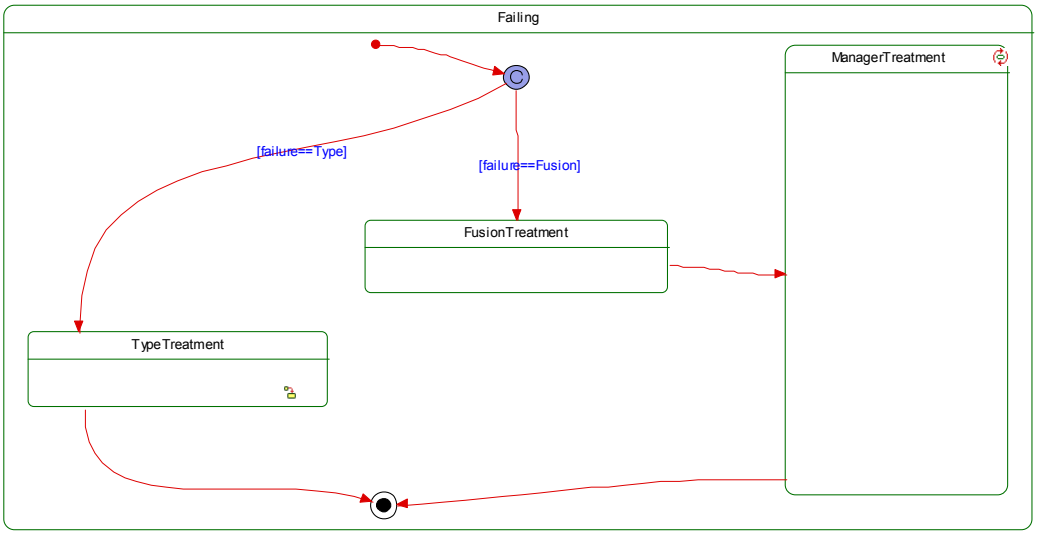

Figura 13-14: UML Statechart - Comportamento do estado Failing do agente Sensor.

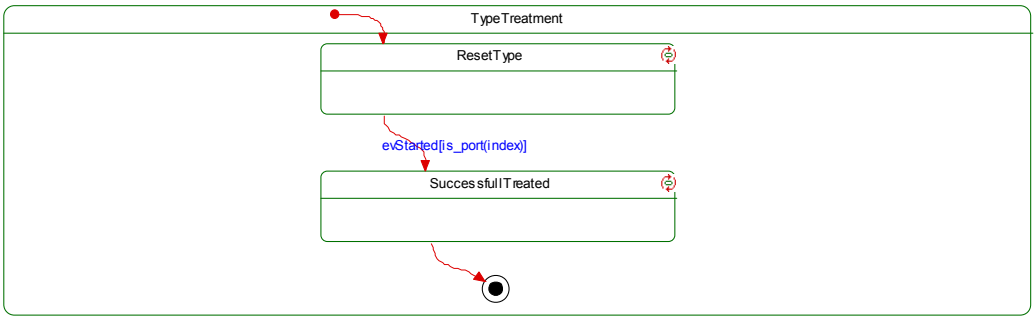

Figura 13-15: UML Statechart - Comportamento do estado Type Treatment do agente Sensor.

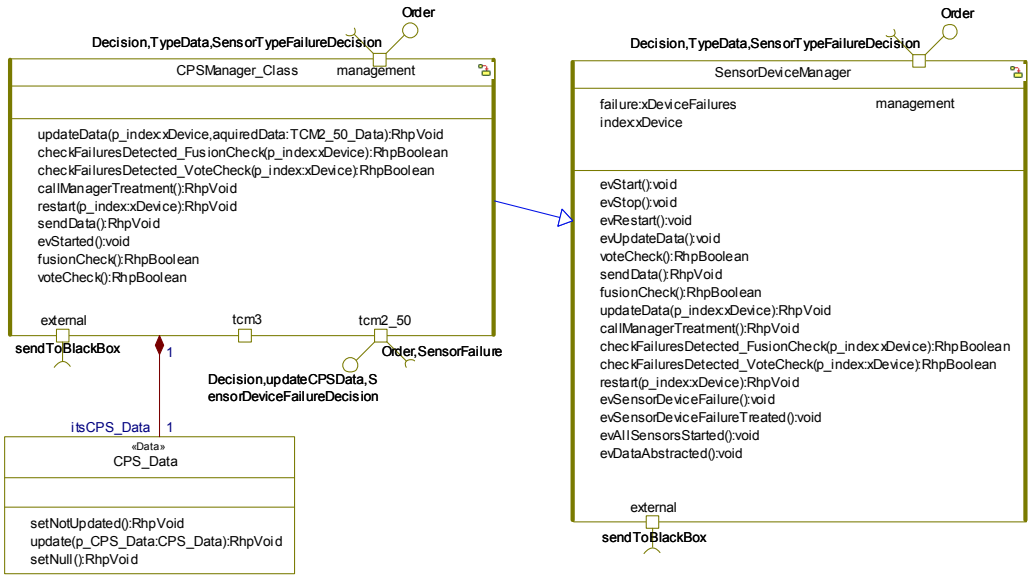

Figura 13-16: UML Object Model - Implementação do agente CPS.

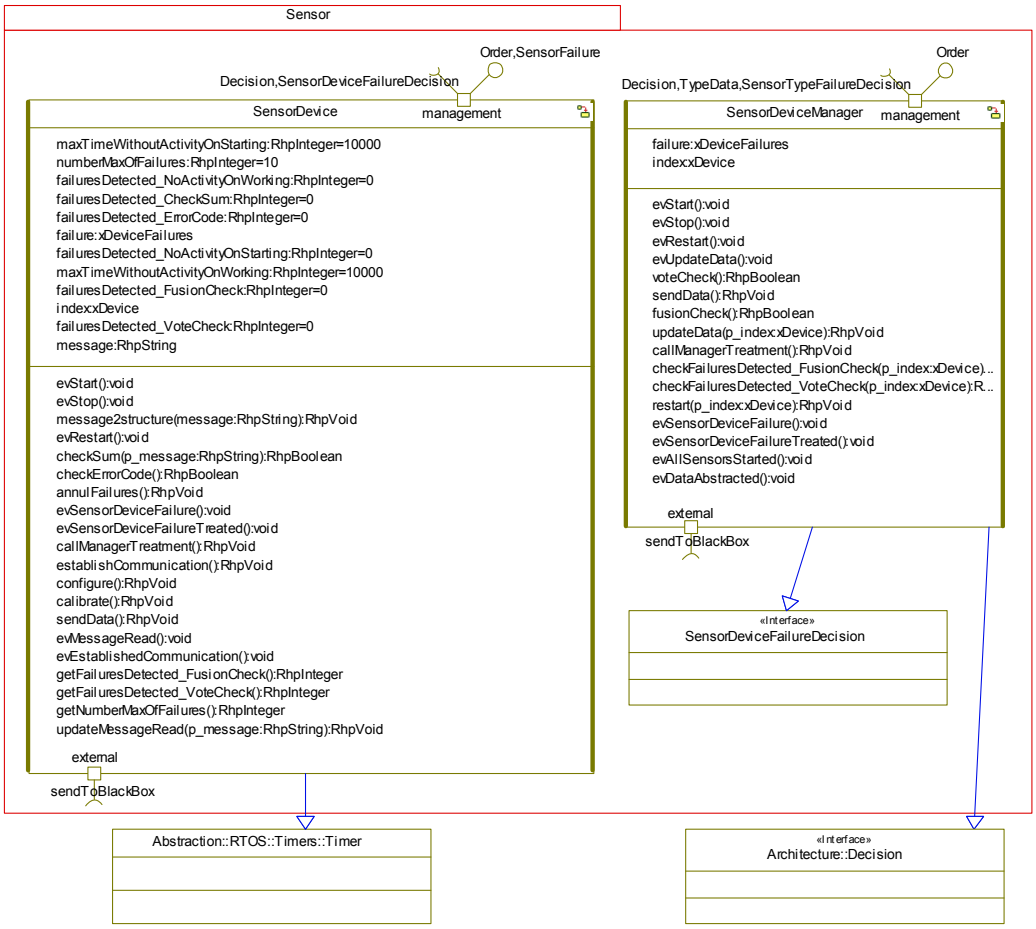

Figura 13-17: UML Object Model - Classe pai para os nível de dispositivo e categoria do agente Sensor.

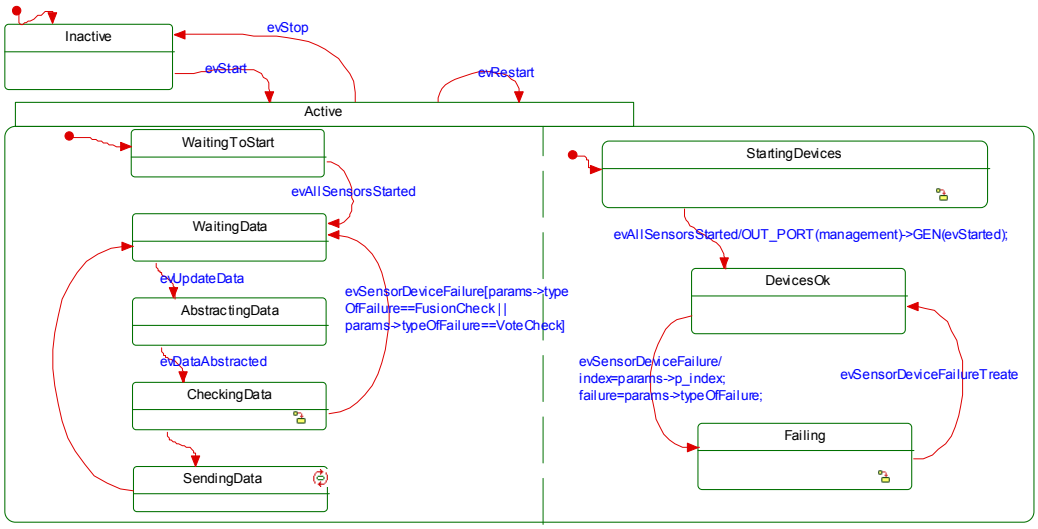

Figura 13-18: UML Statechart - Comportamento de SensorDeviceManager.

Figura 13-19: UML Statechart - Comportamento do estado Starting de SensorDeviceManager.

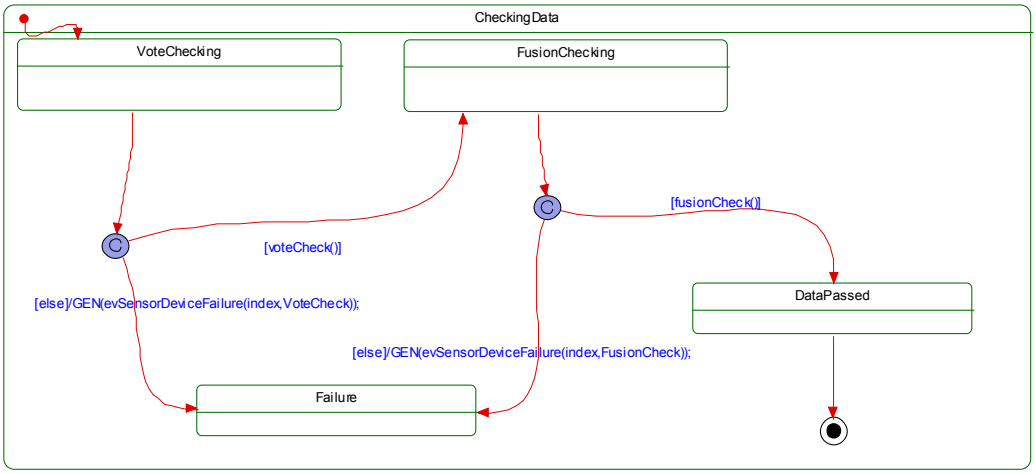

Figura 13-20: UML Statechart - Comportamento do estado CheckingData de SensorDeviceManager.

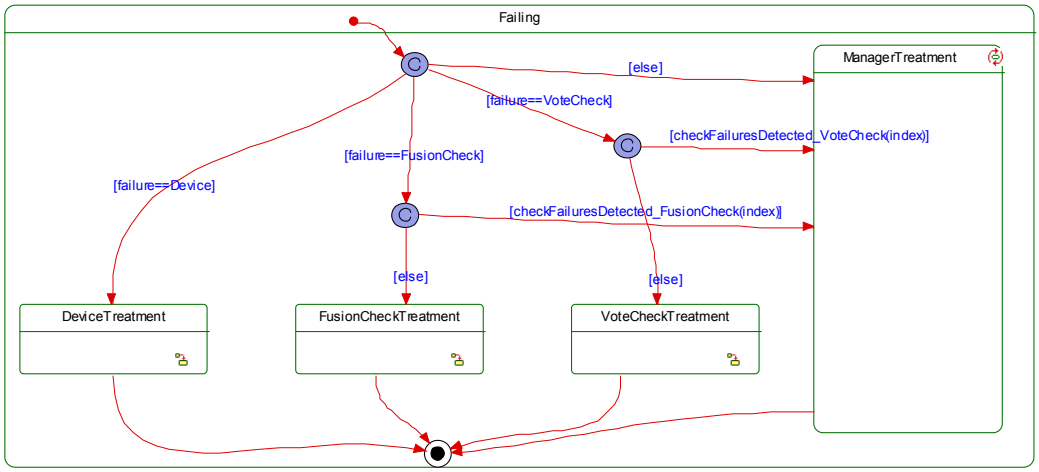

Figura 13-21: UML Statechart - Comportamento do estado Failing de SensorDeviceManager.

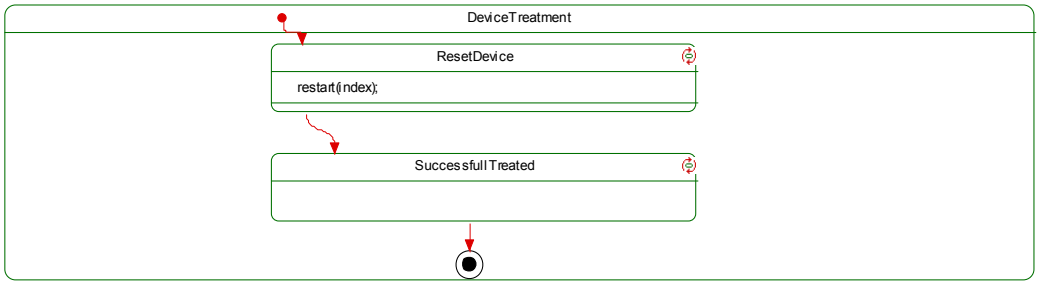

Figura 13-22: UML Statechart - Comportamento do estado DeviceTreatment de SensorDeviceManager.

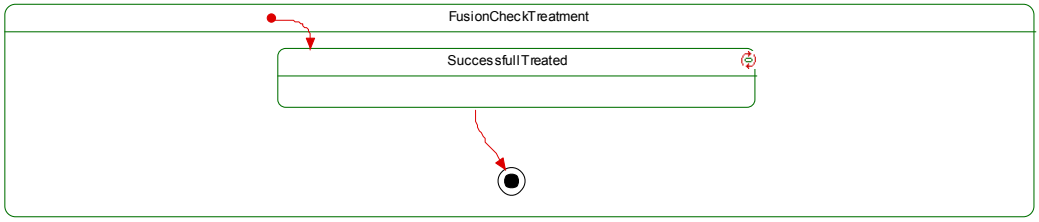

Figura 13-23: UML Statechart - Comportamento do estado Fusion Check Treatment de SensorDeviceManager.

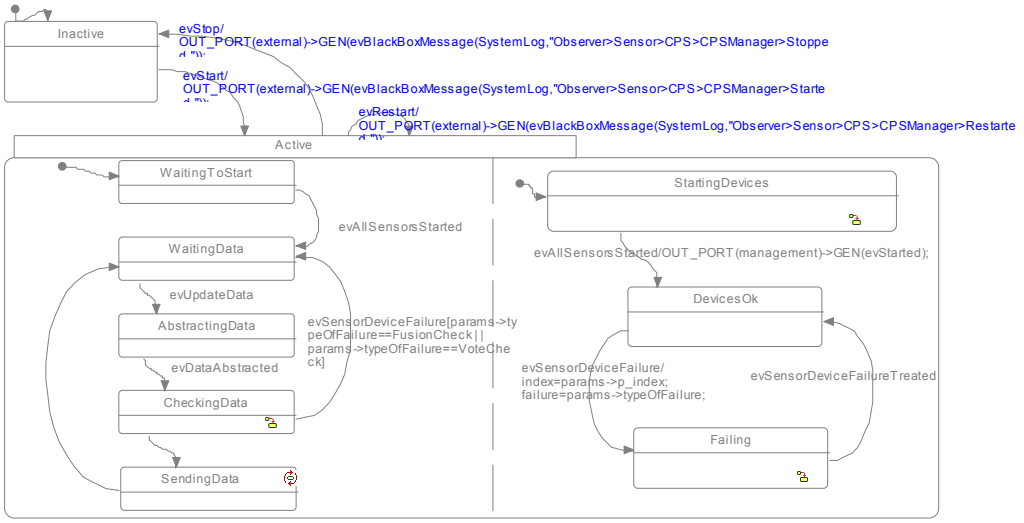

Figura 13-24: UML Statechart - Comportamento do agente CPS.

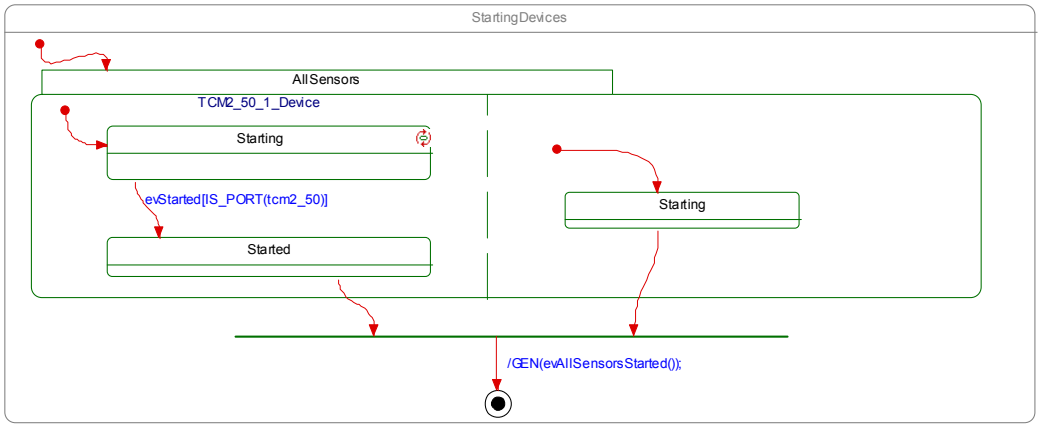

Figura 13-25: UML Statechart - Comportamento do estado StartingDevices do agente CPS.

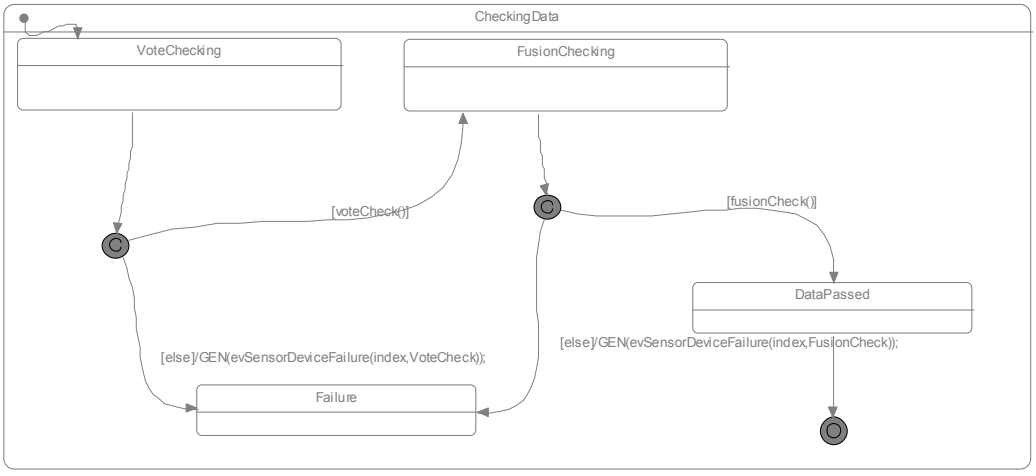

Figura 13-26: UML Statechart - Comportamento do estado CheckingData do agente CPS.

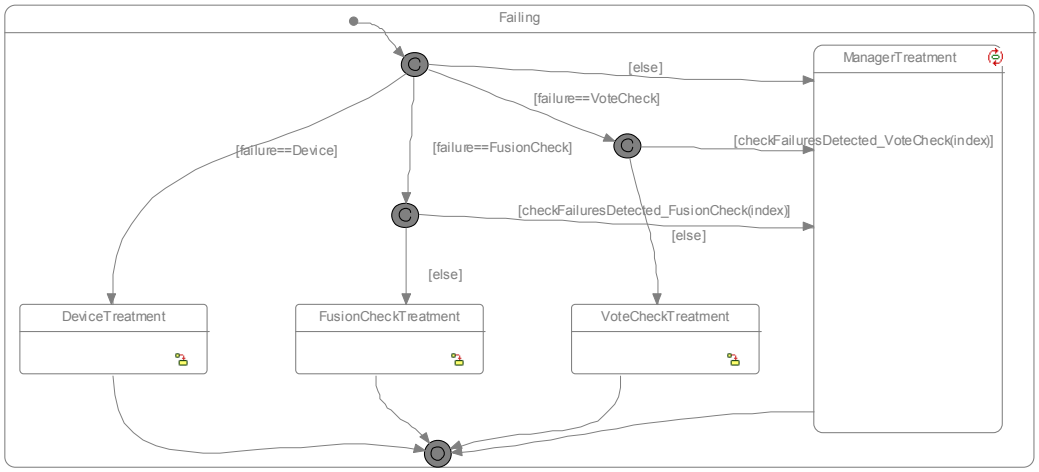

Figura 13-27: UML Statechart - Comportamento do estado Failing do agente CPS.

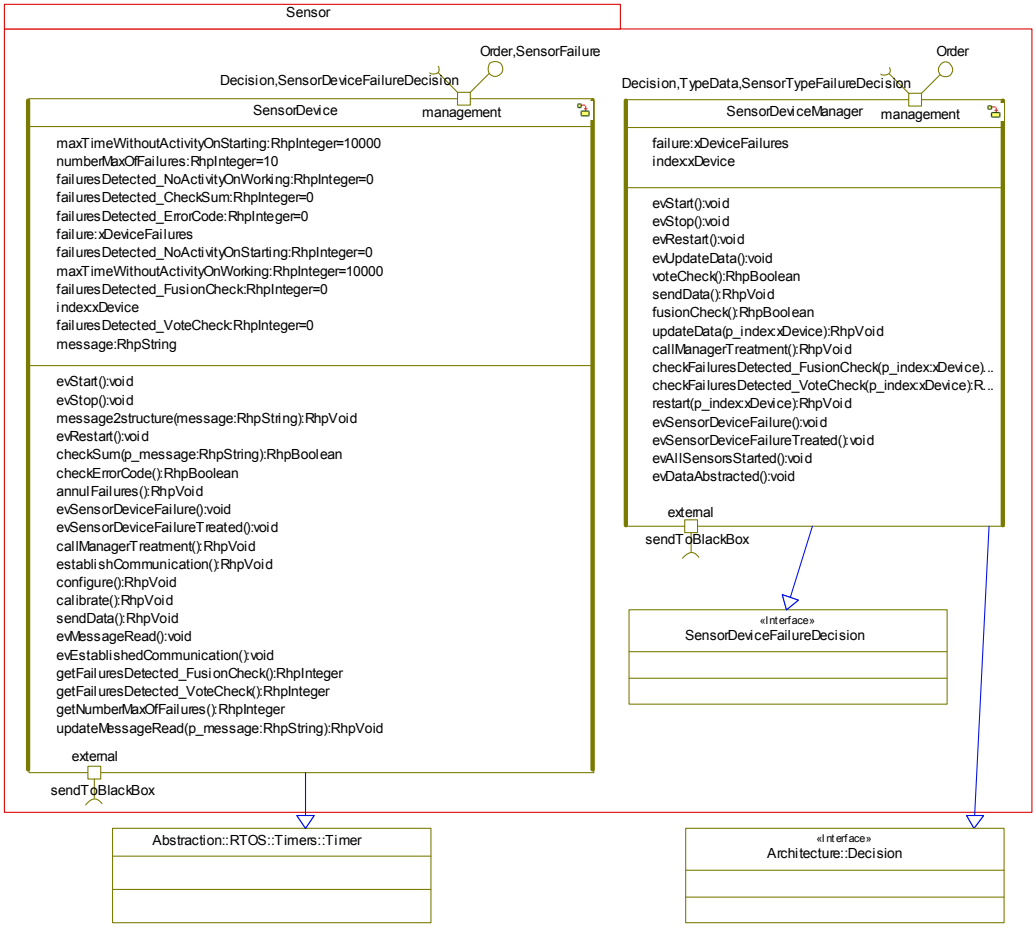

Figura 13-28: UML Object Model - Classe pai para os nível de Dispositivo e Categoria do agente Sensor.

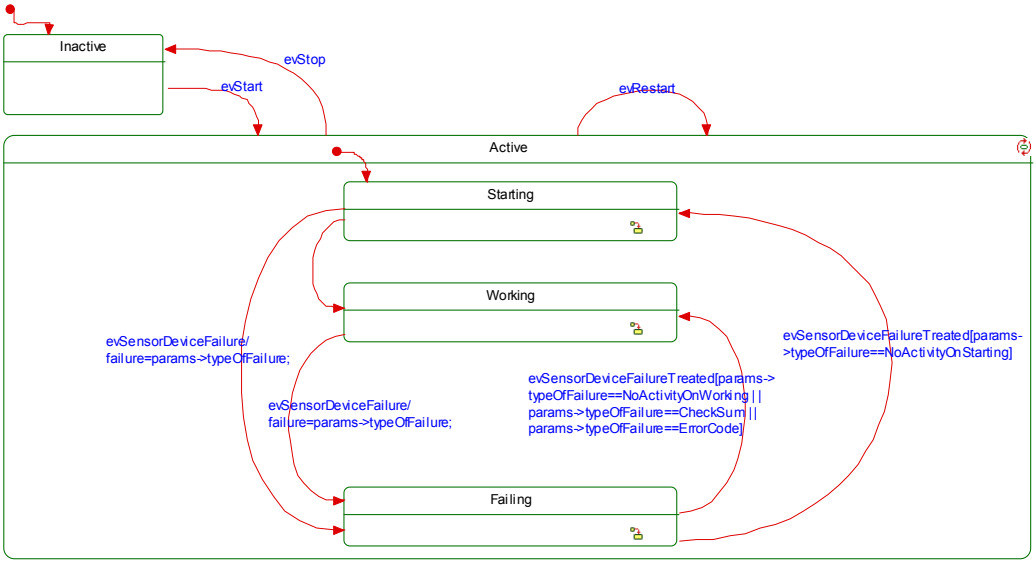

Figura 13-29: UML Statechart - Comportamento de SensorDevice.

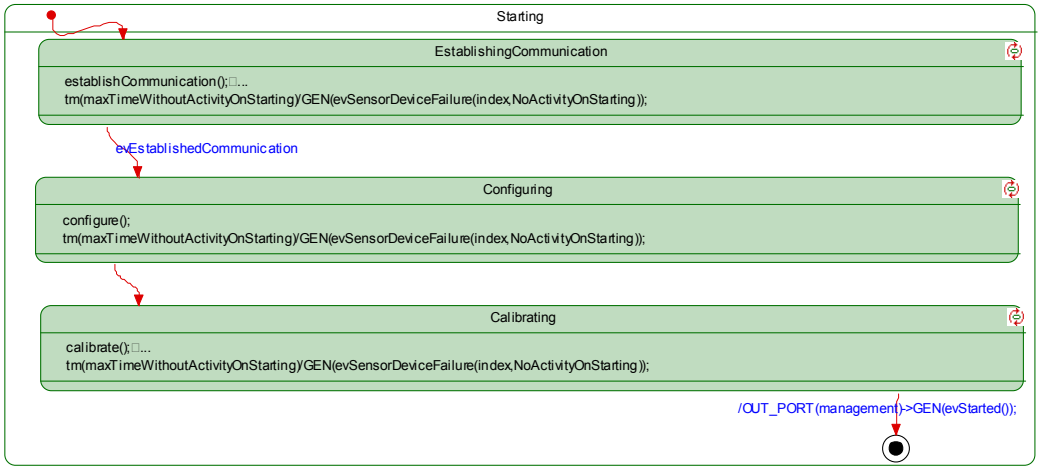

Figura 13-30: UML Statechart - Comportamento do estado Starting de SensorDevice.

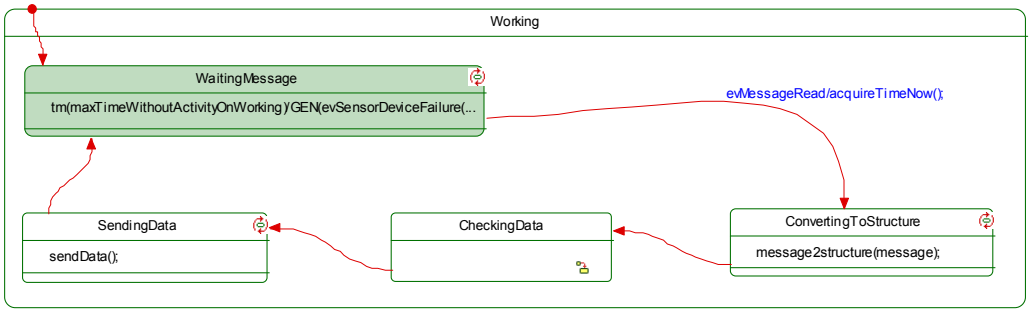

Figura 13-31: UML Statechart - Comportamento do estado Working de SensorDevice.

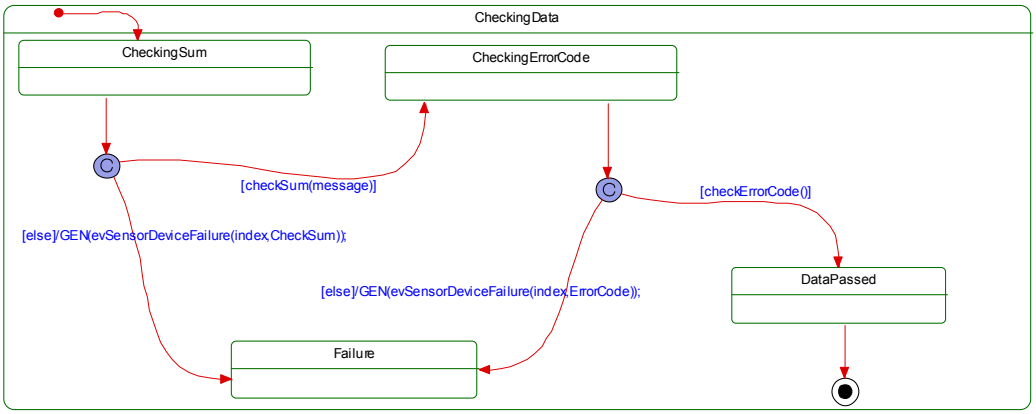

Figura 13-32: UML Statechart - Comportamento do estado CheckingData de SensorDevice.

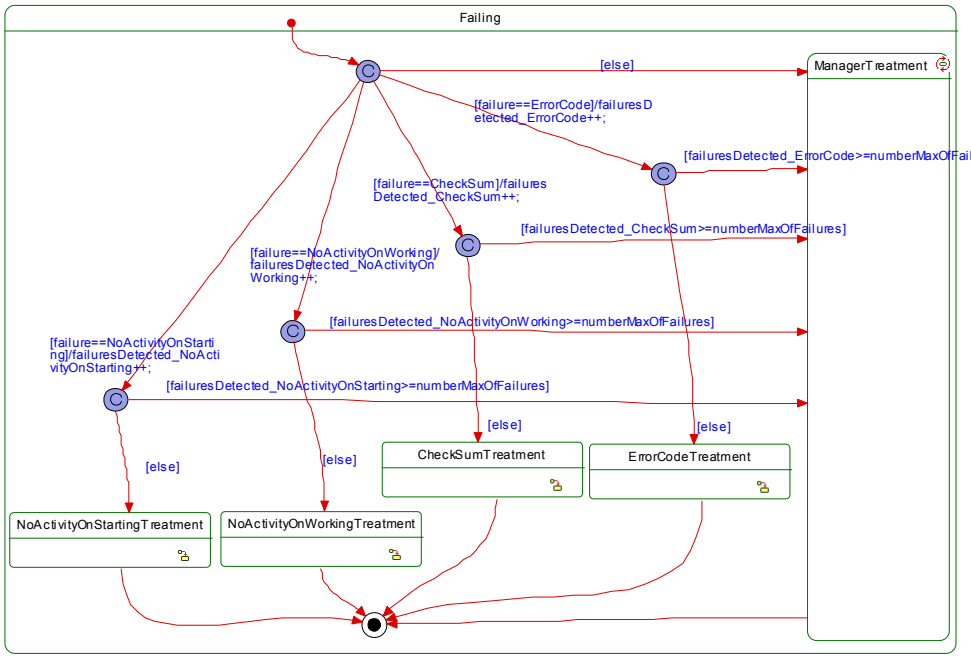

Figura 13-33: UML Statechart - Comportamento do estado Failing de SensorDevice.

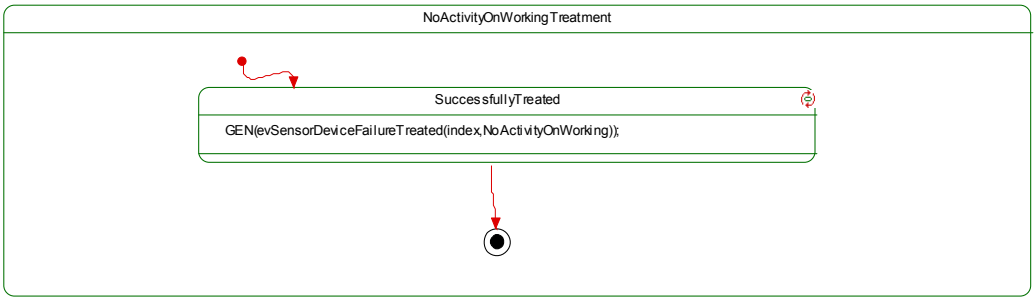

Figura 13-34: UML Statechart - Comportamento do estado Failing de SensorDevice.

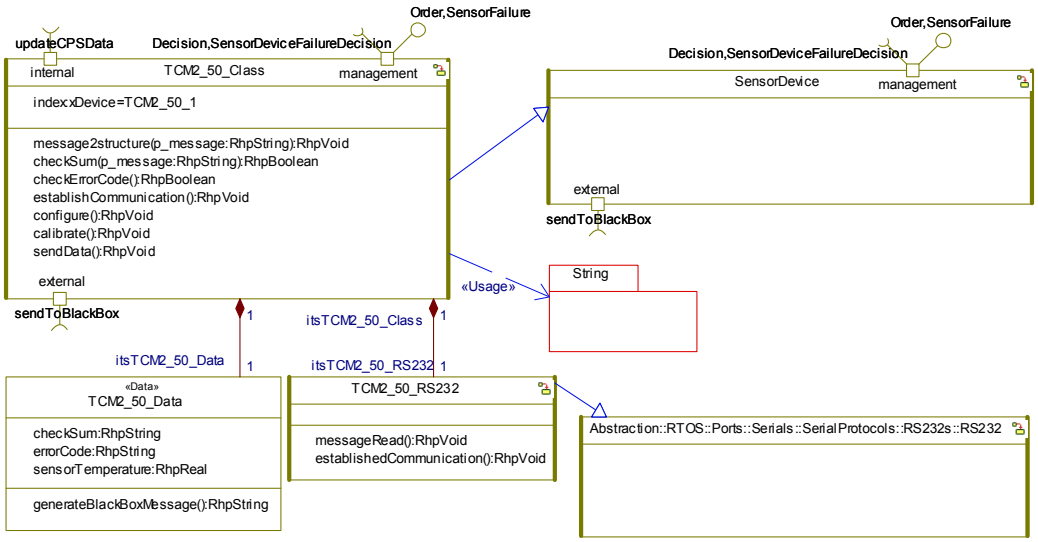

Figura 13-35: UML Object Model - Implementação do agente TCM2-50.

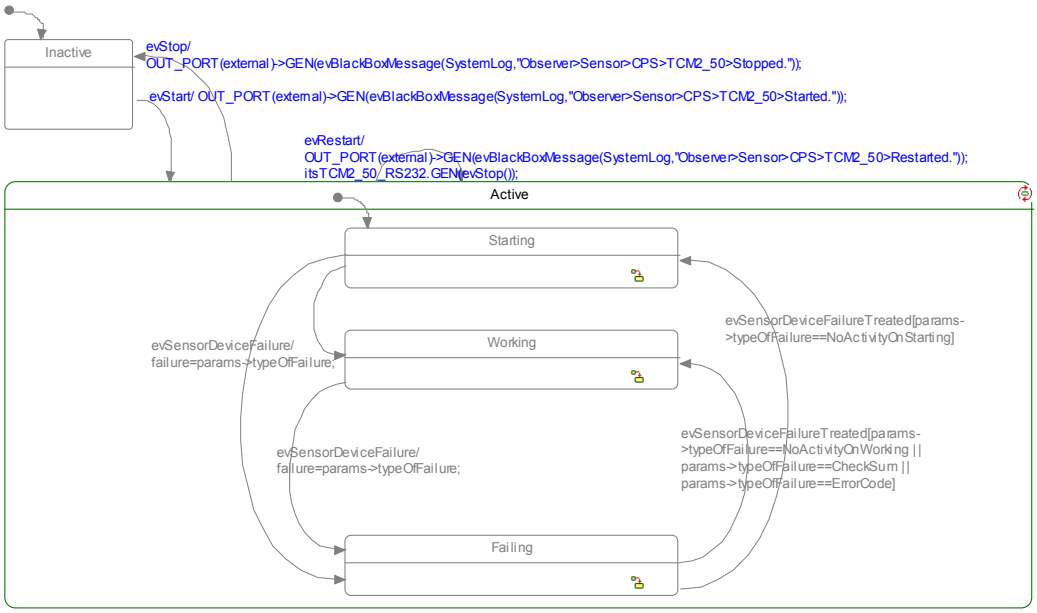

Figura 13-36: UML Statechart - Comportamento do agente TCM2-50.

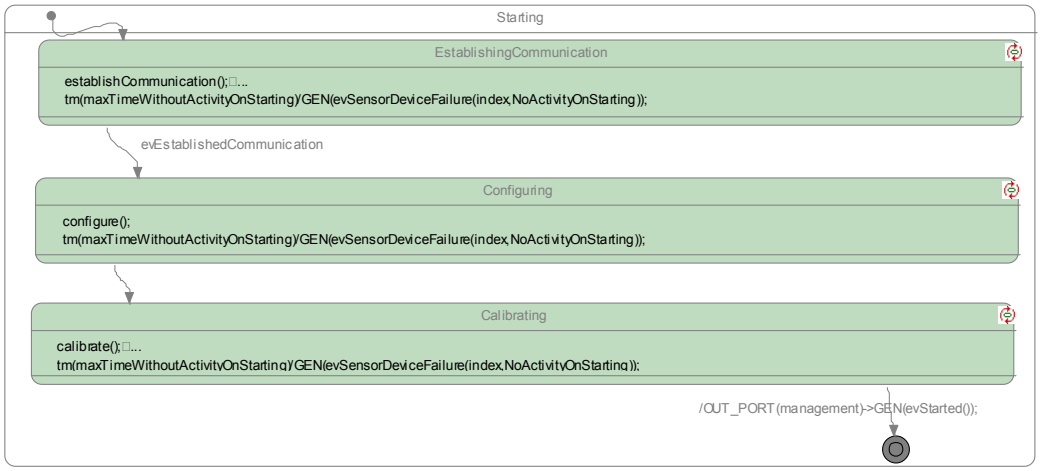

Figura 13-37: UML Statechart - Comportamento do estado Starting do agente TCM2-50.

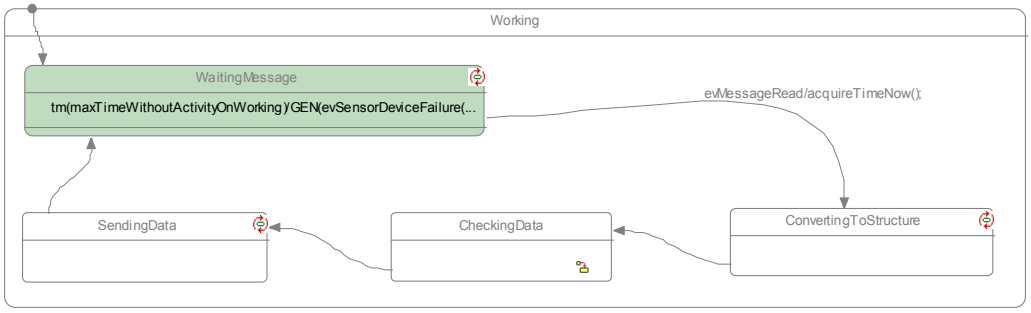

Figura 13-38: UML Statechart - Comportamento do estado Working do agente TCM2-50.

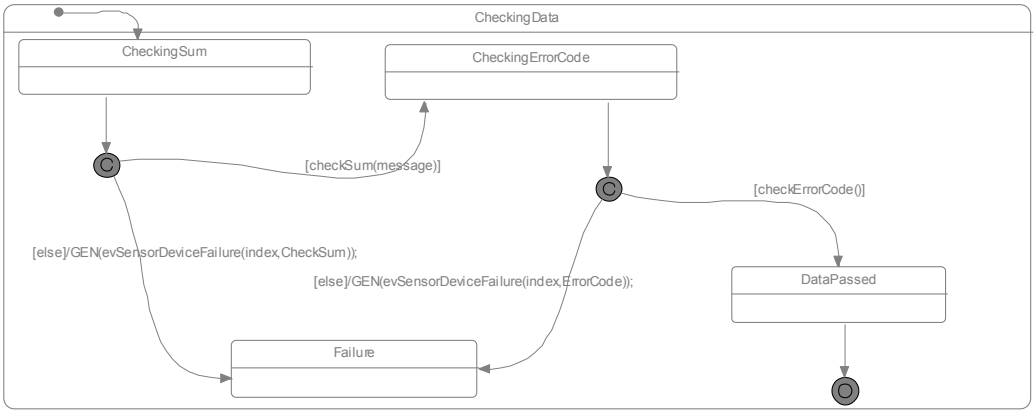

Figura 13-39: UML Statechart - Comportamento do estado CheckingData do agente TCM2-50.

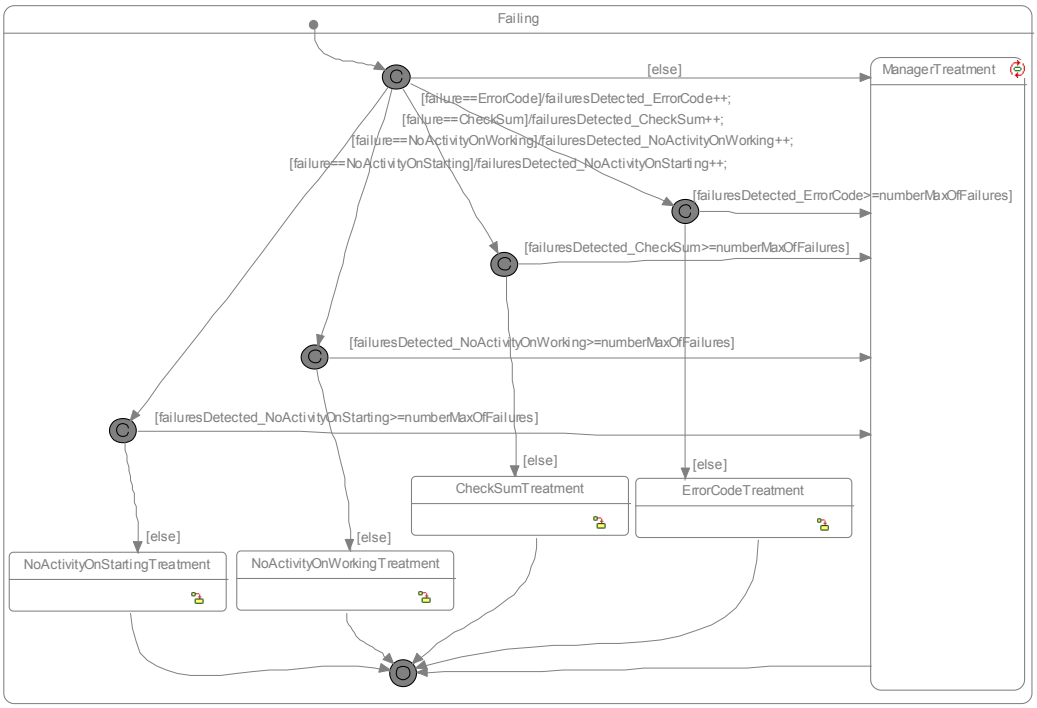

Figura 13-40: UML Statechart - Comportamento do estado Failing do agente TCM2-50.

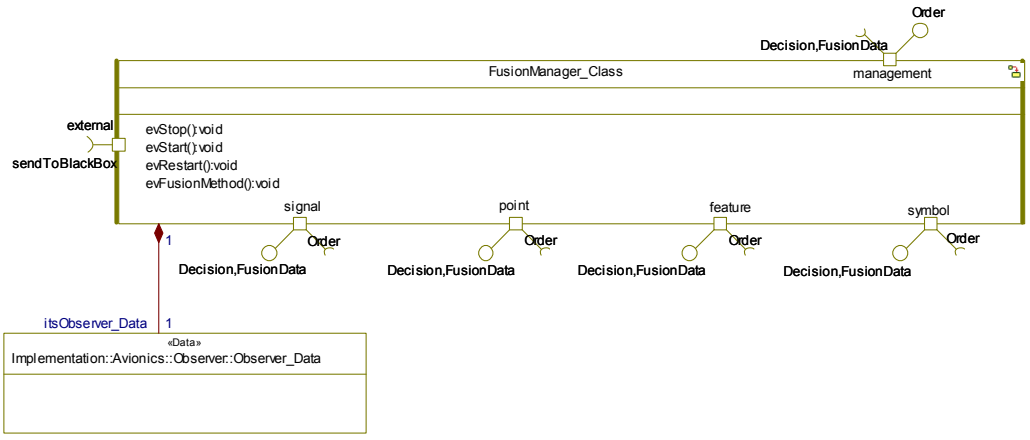

Figura 13-41: UML Object Model - Implementação do agente Fusion.

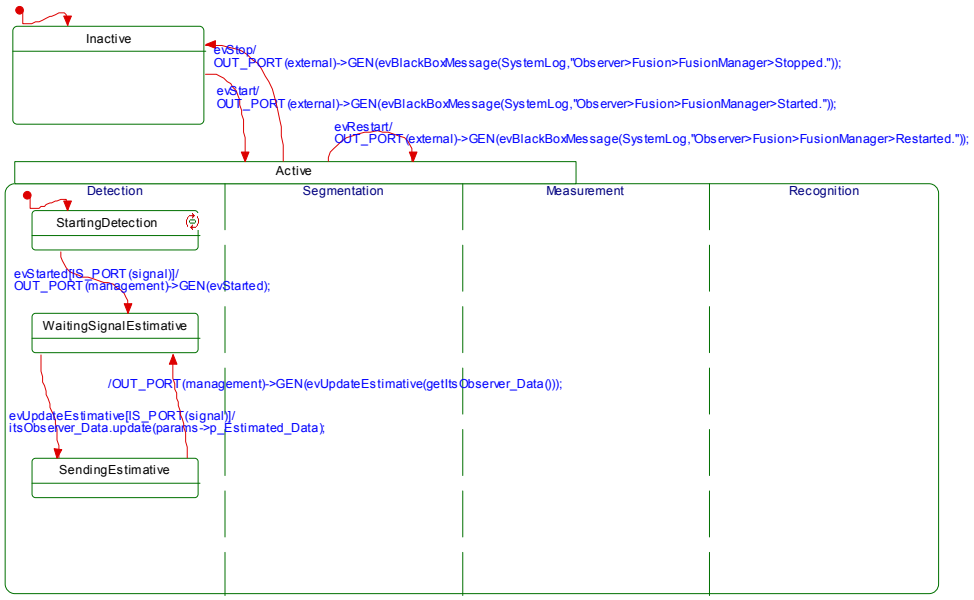

Figura 13-42: UML Statechart - Comportamento do agente Fusion.

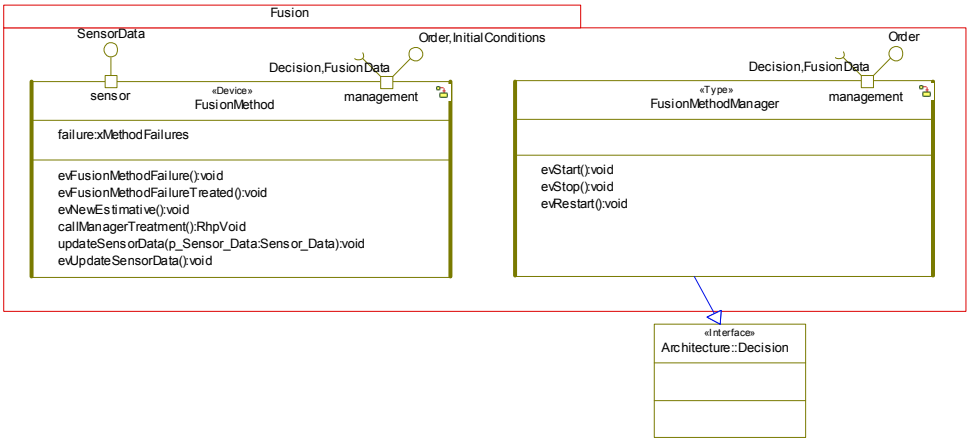

Figura 13-43: UML Object Model - Classe pai para os níveis de Dispositivo e Tipo do agente Fusion.

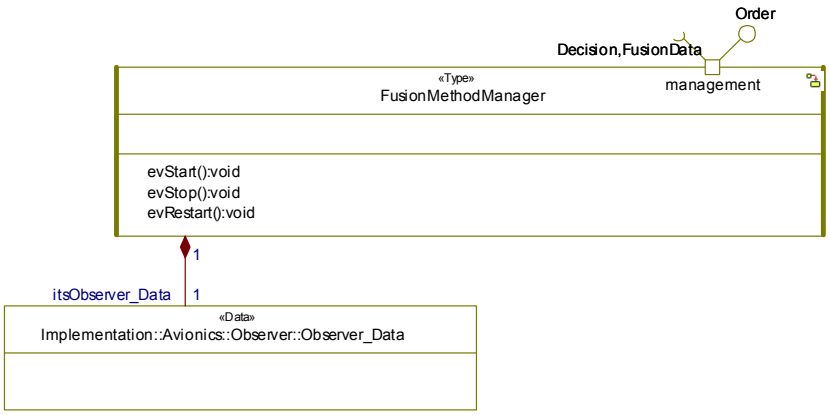

Figura 13-44: UML Object Model - Classe pai para o nível de Tipo do agente Fusion.

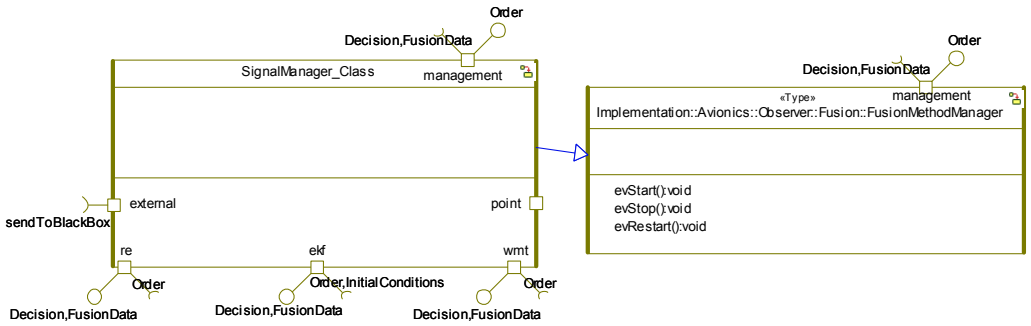

Figura 13-45: UML Object Model - Implementação do agente Signal.

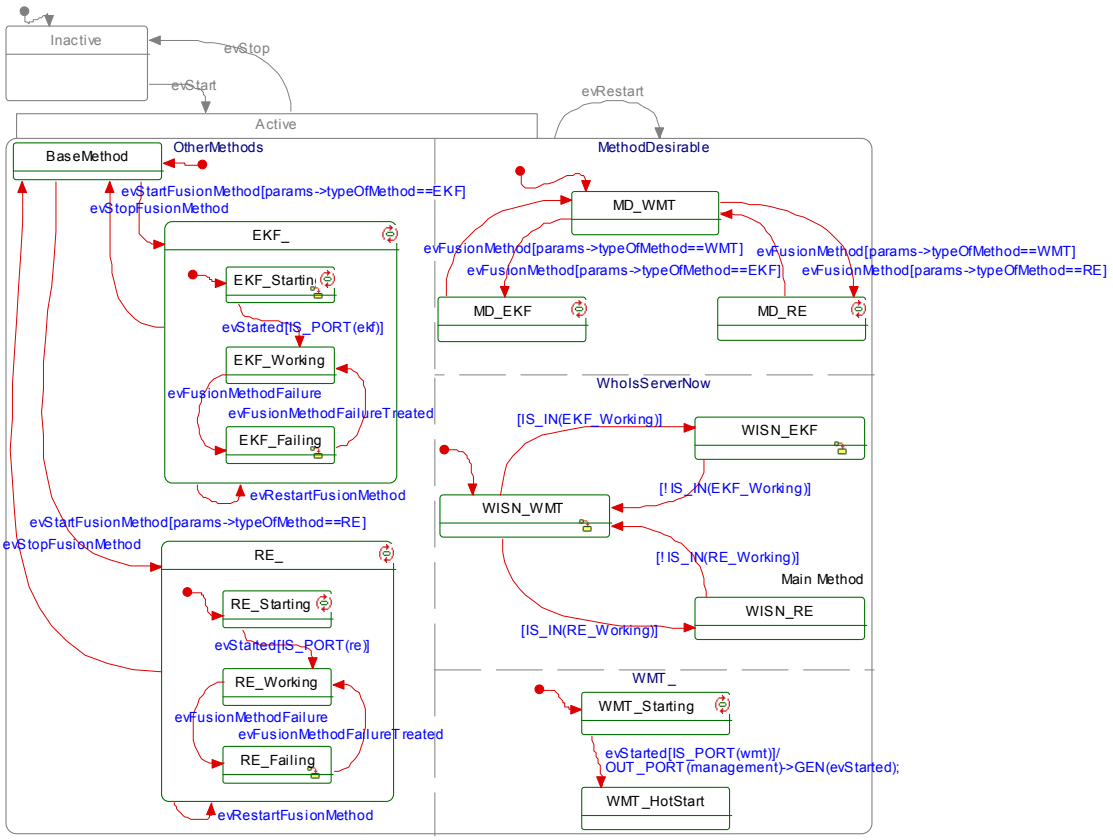

Figura 13-46: UML Statechart - Comportamento do agente Signal.

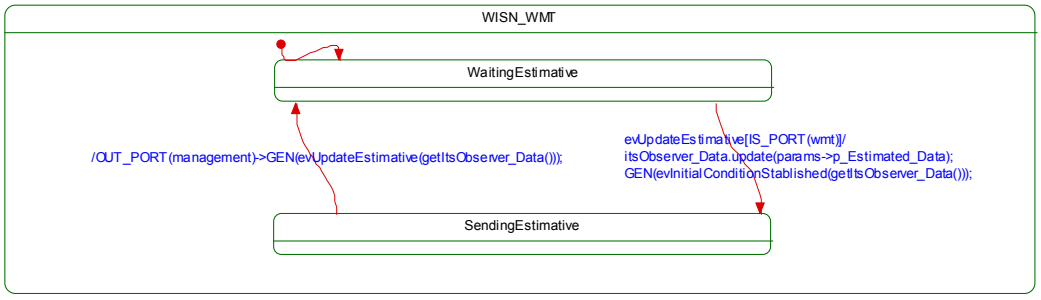

Figura 13-47: UML Statechart - Comportamento do estado WISN_WMT do agente Signal.

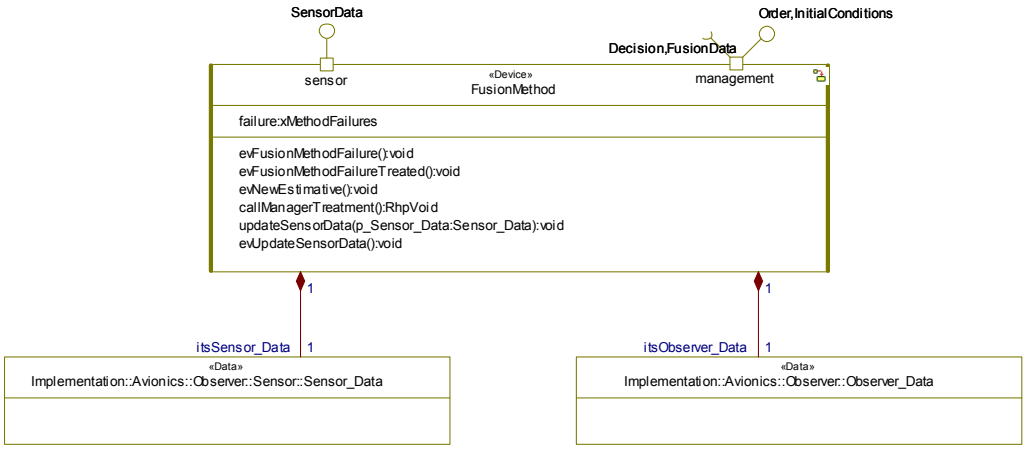

Figura 13-48: UML Object Model - Classe pai para o nível de dispositivo do agente Fusion.

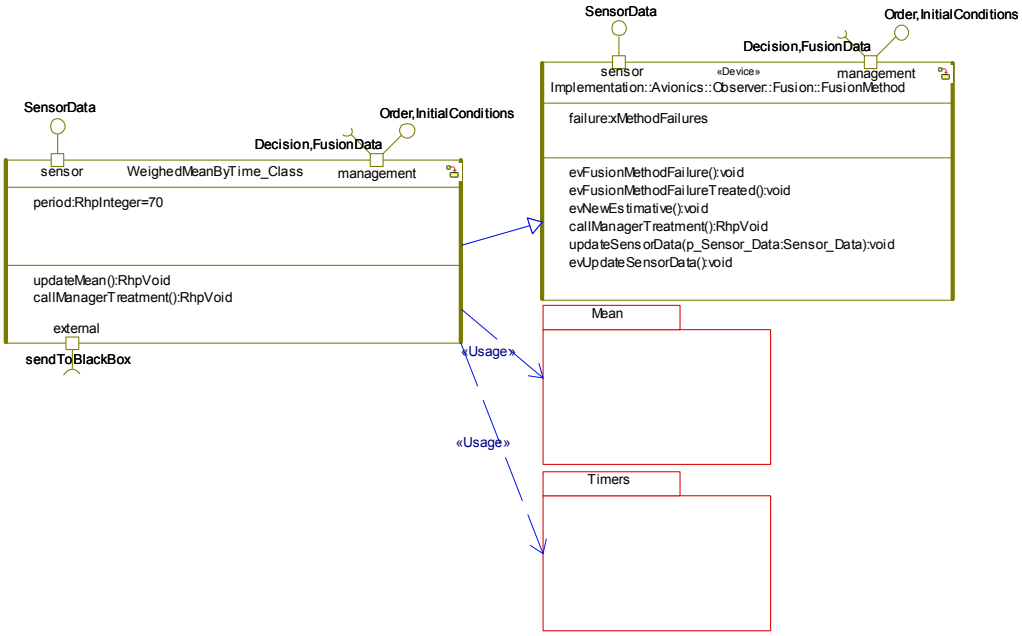

Figura 13-49: UML Object Model - Implementação do agente WMT.

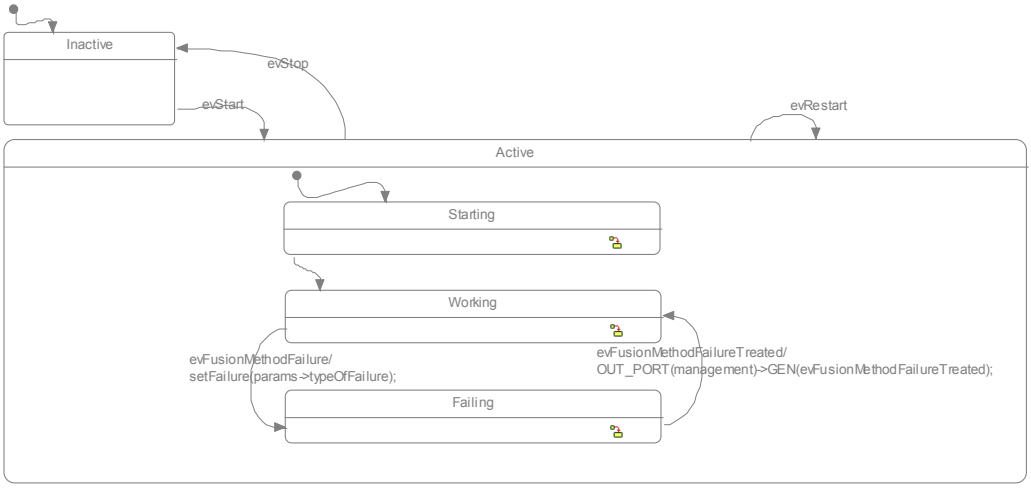

Figura 13-50: UML Statechart - Comportamento do agente WMT.

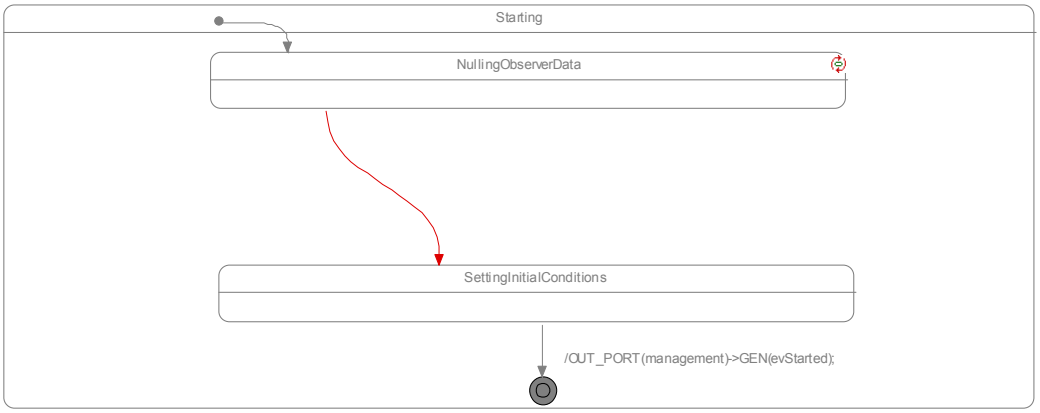

Figura 13-51: UML Statechart - Comportamento do estado Starting do agente WMT.

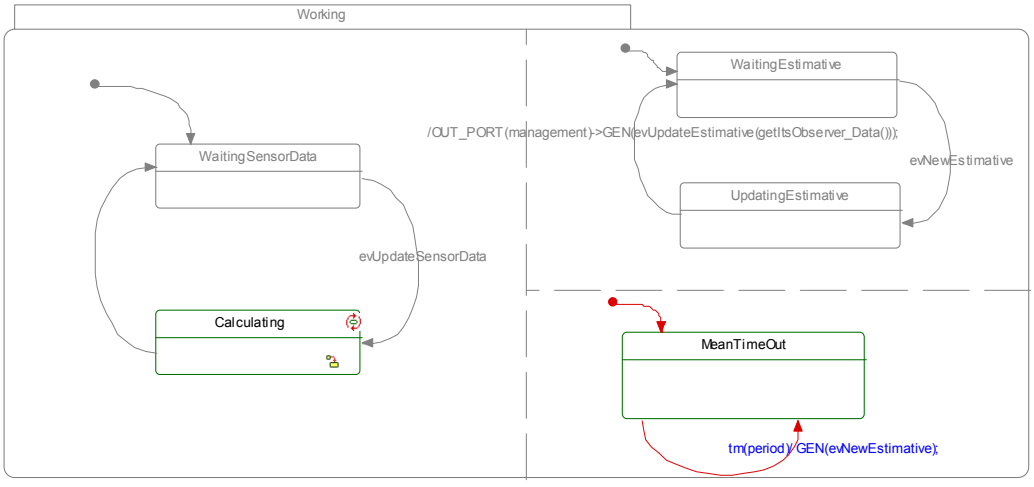

Figura 13-52: UML Statechart - Comportamento do estado Working do agente WMT.

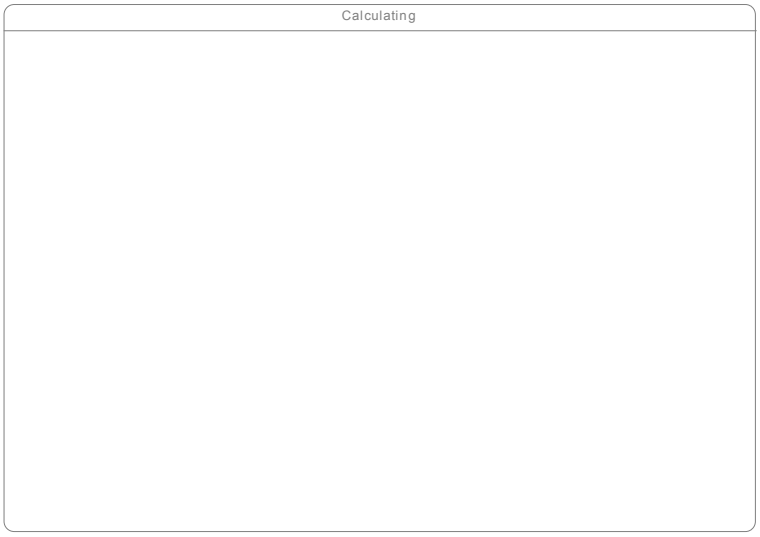

Figura 13-53: UML Statechart - Comportamento do estado Calculating do agente WMT.

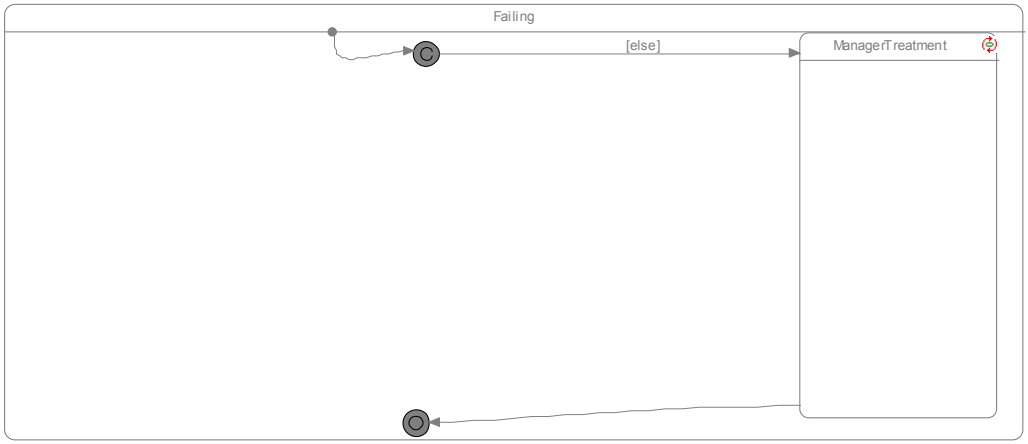

Figura 13-54: UML Statechart - Comportamento do estado Failing do agente WMT.

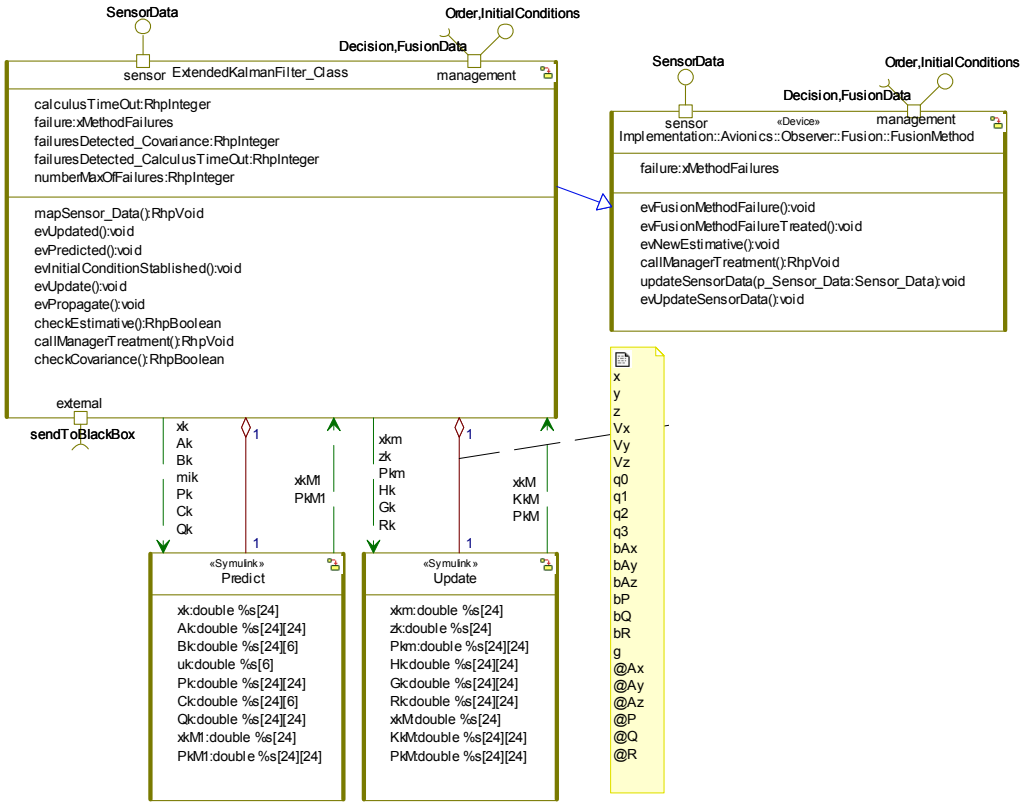

Figura 13-55: UML Object Model - Implementação do agente EKF.

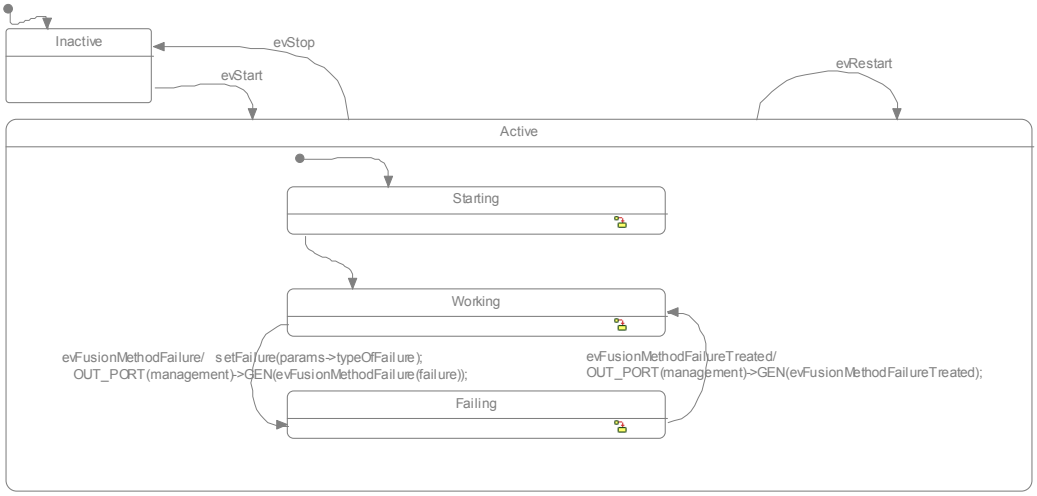

Figura 13-56: UML Statechart - Comportamento do agente EKF.

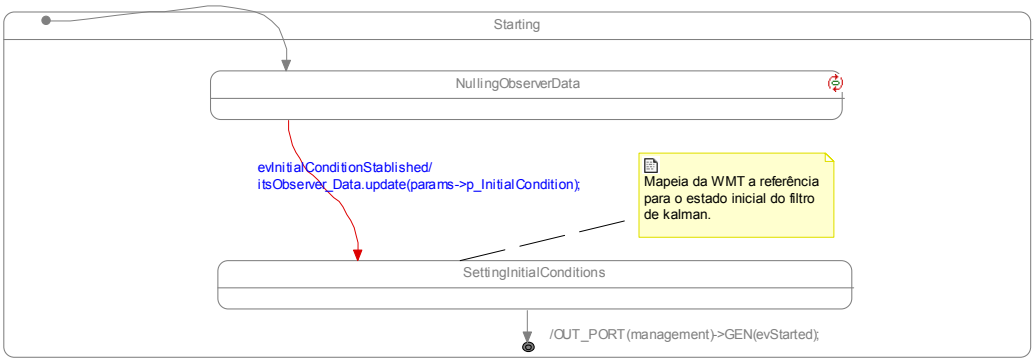

Figura 13-57: UML Statechart - Comportamento do estado Starting do agente EKF.

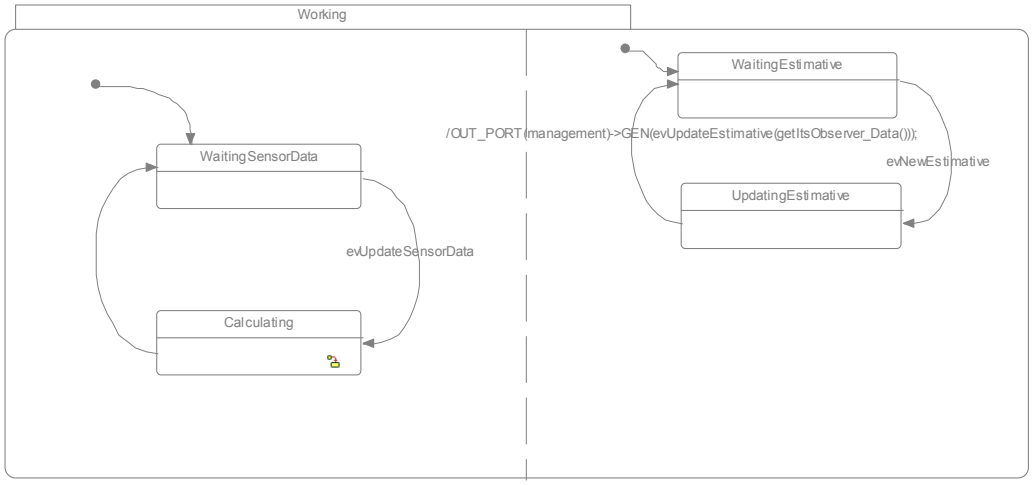

Figura 13-58: UML Statechart - Comportamento do estado Working do agente EKF.

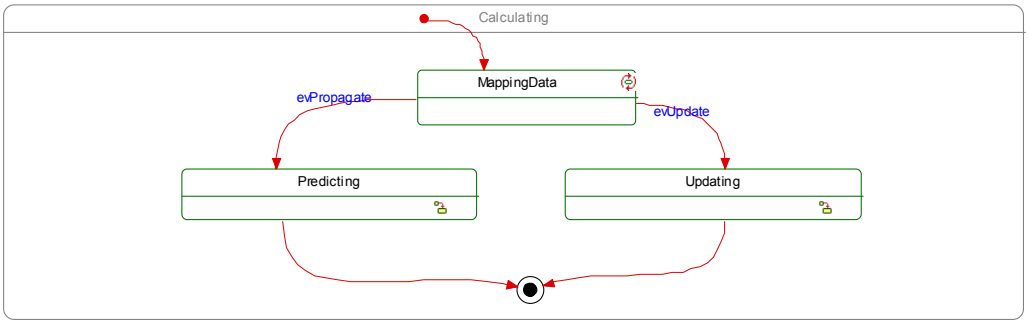

Figura 13-59: UML Statechart - Comportamento do estado Calculating do agente EKF.

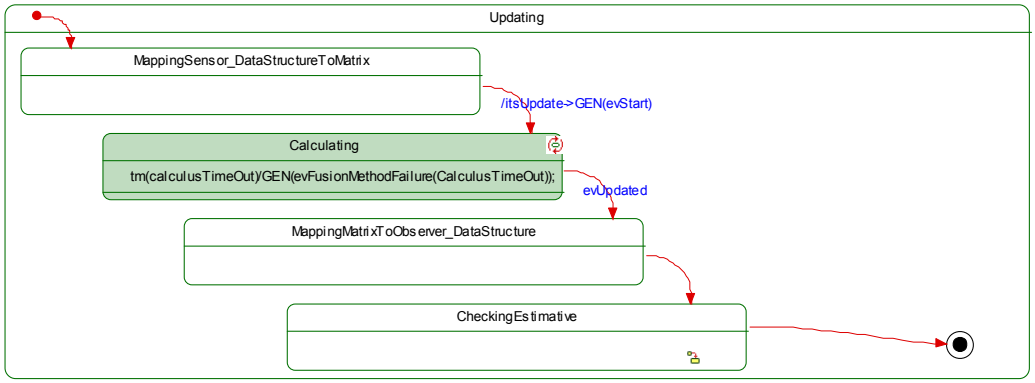

Figura 13-60: UML Statechart - Comportamento do estado Updating do agente EKF.

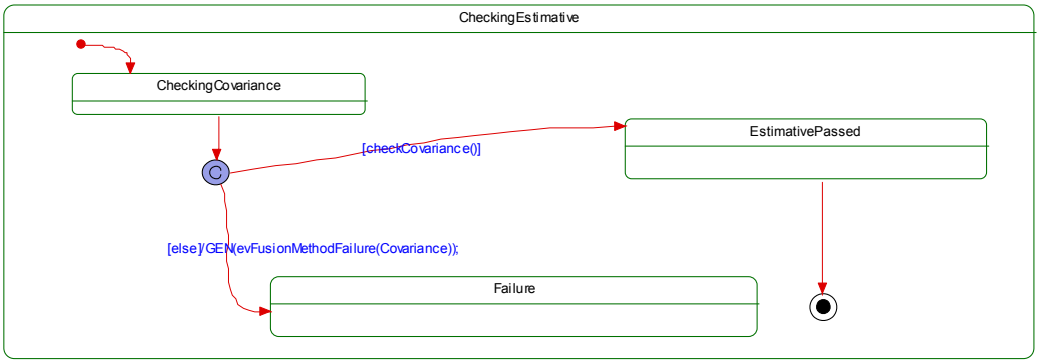

Figura 13-61: UML Statechart - Comportamento do estado Checking Estimative do agente EKF.

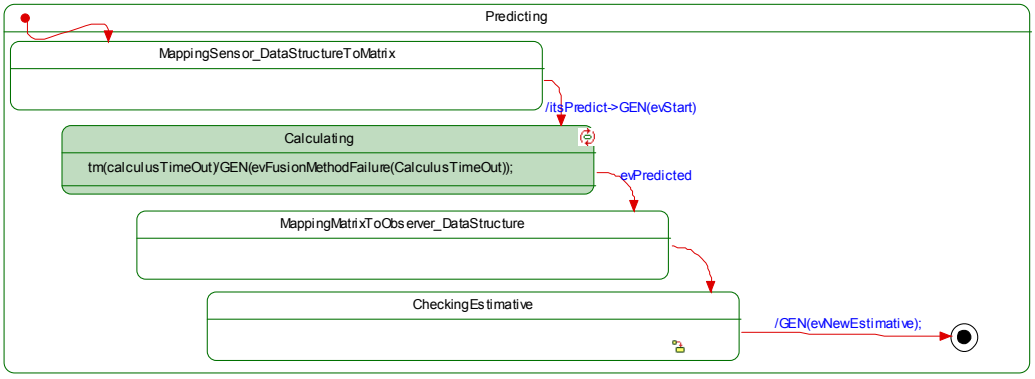

Figura 13-62: UML Statechart - Comportamento do estado Preditcting do agente EKF

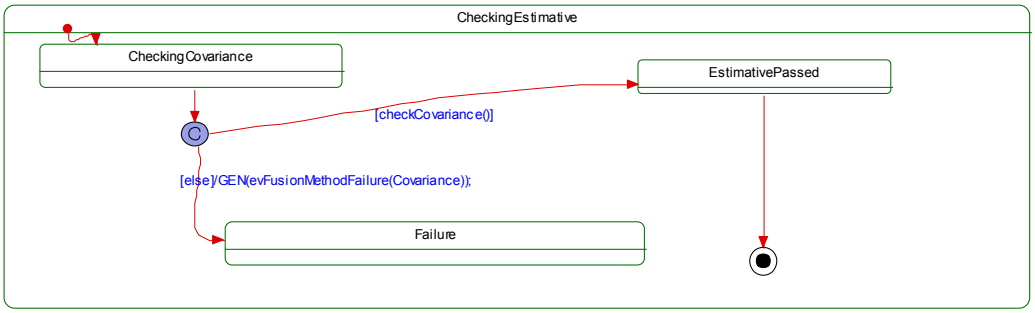

Figura 13-63: UML Statechart - Comportamento do estado CheckingEstimative do agente EKF.

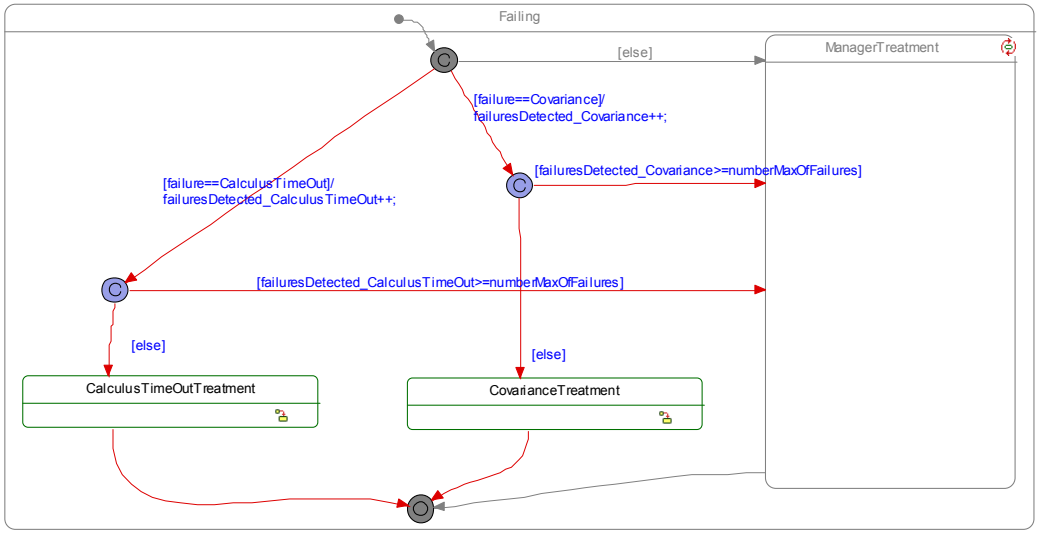

Figura 13-64: UML Statechart - Comportamento do estado Failing do agente EKF.

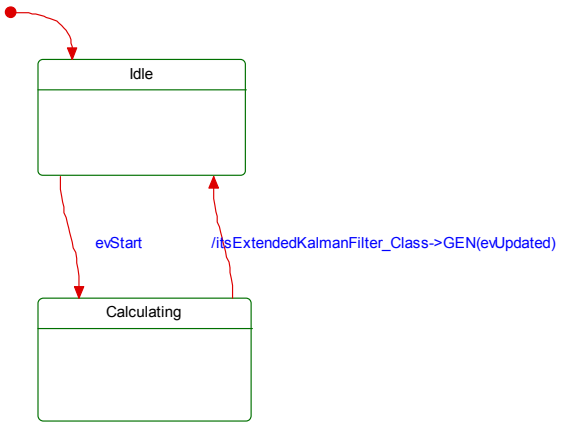

Figura 13-65: UML Statechart - Comportamento da Classe Ativa Update.

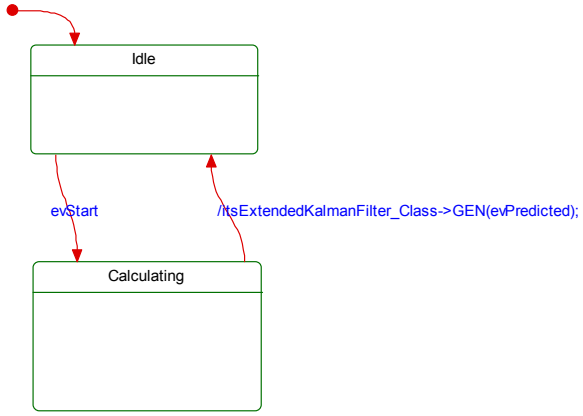

Figura 13-66: UML Statechart - Comportamento da Classe Ativa Predict.

13.3 Package Communicator

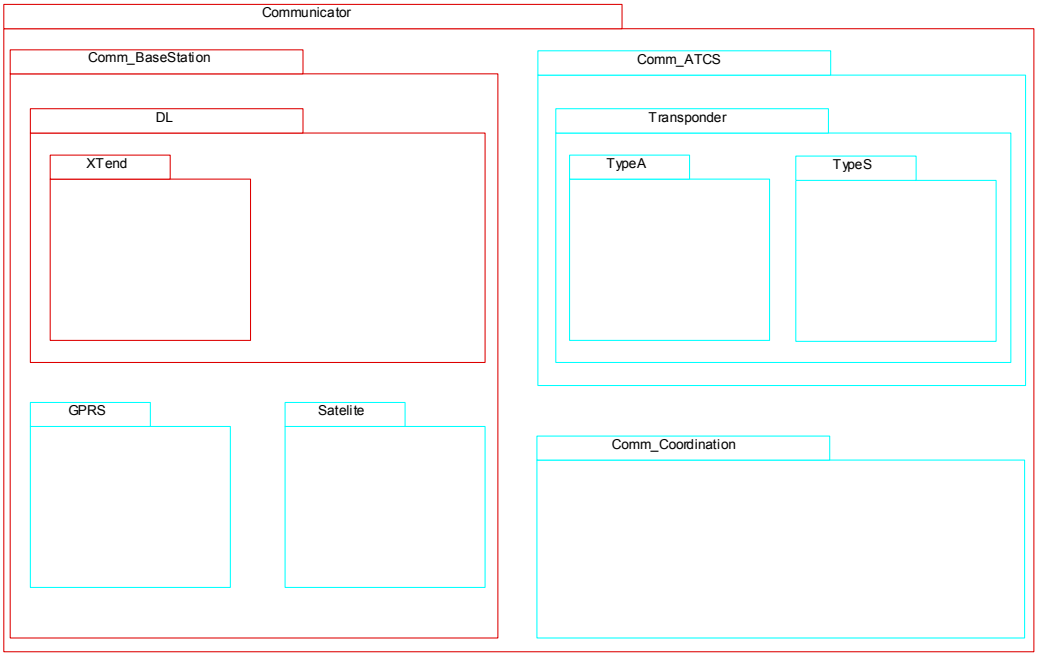

Figura 13-67: UML Object Model - Organização do agente Communicator.

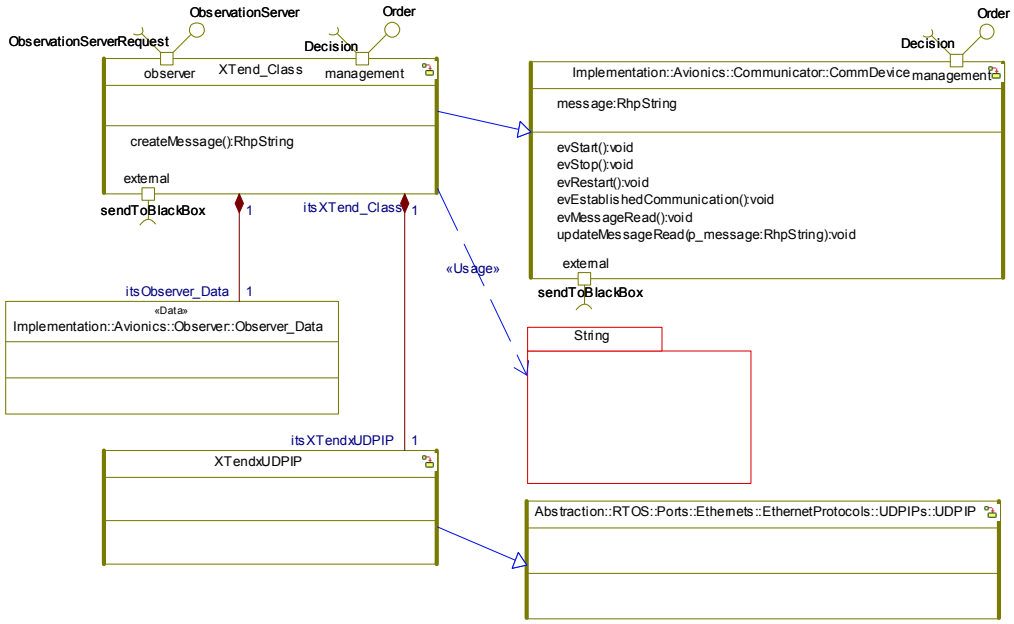

Figura 13-68: UML Object Model - Implementação do agente Xtend.

14 ANEXO - FILTRO DE MÉDIA PONDERADA NO TEMPO (WMT)

Neste anexo será apresentada a formulação do filtro de médias ponderadas no tempo.

A média ponderada da informação redundante fornecida por um grupo de sensores é um dos mais simples e intuitivos métodos de fusão no nível de sinal. A média ponderada da medida de n sensores x_i com ponderações $0 \leq w_i \leq 1$ é dada por:

$$\bar{x} = \sum_{i=1}^{n} w_i . x_i$$

Equação 14-1: WMT.

Os pesos podem ser usados para distinguir diferenças de precisão entre sensores, sendo ainda possível utilizar uma média deslizante para combinar uma seqüência de medidas adquiridas por um único sensor, atribuindo-se às mais recentes maior peso, sendo esta a definição de média ponderada no tempo (WMT).

15 ANEXO - FILTRO DE KALMAN ESTENDIDO (EKF)

Neste anexo será apresentada a formulação do filtro de kalman estendido implementado no computador embarcado em conjunto com o trabalho realizado por Zanoni (2007).

O filtro de kalman é empregado com objetivo de estimar os estados $x_k \in R^n$ de um processo estocástico com o auxílio de medições ruidosas $z_k \in R^m$, conforme representado abaixo:

$$x_{k+1} = A \cdot x_k + B \cdot u_k + C.w_k$$

Equação 15-1: Dinâmica de processo estocástico.

$$\mathrm{z_k} = H \cdot x_k + G \cdot v_k$$

Equação 15-2: Medição.

Onde, A é a matriz de transição de estados, B é a matriz de entradas de controle, C é a matriz de ruídos de processo, H é a matriz de leitura dos sensores e G é a matriz de ruídos de medição. x_k é o vetor de estados, u_k é vetor entrada de controle, z_k é o vetor de medições do processo no instante t_k, e w_k e v_k são ruídos brancos de media nula do processo e da medição, respectivamente.

O filtro de kalman corresponde a uma sucessão de propagação e atualizações. Teremos, então:

Ciclo de Propagação

$$\overline{x}_{k+1} = A \cdot \hat{x}_k + B \cdot u_k$$

Equação 15-3: Propagação da média.

$$\overline{P}_{k+1} = A \cdot \hat{P}_k \cdot A^T + C \cdot Q \cdot C^T$$

Equação 15-4: Propagação da covariância de processo.

Ciclo de Atualização

$$K_k = \overline{P}_k \cdot H^T \left[H \cdot \overline{P}_k \cdot H^T + G \cdot R \cdot G^T \right]^{-1}$$

Equação 15-5: Ganho de kalman.

$$\hat{x}_k = \hat{\overline{x}}_k + K_k \cdot \left[z_k - H \cdot \hat{\overline{x}}_k \right]$$

Equação 15-6: Correção da média.

$$\hat{P}_k = \overline{P}_k - K_k \cdot H \cdot \overline{P}_k$$

Equação 15-7: Correção da covariância de processo.

Onde, Q é a covariância do ruído do processo e R é a covariância do ruído de medição, P é a matriz covariância dos estados. O sobrescrito "–" é para valores na propagação e "^" para valores imediatamente após a atualização.

A versão não linear do filtro de kalman é chamada de filtro de kalman estendido (EKF-*Extended Kalman Filter*). De acordo com Brown (1997) o processo de estimação pode ser representado por:

$$\dot{x}(\mathrm{t}) = f(x,u,t) + w(t)$$

Equação 15-8: Dinâmica de processo estocástico não linear.

$$z = h(x,t) + v(t)$$

Equação 15-9: Medição não linear.

Onde f e h são funções conhecidas e w e v são ruídos brancos de média nula. Como no filtro linear, o estendido tem os ciclos de propagação e atualização.

Ciclo de Propagação

$$\overline{x}_{\mathrm{k+1}} = \hat{x}_k + f(x,u,t).T$$

Equação 15-10: Propagação da média não linear.

$$\overline{P}_{\mathrm{k+1}} = A_k.\hat{P}_k A_k^{\ T} + Q_k$$

Equação 15-11: Propagação da covariância de processo não linear.

Onde $A_k = \left[I + \frac{\partial f}{\partial x}(x_k, u_k, t_k).\Delta t \right]$ é a linearização e discretização do processo e $Q_k = E(w_k, w_k^{\ T})$ é o ruído de processo não linear.

Ciclo de Atualização

$$K_k = \overline{P}_k.H_k^{\ T}\left[H_k.\overline{P}_k.H_k^{\ T} + R_k\right]^{-1}$$

Equação 15-12: Ganho de kalman não linear.

$$\hat{x}_k = \hat{\bar{x}}_k + K_k \cdot \left[z_k - H_k \cdot \hat{\bar{x}}_k \right]$$

Equação 15-13: Correção da média não linear.

$$\hat{P}_k = \overline{P}_k - K_k \cdot H_k \cdot \overline{P}_k$$

Equação 15-14: Correção da covariância de processo não linear.

Onde $H_k = \frac{\partial h}{\partial x}(x_k, t_k)$ é a linearização da medição e $R_k = E(v_k, v_k^T)$ é o ruído de medição não linear.

O modelo desenvolvido consiste no seguinte vetor de estados:

$$x = \begin{bmatrix} \lambda & \phi & h & V_N & V_E & V_D & a & b & c & d & g & b_{a_x} & b_{a_y} & b_{a_z} & b_{g_x} & b_{g_y} & b_{g_z} \end{bmatrix}$$

Onde: $\begin{bmatrix} \lambda & \phi & h \end{bmatrix}$ é a latitude, longitude e altitude, $\begin{bmatrix} V_N & V_E & V_D \end{bmatrix}$ é a velocidade da plataforma no sistema de coordenadas NED, $\begin{bmatrix} a & b & c & d \end{bmatrix}$ são os elementos do quartenions, g é a aceleração da gravidade no sistema de coordenada NED, $\begin{bmatrix} b_{a_x} & b_{a_y} & b_{a_z} \end{bmatrix}$ são as *bias* dos acelerômetros no sistema de coordenadas da plataforma, $\begin{bmatrix} b_{g_x} & b_{g_y} & b_{g_z} \end{bmatrix}$ são as *bias* dos giroscópios no sistema de coordenadas da plataforma.

E consiste no seguinte vetor de controle: $u = \begin{bmatrix} a_x & a_y & a_z & \omega_x & \omega_y & \omega_z \end{bmatrix}$ onde $\begin{bmatrix} a_x & a_y & a_z \end{bmatrix}$ e $\begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}$ são as acelerações e taxas de rotação no corpo, respectivamente.

A equação dinâmica de processo $f(x,u,t)$ desenvolvida pode ser observada abaixo:

$$f(x,u,t)=\begin{bmatrix} \frac{V_N}{(R_\lambda+h)} \\ \frac{V_E}{(R_\phi+h).\cos(\lambda)} \\ -V_D \\ -\frac{V_E{}^2.\sin(\lambda)}{(R_\phi+h).\cos(\lambda)}+\frac{V_N.V_D}{(R_\lambda+h)}-2.V_D.\Omega.\sin(\lambda)+\left[a_N-b_{a_N}\right] \\ \frac{V_E.V_N.\sin(\lambda)}{(R_\phi+h).\cos(\lambda)}+\frac{V_E.V_D}{(R_\lambda+h)}+2.\Omega.(V_N.\sin(\lambda)+V_D.\cos(\lambda))+\left[a_E-b_{a_E}\right] \\ -\frac{V_E{}^2}{(R_\phi+h)}-\frac{V_N{}^2}{(R_\lambda+h)}-2.V_E.\Omega.\cos(\lambda)+g+\left[a_D-b_{a_D}\right] \\ \frac{1}{2}\begin{bmatrix} -b & -c & -d \\ a & -d & c \\ d & a & -b \\ -c & b & a \end{bmatrix}.\begin{bmatrix} \omega_x-b_{g_x} \\ \omega_y-b_{g_y} \\ \omega_z-b_{g_z} \end{bmatrix} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

Equação 15-15: Equação de dinâmica de processo não linear.

Onde $\left[a_N-b_{a_N}\right]$, $\left[a_E-b_{a_E}\right]$ e $\left[a_D-b_{a_D}\right]$ são obtidos conforme:

$$\begin{bmatrix} a_N-b_{a_N} \\ a_E-b_{a_E} \\ a_D-b_{a_D} \end{bmatrix}=\begin{bmatrix} (a^2+b^2-c^2-d^2) & 2.(b.c-a.d) & 2.(b.d+a.c) \\ 2.(b.c+a.d) & (a^2-b^2+c^2-d^2) & 2.(c.d-a.b) \\ 2.(b.d-a.c) & 2.(c.d+a.b) & (a^2-b^2-c^2+d^2) \end{bmatrix}.\begin{bmatrix} a_x-b_{a_x} \\ a_y-b_{a_y} \\ a_z-b_{a_z} \end{bmatrix}$$

A equação do ruído de processo $Q_k=E(w_k,w_k{}^T)$ foi desenvolvida conforme a seguinte método:

$$Q_k=\left[\frac{\partial f}{\partial\begin{bmatrix} x & u \end{bmatrix}}(x_k,u_k,t_k)\right].\begin{bmatrix} w & 0 \\ 0 & \begin{bmatrix} w_a & 0 \\ 0 & w_\omega \end{bmatrix} \end{bmatrix}.\left[\frac{\partial f}{\partial\begin{bmatrix} x & u \end{bmatrix}}(x_k,u_k,t_k)\right]^T$$

Equação 15-16: Ruído de processo não linear.

Onde w, w_a e w_ω são ruídos do processo, acelerômetros e giros, respectivamente.

A equação de medição do GPS e da CPS desenvolvidas podem ser observadas abaixo:

$$h_{GPS}=\left[I^{3x3}\mid 0^{3x14}\right].x$$

Equação 15-17: Equação de medição do GPS não linear.

$$h_{CPS} = \left[C_p^n\right]^{-1} \begin{bmatrix} B_N \\ B_E \\ B_D \end{bmatrix}$$

Equação 15-18: Equação de medição do CPS não linear.

Finalmente os ruídos de medição do GPS e CPS representado por δ desenvolvidos podem ser observados abaixo:

$$R_{GPS} = \begin{bmatrix} \delta_\lambda^2 & 0 & 0 \\ 0 & \delta_\phi^2 & 0 \\ 0 & 0 & \delta_h^2 \end{bmatrix}$$

Equação 15-19: Ruído de medição do GPS não linear.

$$R_{CPS} = \begin{bmatrix} \delta_{B_X}^2 & 0 & 0 \\ 0 & \delta_{B_Y}^2 & 0 \\ 0 & 0 & \delta_{B_Z}^2 \end{bmatrix}$$

Equação 15-20: Ruído de medição do CPS não linear.

16 ANEXO - TABELA DE PRIORIDADES

Priority	High Level Agent	Thread Name	Thread OS ID
63	Reservado para uso futuro		
62	Reservado para uso futuro		
61	Reservado para uso futuro		
60	Reservado para uso futuro		
59	Reservado para uso futuro		
58	Manager	Manager	1
57			
56			
55			
54			
53			
52			
51		BlackBox	
50	Observer	ObserverManager	56
49		SensorManager	53
48		FusionManager	35
47		CPSMAnager	37
		GPSMAnager	44
		ADBManager; TDManager; EDManager; EOManager; IRManager	
46		TCM2_50_RS232	39
		InterruptDrivenRead	40
		MiLLennium_RS232	46
		InterruptDrivenRead	47
45		TCM2_50	38
		PropakII	45
44		IMUManager	48
43		VG600AA_RS232	50
		InterruptDrivenRead	51
42		VG600AA	49
41		SignalManager	22
40		WeighedMeanTime	31
39		ExtendedKalmanFilter; RobustEstimator	
38		PointManager; FeatureManager; SymbolManager	
37			
36			
35	Controller	ControllerManager	
34			
33			
32			
31			
30			
29			
28			
27			
26	Actuator	ActuatorManager	
25			
24			
23			
22			
21			
20	Communicator	CommunicatorManager	
19		BaseStationManager; ATCSManager; CoordinationManager	
18		DataLinkManager; SateliteManager; GPRSManagemer; TransponderManager	
17		Xtend_UDPIP	11
		InterruptDrivenRead	12
16		Xtend	10
		TypeC	
15			
14	Payload	PayloadManager	
13			
12			
11			
10			
9			
8			
7			
6			
5			
4	Reservado para uso futuro		
3	Reservado para uso futuro		
2	Reservado para uso futuro		
1	Reservado para uso futuro		
0	Reservado para uso futuro		

17 ANEXO - MODIFICAÇÕES NA RHAPSODY OXF

Este anexo trata das modificações implementadas na OXF do Rhapsody.

-Inserção da função de nomeação de *thread* (pthread_setname_np(threadID, name))na função da OXF de criação da POSIX *thread* no RTOS QNX. A função pthread_setname_np() não pertence à norma POSIX (que não provê recursos de nomeação de threads), entretanto revelou-se imprescindível para a identificação das dezenas de *threads* no sistema.

```
QNXThread::QNXThread(void tfunc(void *), void *param, const char* const name, const long stackSize)
                                : endOSThreadInDtor(TRUE)
{
    isWrapperThread = 0;
    m_SuspEventFlag = new QNXEventFlag();

    m_SuspEventFlag->reset();
    // Create SUSPENDED thread !!!!!!
    m_ExecFunc = tfunc;
    m_ExecParam = param;

    signal(SIGUSR1,User1handler);
    signal(SIGUSR2,User2handler);

    // Set thread attributes prior to thread creation.
    pthread_attr_t tthread_attr;
    pthread_attr_init( &tthread_attr );
    pthread_attr_setstacksize( &tthread_attr, (size_t)stackSize );
    pthread_attr_setdetachstate( &tthread_attr, PTHREAD_CREATE_DETACHED);
    pthread_attr_setinheritsched( &tthread_attr, PTHREAD_INHERIT_SCHED );

    pthread_create (&hThread, &tthread_attr , (FormalThrCreateFunc)preExecFunc, (void*)this );
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////MODIFICAÇÃO NA OXF ORIGINAL////////////////////////////////
    pthread_setname_np(hThread,name);
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo