UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ATHOS ALEXANDRE LIMA FONTANARI

Sistema de planejamento e controle de missão de um veículo aéreo não-tripulado aplicado em redes de sensores sem fio

Trabalho de Graduação

Prof. Dr. Flávio Rech Wagner
Orientador

Prof. Dr. Carlos Eduardo Pereira
Co-orientador

Porto Alegre, novembro de 2011.

AGRADECIMENTOS

Agradeço inicialmente a Deus por me abençoar com força, coragem e saúde em todos os momentos que precisei. Agradeço aos meus pais Athos José Mahfuz Fontanari e Maria Conceição Lima Fontanari pelo carinho e suporte em todos os momentos da minha vida, aos meus irmãos Rodrigo Lima Fontanari e Lígia Lima Fontanari pelo apoio e incentivo e principalmente por sempre estarem presentes na minha vida.

Agradeço a família e aos amigos pelos momentos que compartilhamos e que sempre estiveram presentes em todos os momentos.

Agradeço ao meu orientador Flávio Rech Wagner pela oportunidade de desenvolver este trabalho e aos meus co-orientadores Edison Pignaton e Rodrigo Allgayer pelo incentivo e pelos ensinamentos que tornaram possíveis o desenvolvimento deste trabalho. Finalmente, agradeço ao leitor pela oportunidade de compartilhar este conhecimento e que este trabalho sirva como inspiração ou fonte para trabalhos futuros na área.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

VANT	Veículo aéreo não-tripulado
BR	Brasil
UFRGS	Universidade Federal do Rio Grande do Sul
SMA	Sistemas Multiagentes
MDL	Mission Description Language
XML	Extensible Markup Language
RSSF	Rede de sensores sem fio
FIPA	Foundation for Intelligent Physical Agents

LISTA DE FIGURAS

LISTA DE TABELAS

LISTINGS

RESUMO

Este projeto descreve um sistema de planejamento e controle em alto nível para gerenciamento de uma missão de um veículo aéreo não-tripulado. O VANT é utilizado como um sensor móvel em uma rede de sensores sem fio. Para atingir este objetivo foi desenvolvida uma linguagem de descrição de missões chamada MDL baseada na especificação da SensorML. SensorML é uma linguagem de markup utilizada para descrever sensores e processos de medida em redes de sensores sem fio. O controle da missão é modelado considerando cada componente do sistema como um agente em um sistema multiagente para especificar um comportamento autônomo aos componentes da missão.

Palavras-Chave: redes de sensores sem fio, veículo aéreo não-tripulado, SensorML, sistemas multiagentes.

Mission planning and control system for an unmanned aerial vehicle applied in wireless sensor network

ABSTRACT

This project reports a planning and control system for high-level management of missions in an unmanned aerial vehicle. The UAV is used as a mobile sensor in a wireless sensor network. To achieve this goal it was developed a mission description language called MDL that is based on SensorML specification. SensorML is a markup language used to describe sensors and measurement process in wireless sensor networks. The mission control system is modeled considering each component of the system as an agent in a multiagent system to specify an autonomous behavior to the components of a mission.

Keywords: wireless sensor networks, unmanned aerial vehicle, SensorML, multiagent system.

1 INTRODUÇÃO

Redes de sensores sem fio têm sido utilizadas em inúmeras aplicações com o objetivo de monitorar e estudar um determinado fenômeno e prover informações a sistemas computacionais que auxiliem o homem na tomada de decisão. Estas redes de sensores tem grande aplicação em locais de difícil acesso ou áreas perigosas. Uma maneira de aumentar as possibilidades de uso das redes de sensores e fazer com que elas sejam capazes de fornecer informações mais ricas se dá pelo uso de sensores com capacidades e características heterogêneas. Estas características vão desde o tipo de dado que o sensor é capaz de fornecer, passando pelas diferentes plataformas computacionais que suportam a sua atividade, até chegar a sua capacidade de mobilidade.

A utilização de nós sensores móveis aumenta significativamente a capacidade e a aplicabilidade de uma rede de sensores sem fio. Neste contexto, destaca-se o uso de sensores capazes de se mover em três dimensões, como os sensores embarcados em veículos aéreos, por exemplo. Dentre estes tipos de sensores, destaca-se os embarcados em Veículos Aéreos Não-Tripulados (VANT). Um VANT, também chamado UAV, do inglês *Unmanned Aerial Vehicle*, é o termo usado para descrever todo e qualquer tipo de aeronave que não necessita de tripulação para ser guiada. Este tipo de aeronave é controlada à distância através de computadores sob a supervisão de humanos ou, em alguns casos, sem a sua intervenção por meio de computadores embarcados (FERREIRA, A. M., 2008).

Os VANTs podem ser utilizados em diversas aplicações, tais como: aplicações militares, vigilância policial de áreas urbanas e de fronteira, inspeções de linhas de transmissão de energia, monitoramento de dutos de petróleo, levantamento de áreas agrícolas, acompanhamento de safra, controle de pragas e de queimadas. O custo da utilização de um VANT para aplicações como estas é relativamente baixo quando comparado à utilização de aeronaves tripuladas.

O avanço das tecnologias computacionais e o aumento da complexidade destas redes permitiram a atuação destes sistemas de maneira autônoma para realizar as tarefas determinadas pelo usuário, exigindo a intervenção humana apenas em situações críticas. Uma rede de sensores sem fio é composta pelo sensor, o observador e o fenômeno. O sensor é responsável por monitorar o fenômeno que está sendo analisado e enviar as informações ao observador, que é o sistema computacional responsável pela tomada de decisões. Neste trabalho é desenvolvida uma linguagem de alto nível para planejar e gerenciar o movimento de um VANT que fará parte de uma Rede de Sensores Sem Fio (RSSF), como nó sensor móvel.

A linguagem de descrição da missão define todas as ações que o VANT deverá realizar durante uma missão de observação de um fenômeno, assim como, define a missão em alto nível apenas com os objetivos a serem cumpridos. Esta linguagem chamada de Mission Description Language é baseada na especificação da SensorML.

Além disso, o VANT se comunicará com os sensores estáticos para obter as informações coletadas por estes e então tomar uma decisão para continuação da missão ou exigir intervenção humana. Neste trabalho, será desenvolvido também um algoritmo de tomada de decisão utilizando sistemas multiagentes para gerenciar a cooperabilidade entre VANTs, a negociação para realização das tarefas e a tomada de decisões a partir das informações recebidas. A interface de planejamento e controle da missão foi desenvolvida utilizando o conceito de Model, View and Controller (vide ANEXO A) para particionar os módulos da aplicação de maneira inequívoca e completa.

1.1 Motivação

A motivação deste projeto é a criação de um sistema para atribuir um comportamento autônomo a um veículo aéreo não-tripulado para que este possa realizar desde missões de coleta de dados até a substituição de veículos tripulados na realização de tarefas complexas e perigosas para seres humanos. A combinação de veículos aéreos não-tripulados e redes de sensores sem fio possui diversas aplicações, como por exemplo, agricultura de precisão e no monitoramento e manutenção preditiva de linhas de transmissão de energia elétrica.

1.2 Objetivos

Este projeto tem como objetivo o desenvolvimento de um descritor de missões em alto nível que gera um arquivo de missão como entrada para um sistema de controle de baixo nível que efetivamente realiza o controle de um VANT na realização de uma missão. A missão é descrita utilizando o formato XML e é interpretada pelo sistema de controle, que utiliza um algoritmo que atribui sub-missões aos agentes do sistema para atingir os objetivos da missão.

2 REDES DE SENSORES SEM FIO

Redes de sensores têm o objetivo de monitorar algum fenômeno. Este pode ser observado por sensores que recebem e respondem a sinais ou estímulos e são usados para analisar fenômenos e converter os dados obtidos em sinais eletrônicos para serem analisados. Uma rede de sensores é composta por sensores, um ou mais observadores e os fenômenos a serem observados. O sensor é a entidade que realiza a monitoração do fenômeno e transmite as informações ao observador. O fenômeno é qualquer acontecimento passível de observação, seja ele uma quantidade física mensurável, seres vivos ou inanimados. O observador é o usuário final que analisa os dados coletados. Uma rede de sensores pode ser composta por nós sensores estáticos e nós sensores móveis. Nós sensores móveis podem ser VANTs com sensores acoplados. Normalmente estas redes possuem um grande número de nodos distribuídos, possuem restriçõoes de energia, e devem possuir mecanismos para auto-configuração e adaptação devido a problemas como falhas de comunicação e perda de nodos. Uma RSSF se torna mais eficiente quando dispõe de autonomia na tomada de decisões e.g. quanto ao particionamento das tarefas, e requer um alto grau de cooperação para executar as tarefas definidas.

2.1 Áreas de aplicação

Redes de sensores podem ser homogêneas ou heterogêneas em relação aos tipos, dimensões e funcionalidades dos nodos sensores. Algumas áreas de aplicação são:

- **Controle:** Para prover um mecanismo de controle, seja em um ambiente industrial ou não. Por exemplo, sensores sem fio podem ser embutidos em "peças" numa linha de montagem para fazer testes no processo de manufatura.
- **Ambiente:** Para monitorar variáveis ambientais em locais internos como prédios e residências, e locais externos como florestas, desertos, oceanos, etc.
- **Tráfego:** Para monitorar tráfego de veículos em rodovias, malhas viárias urbanas, etc.
- **Segurança:** Para prover segurança em centros comerciais, estacionamentos, etc.

- **Medicina/Biologia:** Para monitorar o funcionamento de órgãos como o coração, detectar a presença de substâncias que indicam a presença ou surgimento de um problema biológico, seja no corpo humano ou animal.
- **Militar:** Para detectar movimentos inimigos, explosões, a presença de material perigoso como gás venenoso ou radiação, etc. Neste tipo de aplicação, os requisitos de segurança são fundamentais. O alcance das transmissões dos sensores é geralmente reduzido para evitar escutas clandestinas e reduzir o consumo de energia para se realizar as transmissões. Os dados são criptografados e submetidos a processos de assinatura digital. As dimensões variam conforme a aplicação específica e podem utilizar nodos sensores móveis como os transportados por robôs e VANTs.

De forma geral, uma RSSF pode ser utilizada em segurança e monitoramento, controle, manutenção de sistemas complexos, e monitoramento de ambientes internos e externos.

2.2 Tarefas típicas em uma RSSF

Uma RSSF necessita executar tarefas colaborativas. Geralmente os objetivos de uma RSSF dependem da aplicação, mas as seguintes tarefas são comumente utilizadas neste tipo de rede.

- **Determinar o valor de algum parâmetro num dado local:** Por exemplo, numa aplicação ambiental pode-se desejar saber qual é o valor da temperatura, pressão atmosférica, quantidade de luz e umidade relativa em diferentes locais.
- **Detectar a ocorrência de eventos de interesse e estimar valores de parâmetros em função do evento detectado:** Por exemplo, numa aplicação de tráfego pode-se desejar saber se há algum veículo trafegando num cruzamento e estimar a sua velocidade e direção.
- **Classificar um objeto detectado:** Por exemplo, ainda na aplicação de tráfego pode-se saber se o veículo é uma moto, um carro, um ônibus ou um caminhão.
- **Rastrear um objeto:** Por exemplo, numa aplicação biológica pode-se querer determinar a rota de migração de baleias.

Em todos estes exemplos, a utilização de mais de um nó sensor é necessária, seja para transmissão de dados, i.e. roteamento de mensagens, ou para determinar e gerar uma informação baseada nos dados observados por vários sensores.

2.3 Exemplos de aplicações de redes de sensores sem fio

Existem diversas aplicação de redes de sensores sem fio. A utilização e a aplicação de novos conhecimentos no meio rural auxiliam o produtor a identificar estratégias que possam aumentar a eficiência no gerenciamento da agricultura, maximizando a rentabilidade das colheitas e tornando o agronegócio mais competitivo (FREITAS, E. P.; WEHRMEISTER, M. A.; PEREIRA, C. E.; LARSSON, T. , 2008). A introdução do conceito de agricultura de precisão em propriedades onde se tem como objetivo maximizar a produtividade e minimizar os danos ambientais é imprescindível (ALONÇO, A. dos S. et al., 2005). Outra

aplicação a ser mencionada é o monitoramento de linhas aéreas de transmissão de energia elétrica. O método principal utilizado para diagnóstico de falhas é a inspeção visual que possui a vantagem de evitar o contato direto com a linha de transmissão. O uso de helicópteros tripulados é a solução comumente utilizada para monitoramento por imagens ao longo de toda a extensão da linha, a uma distância suficientemente próxima para captura de imagens com boa resolução. Esta abordagem, contudo, coloca em risco a segurança dos tripulantes da aeronave devido à proximidade aos condutores de alta tensão. Uma alternativa de menor custo e que evita colocar operadores em situação de risco é o uso de aeronaves não tripuladas, de dimensões reduzidas, mais baratas e devidamente instrumentadas. Estes equipamentos podem realizar interação com sensores colocados na extensão da linha de transmissão para facilmente detectar a localização exata do trecho onde há ocorrência de falha. Um VANT pode realizar o monitoramento de uma linha de transmissão de uma forma autônoma, voando ao longo da linha e analisando pontos de interesse, quando necessário. Os pontos da linha a serem monitorados podem ser definidos a partir de um plano de monitoramento enviado pelo operador antes ou durante o vôo. O VANT por sua vez se baseia no plano descrito pelo operador e nas informações fornecidas pelos sensores dispostos ao longo da linha para selecionar as áreas de interesse e focar a atividade de monitoramento. Estas duas aplicação são descritas para exemplificar as aplicações deste sistema, porém as aplicações deste projeto vão bem além das aplicações descritas.

2.4 Vantagens da utilização de VANTs como nós móveis em uma RSSF

A utilização de nós móveis em uma RSSF é interessante pois garante mobilidade aos sensores, o que aumenta a cobertura da aquisição de dados em uma área. Sensores estáticos observam apenas o fenômeno a sua volta, enquanto que sensores dinâmicos podem observar o fenômeno com maior proximidade e precisão. Além disso estes nós móveis podem receber os dados dos sensores estáticos e assim obter mais informações sobre o fenômeno que está sendo observado, ou basear-se nestes dados para deslocar-se para outra região onde pode ser mais útil. A utilização de VANTs se mostra ainda mais vantajosa pois estes veículos se movimentam no ar, podendo superar obstáculos com maior facilidade e liberdade aumentando assim o alcance da observação. As principais aplicações e vantagens são:

Aplicação em agricultura de precisão: Possibilita a coleta de dados de sensores sem fio, elimina a necessidade da utilização de veículos aéreos tripulados para fotografias aéreas e integra as imagens aéreas com as informações dos sensores de solo, agregando uma maior quantidade de dados para a tomada de decisões.

Aplicação em Monitoramento de Linha Aérea de Transmissão de Energia Elétrica: Possibilita a coleta de dados dos sensores da linha, realiza inspeções urgentes em caso de alarmes, elimina a necessidade de inspeção visual com veículos tripulados e reduz o risco de acidentes.

Aplicação em Monitoramento de Fronteiras: Reduz a necessidade de uso de pessoal em atividades monótonas e repetitivas altamente suscetíveis a erros e acidentes, além de reduzir os elevados custos operacionais em um país de grandes dimensões como o Brasil. Ademais, aeronaves convencionais são facilmente visualizadas, o que facilita a sua exposição a contramedidas que podem anular a sua atuação.

Aplicação de Apoio à Segurança Pública: Reduz a exposição do homem a atividades de alto risco, amplia o alcance restrito das medidas convencionais de prevenção à criminalidade e proporciona o monitoramento de áreas de difícil acesso.

Além das aplicações citadas, pode-se estender a utilização de redes de sensores sem fio em conjunto com veículos aéreos não-tripulados a qualquer aplicação que usufrua de sensores no solo que emitam alarmes quando captam um evento para que o VANT navegue até o local e adquira mais informações sobre a situação de interesse. Os sensores em solo podem ser simples com uma ou mais funções bem definidas, enquanto que os sensores embarcados na plataforma móvel podem ser mais sofisticados e assim aproveitar a mobilidade. Assim, pode-se ter sensores em solo em grande quantidade o que possibilita monitorar uma área com maiores proporções e quando necessário os sensores mais sofisticados embarcados no VANT podem se mover até o local de interesse. Um alarme é uma mensagem enviada pelo sensor estático ao VANT para que este navegue até o sensor para obter informações sobre um evento de interesse. Um exemplo de aplicação deste cenário é em monitoramento de fronteiras, onde coloca-se diversos sensores de detecção de movimento e quando emitido um alarme o VANT navega até o local e capta imagens em alta resolução da área de interesse.

3 VEÍCULOS AÉREOS NÃO-TRIPULADOS

Veículo aéreo não-tripulado é o temo usado para descrever todo e qualquer tipo de aeronave que não necessita de pilotos embarcados para ser guiada. Este tipo de aeronave é controlado à distância, por meios eletrônicos e computacionais, sob a supervisão humana. Inicialmente, o VANT foi idealizado para fins militares para ser utilizado em missões perigosas para serem executadas por humanos como nas áreas de inteligência militar, apoio e controle de tiro de artilharia, apoio aéreo às tropas de infantaria e cavalaria no campo de batalha, controle de mísseis de cruzeiro, atividades de patrulhamento urbano, costeiro, ambiental e de fronteiras, atividades de busca e resgate, entre outras. Com os avanços tecnológicos a intervenção humana se mostra cada vez menos necessárias. Para tanto, utiliza-se um sistema de controle que atribui missões aos veículos para que estes funcionem de maneira autônoma para atingir os seus objetivos.

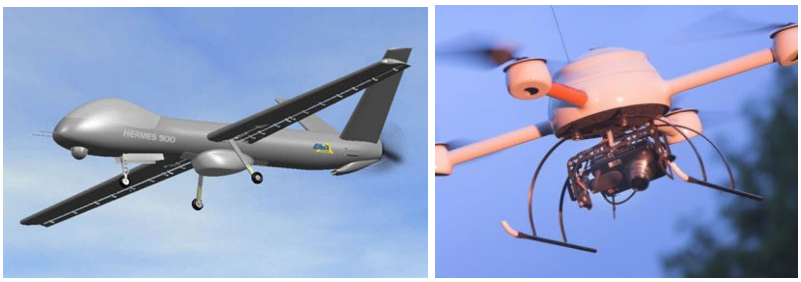


Figura 3.1: Exemplos de veículos aéreos não-tripulados.

3.1 Aplicações

Existem diversas aplicações para veículos aéreos não-tripulados, algumas delas são:

- Vigilância policial de áreas urbanas.
- Vigilância de áreas de fronteira.
- Inspeção de oleodutos e gasodutos.
- Controle de safras agrícolas.
- Levantamento de recursos florestais.
- Controle de queimadas.
- Enlace de comunicações e cobertura de eventos para as redes de TV.
- Detecção precoce e monitorização e apoio ao combate de incêndios em florestas.

- Vigilância da costa marítima, tendo em vista detectar e dissuadir contrabando e narcotráfico marítimo.
- Vigilância da orla costeira tendo em vista fiscalizar o tráfego de navegação comercial e promover a segurança marítima, fiscalizar a exploração e a extração de recursos naturais (com ênfase para a pesca e a aqüicultura oceânica).
- Vigilância aérea das vias rodoviárias para dissuadir, detectar e identificar eventuais pontos críticos, bem como para apoiar a gestão e o planejamento de infra-estruturas.
- Vigilância aérea e terrestre de fronteiras e sistemas de transportes para imigração ilegal.
- Definição e execução de planos de ordenamento do território e, em particular, das reservas naturais, patrimônio florestal (espécies e higiene) e ordenamento urbano.
- Meteorologia: estudos de uma grande gama de fenômenos naturais.
- Monitorização ambiental aérea (qualidade do ar).
- Serviços de vigilância e segurança pública e de instalações.
- Busca e salvamento.

3.2 Plataforma Skydrones

A plataforma da Skydrones (Skydrones) é uma plataforma aérea com sofisticada eletrônica embarcada que permite transportar diferentes sistemas para a observação de fenômenos. Sua instrumentação aviônica e sistemas de controle permitem vôos com alta estabilidade. Suas principais características são:

- Estabilização autônoma das atitudes em vôo da plataforma obtido pelo acionamento direto de quatro hélices e sistema de controle embarcado.
- Pouso e decolagem vertical permitindo uso em espaço restrito, necessitando apenas 1m² de área.
- Possibilidade de programação de vôo estacionário ou avanço em alta velocidade até pontos pré-determinados (GPS), por computador.
- Comando de retorno autônomo para a base operacional.
- Baixo peso da plataforma e alto potencial de carregamento (sensores e câmeras embarcados).
- Possibilidade de uso de câmeras especiais, como infra vermelhas (FLIR) e de alta resolução (HD) de foto e vídeo.
- Base de comando (em terra) com integração de dados de vôo, captura de imagem e cartografia.

Este trabalho foi desenvolvido para ser testado utilizando esta plataforma móvel, que é um pequeno quadricóptero com quatro hélices, equipado com um controle de vôo e navegação por GPS. O equipamento eletrônico embarcado foi desenvolvido pela

companhia alemã Mikrokopter (Mikrokopter) e adaptado pela companhia brasileira Skydrones. A figura 3.2 apresenta uma foto ilustrativa da plataforma.

Figura 3.2: Quadricóptero e controle remoto.

A plataforma pesa 850 gramas e sua dimensão é definida pelas quatro barras de suporte nas quais as hélices estão fixadas. Cada barra tem 20 centímetros de comprimento. A plataforma é equipada com cinco sensores: um giroscópio para cada hélice, uma bússola, um acelerômetro de três eixos, um GPS e um sensor de altitude e pressão. O sistem de controle de vôo é executado em um Atmel ATMEGA644 - microcontrolador de 20MHz, enquanto que o controle de navegação é executado em um processador ARM-9. Ele usa uma bateria LiPO 5000 mAh que é a fonte de energia para as hélices e os dispositivos eletrônicos, oferecendo uma duração de 15 a 20 minutos de vôo. Um sistema de rádio de 6 canais é usado como controle remoto.

4 SENSORML

Uma forma de se configurar redes de sensores é utilizando-se uma linguagem de markup como a SensorML (OPEN GEOSPATIAL CONSORTIUM INC.). A SensorML especifica modelos padrão e uma codificação XML para descrever qualquer processo, incluindo processos de medida realizados por sensores e instruções para obter informações a partir das observações. Processos descritos em SensorML são executáveis e aprendem a partir da experiência obtida com as observações. Todos os processos têm definidos suas entradas, saídas, parâmetros e métodos. Esta linguagem modela detectores e sensores como processos para converter fenômenos reais em dados. A principais funções da SensorML são:

- Prover descrições de sensores e sistemas de sensores para gerenciamento de inventário.
- Prover informações sobre sensores e processos para dar suporte em obervações e descoberta de recursos.
- Prover suporte ao processo e análise das observações realizadas por sensores.
- Prover suporte à localização geoespacial de valores observáveis (medida de dados).
- Prover características de performance (i.e., precisão, limites, etc.).
- Prover uma descrição explícita de processos obtidos de observações.
- Prover uma cadeia de processos executáveis para gerar novos dados sob demanda.
- Prover um sistema de arquivamento das propriedades e hipóteses sobre sistemas de sensores.

De uma forma geral, a SensorML fornece ferramentas de descrição de qualquer processo e cadeia de processos, entretanto é particularmente melhor aplicada para a descrição de sensores, sistemas e processamento das observações realizadas. Sensores e transdutores (detectores, transmissores, atuadores e filtros) são modelados como processos que podem ser conectados e participar de forma igualitária dentro de uma cadeia de processos ou sistema utilizando o mesmo modelo de processo como qualquer outro processo.

Processos são entidades que têm uma ou mais entradas e através da aplicação de métodos bem definidos usando parâmetros específicos resultam em uma ou mais saídas. O modelo de processo definido na SensorML pode ser usado para descrever uma ampla variedade de processos, podemos citar como exemplo atuadores, transformadas espaciais e processos de dados. SensorML também suporta a conexão entre processos e,

portanto, suporta o conceito de cadeia de processos, que são também definidos como processos.

A SensorML fornece ferramentas para a definição das características geométricas, dinâmicas e observacionais de sensores e sistemas de sensores. Há uma grande variedade de tipos de sensores, desde simples termômetros até microscópios eletrônicos complexos e satélites de observação da Terra. Estes podem ser suportados através da definição de modelos de processos atômicos e cadeias de processos.

As observações e medidas de dados devem respeitar três características fundamentais:

- As propriedades de entidades físicas e fênomenos devem ser possíveis de serem medidos e quantificados. Uma propriedade que é possível de ser medida é denominada como “Propriedade observável” ou “Fênomeno”. Por exemplo: temperatura, concentração química, emissividade de radiação, etc.
- Deve existir sensores capazes de observar e medir uma propriedade particular. Estes sensores possuem respostas características que são usadas para determinar os valores das medidas e a qualidade das medidas realizadas. Além disso, estes sensores possuem propriedades de localização e orientação que permitem a associação dos valores medidos com uma localização geoespacial específica.
- Devem existir dados que são retornados por um sistema de sensores ou derivados das medidas realizadas pelos sensores. Estas medidas podem ser acessadas diretamente pelo sensor, ou por centrais de armazenamento de dados.

A SensorML geralmente não fornece uma descrição detalhada do hardware de um sensor, entretanto pode ser utilizada para este fim. Preferencialmente, é utilizada para descrever um esquemático geral e funcional dos sensores. A especificação de um sensor suporta o processamento e geolocalização dos dados de praticamente qualquer sensor, seja ele móvel ou dinâmico, fixo ou sensível remotamente, ativo ou passivo. Isto permite o desenvolvimento de software de aplicação geral que pode processar e geolocalizar os dados de uma grande variedade de sensores, indo desde simples sensores até complexos sistemas de sensores.

4.1 Componentes essenciais de SensorML

Componente – Processo atômico físico que transforma um tipo de informação em outro. Por exemplo, um detector tipicamente transforma uma propriedade ou fenômeno físico observável em um número digital. Exemplo de componentes incluem detectores, atuadores e filtros.

Sistema – Composto por modelos físicos de um grupo ou matriz de componentes. Estes componentes podem ser detectores, atuadores ou subsistemas. Um sistema está relacionado com um processo no mundo real e fornece definições adicionais de acordo com as posições relativas de seus componentes e suas interfaces de comunicação.

Modelo de processo – Bloco atômico não-físico de processamento usado normalmente numa cadeia mais complexa de processos. É associado com o método de processo que define a interface dos processos, e define como executar o modelo. Ele também descreve precisamente suas próprias entradas, saídas e parâmetros.

Cadeia de processos – Bloco de componente não-físico de processamento que consiste de sub-processos interconectados, os quais podem ser modelos de processo ou cadeia de processos. Uma cadeia de processos também inclui possíveis fontes de dados, assim como conexões que ligam explicitamente sinais de entrada e saída de sub-processos. Define precisamente suas próprias entradas, saídas e parâmetros.

Método de processo – Define o comportamento e a interface de um modelo de processo. Pode ser armazenado em uma biblioteca para que possa ser reutilizado por diferentes instâncias de modelos de processo. Ele descreve em essencial a interface do processo e o algoritmo.

Detector – Perfil de modelo de processo que representa um componente atômico de um sistema de medidas definindo amostragem e resposta característica de um dispositivo simples de detecção. Um detector tem apenas uma entrada e uma saída, ambas são quantidades escalares. Sensores mais complexos como o de uma câmera que é composto por múltiplos detectores pode ser descrito como um grupo ou matriz de detectores usando um sistema ou sensor. Em SensorML, um detector é um tipo particular de modelo de processo.

Sensor – Tipo específico de sistema de representação completa de sensores. Por exemplo, um scanner completo de veículo aéreo, o que inclui os diversos detectores (um para cada banda).

4.2 Aplicações de SensorML

Folha de especificação eletrônica – pode ser usado como padrão em meio digital para descrever a especificação de componentes de sensores e sistemas.

Descoberta de sensores, sistemas de sensores e processos – SensorML é o meio pelo qual sistemas de sensores ou processos podem se tornar conhecidos e descobríveis. SensorML disponibiliza uma rica coleção de metadados que pode ser extraída e utilizada para descobrir sistemas de sensores e processos de observação. Estes metadados incluem: identificadores, classificadores, restrições (tempo, leis e segurança), capacidades, características, contatos e referências em adição às entradas, saídas, parâmetros e localização do sistema.

Linhagem de observações – SensorML disponibiliza uma descrição completa e sem ambiguidades da linhagem de uma observação. Em outras palavras, pode descrever em detalhes o processo no qual uma observação foi feita pela aquisição por um ou mais detectores para ser processado e talvez até interpretado por um analista. Isto fornece um nível de confidencialidade em se tratando da observação. Na maioria dos casos, parte ou todos os processos podem ser repetidos, possivelmente com algumas modificações no processo ou por simulação da observação com uma fonte de assinatura conhecida.

Processamento de observações sob demanda – Cadeia de processos para geolocalização ou processamento de alto nível de observações podem ser descritos em SensorML, distribuídos pela internet e executados sob demanda sem um conhecimento a priori do sensor ou processo característico. Esta foi a motivação original para SensorML, como meio de combater a proliferação de sistemas distintos e afunilados para processamento de dados de sensores entre diversas comunidades de sensores. SensorML também permite a distribuição do processamento para qualquer ponto de uma cadeia de sensores, de um sensor para a central de dados para o usuário utilizando

um PDA. SensorML habilita este processamento sem a necessidade de um software de sensor específico.

Suporte para tarefa, observação e serviços de alerta – A descrição de sistemas de sensores ou simulações em SensorML pode ser utilizada em suporte no estabelecimento de Sensor Observation Services (SOS), Sensor Planning Services (SPS), e Sensor Alert Services (SAS). SensorML define dados comuns que são usados através da estação de trabalho OGC Sensor Web Enablement (SWE) (OPEN GEOSPATIAL CONSORTIUM INC.).

Plug and Play, auto reconfiguração e redes de sensores autônomas – SensorML permite o desenvolvimento de sensores plug and play, simulações e processos, os quais podem ser adicionados ao sistema de suporte de decisões. As características de auto descrição da SensorML permitem que sensores e processos também suportem o desenvolvimento de auto configuração de redes de sensores, assim como o desenvolvimento de redes de sensores autônomas as quais podem publicar alertas e tarefas para os quais outros sensores podem subscrever e reagir.

Arquivamento dos parâmetros do sensor – SensorML disponibiliza um mecanismo de arquivamento de parâmetros fundamentais em se tratando de sensores e processos, de modo que observações provenientes destes sistemas podem ainda ser reprocessadas e melhoradas muito após a missão de origem terminar. Isto se mostra crítico para aplicações de longo alcance como monitoração e modelagem de mudanças globais.

4.3 Exemplo de descrição utilizando SensorML

Um exemplo de aplicação da SensorML é sua utilização para descrever uma estação de monitoramento de condições climáticas. Esta estação é modelada como um sistema onde os diferentes dispositivos sensores da estação são classificados como componentes.

Keywords

A seção de palavras-chave contém os termos que descrevem as condições climátivas.

Listing 4.1: Exemplo de definição de palavras-chave em SensorML.

```
1 <keywords>
2 <KeywordList>
3 <keyword>weather station</keyword>
4 <keyword>precipitation</keyword>
5 <keyword>wind speed</keyword>
6 <keyword>temperature</keyword>
7 </KeywordList>
8 </keywords>
```

Identification

Neste exemplo três identificadores diferentes são criados: um identificador único, um nome longo e um nome pequeno para descrever as estações.

Listing 4.2: Exemplo de definição de identificadores de um sistema em SensorML.

```
<identification>
<IdentifierList>
<identifier name="uniqueID">
<Term definition="urn:ogc:def:identifier:OGC:uniqueID">
<value>urn:ogc:object:feature:Sensor:IFGI:weatherStation123</value>
</Term>
</identifier>
<identifier name="longName">
<Term definition="urn:ogc:def:identifier:OGC:1.0:longName">
<value>OSIRIS weather station 123 on top of the IfGI building</value>
</Term>
</identifier>
<identifier name="shortName">
<Term definition="urn:ogc:def:identifier:OGC:1.0:shortName">
<value>OSIRIS Weather Station 123</value>
</Term>
</identifier>
</IdentifierList>
</identification>
```

Classification

A seção de classificação fornece informações sobre qual a aplicação dos sensores, neste caso para monitorar condições climáticas.

Listing 4.3: Exemplo de definição de classificação de um sistema em SensorML.

```
<classification>
<ClassifierList>
<classifier name="intendedApplication">
<Term definition="urn:ogc:def:classifier:OGC:1.0:application">
<value>weather</value>
</Term>
</classifier>
</ClassifierList>
</classification>
```

Valid Time

Neste exemplo, a descrição do tempo do sensor é válida apenas por um certo período de tempo.

Listing 4.4: Exemplo de definição de período de tempo de validade de um sistema em SensorML.

```
<validTime>
<gml:TimePeriod>
<gml:beginPosition>2009-01-15</gml:beginPosition>
<gml:endPosition>2009-01-20</gml:endPosition>
</gml:TimePeriod>
</validTime>
```

Capabilities

A seção de capacidades deve conter dois itens principais. Em primeiro lugar, o estado da estação meteorológica é descrito. Neste exemplo, o estado é “ativo”. Em seguida, é delimitado a área que será observada pela estação. Para esta estação (um sensor in-situ), é determinado apenas um único ponto com a localização da estação.

Listing 4.5: Exemplo da definição das capacidades de um sistema em SensorML.

```
<capabilities>
<swe:DataRecord definition="urn:ogc:def:property:capabilities">
<swe:field name="status">
<swe:Text definition="urn:ogc:def:property:OGC:1.0:status">
<gml:description>The operating status of the system.</gml:description>
<swe:value>active</swe:value>
</swe:Text>
</swe:field>
<swe:field name="observedBBOX">
<swe:Envelope
definition="urn:ogc:def:property:OGC:1.0:observedBBOX">
<swe:lowerCorner>
<swe:Vector>
<swe:coordinate name="easting">
<swe:Quantity axisID="x">
<swe:uom code="m"/>
<swe:value>2592308.332</swe:value>
</swe:Quantity>
</swe:coordinate>
<swe:coordinate name="northing">
<swe:Quantity axisID="y">
<swe:uom code="m"/>
<swe:value>5659592.542</swe:value>
</swe:Quantity>
</swe:coordinate>
</swe:Vector>
</swe:lowerCorner>
```

```
27 <swe:upperCorner>
28 <swe:Vector>
29 <swe:coordinate name="easting">
30 <swe:Quantity axisID="x">
31 <swe:uom code="m"/>
32 <swe:value>2592308.332</swe:value>
33 </swe:Quantity>
34 </swe:coordinate>
35 <swe:coordinate name="northing">
36 <swe:Quantity axisID="y">
37 <swe:uom code="m"/>
38 <swe:value>5659592.542</swe:value>
39 </swe:Quantity>
40 </swe:coordinate>
41 </swe:Vector>
42 </swe:upperCorner>
43 </swe:Envelope>
44 </swe:field>
45 </swe:DataRecord>
46 </capabilities>
```

Contact

A seção de contato neste exemplo determina o responsável pela operação da estação. Neste caso, um grupo de pesquisa da universidade de Münster.

Listing 4.6: Exemplo da definição de contatos de um sistema em SensorML.

```
1  <contact>
2  <ResponsibleParty gml:id="WWU_IfGI_weather_station_contact">
   <organizationName>Westfälische Wilhelms-Universität Münster - Sensor
3  Web and Simulation Lab</organizationName>
4  <contactInfo>
5  <address>
   <electronicMailAddress>swsl-ifgi@listserv.uni-
6  muenster.de</electronicMailAddress>
7  </address>
8  </contactInfo>
9  </ResponsibleParty>
10 </contact>
```

Position

Determina a posição do sistema, neste caso é identica a área de observação do sensor.

Listing 4.7: Exemplo da definição da posição de um sistema em SensorML.

```
<position name="systemPosition">
<swe:Position referenceFrame="urn:ogc:def:crs:EPSG:6.14:31466">
<swe:location>
<swe:Vector gml:id="SYSTEM_LOCATION">
<swe:coordinate name="easting">
<swe:Quantity axisID="x">
<swe:uom code="m"/>
<swe:value>2592308.332</swe:value>
</swe:Quantity>
</swe:coordinate>
<swe:coordinate name="northing">
<swe:Quantity axisID="y">
<swe:uom code="m"/>
<swe:value>5659592.542</swe:value>
</swe:Quantity>
</swe:coordinate>
<swe:coordinate name="altitude">
<swe:Quantity axisID="z">
<swe:uom code="m"/>
<swe:value>297.0</swe:value>
</swe:Quantity>
</swe:coordinate>
</swe:Vector>
</swe:location>
</swe:Position>
</position>
```

Inputs

As entradas são os fenômenos observados pela estação. Neste caso: precipitação, vento e temperatura.

Listing 4.8: Exemplo da definição das entradas de um sistema em SensorML.

```
<inputs>
<InputList>
<input name="precipitation">
<swe:ObservableProperty
definition="urn:ogc:def:property:OGC:1.0:precipitation"/>
</input>
<input name="wind">
```

```
    <swe:ObservableProperty
 7  definition="urn:ogc:def:property:OGC:1.0:wind"/>
 8  </input>
 9  <input name="atmosphericTemperature">
    <swe:ObservableProperty
10  definition="urn:ogc:def:property:OGC:1.0:temperature"/>
11  </input>
12  </InputList>
13  </inputs>
```

Outputs

As entradas deste exemplo geram quatro tipos de saídas que são fornecidas pelos sensores da estação meteorológica: precipitação, direção do vento, velocidade do vento e temperatura.

Listing 4.9: Exemplo da definição das saídas de um sistema em SensorML.

```
 1  <outputs>
 2  <OutputList>
 3  <output name="precipitation">
 4  <swe:Quantity definition="urn:ogc:def:property:OGC:1.0:precipitation">
 5  <swe:uom code="mm"/>
 6  </swe:Quantity>
 7  </output>
 8  <output name="windDirection">
 9  <swe:Quantity definition="urn:ogc:def:property:OGC:1.0:windDirection">
10  <swe:uom code="deg"/>
11  </swe:Quantity>
12  </output>
13  <output name="windSpeed">
14  <swe:Quantity definition="urn:ogc:def:property:OGC:1.0:windSpeed">
15  <swe:uom code="m/s"/>
16  </swe:Quantity>
17  </output>
18  <output name="temperature">
19  <swe:Quantity definition="urn:ogc:def:property:OGC:1.0:temperature">
20  <swe:uom code="Cel"/>
21  </swe:Quantity>
22  </output>
23  </OutputList>
24  </outputs>
```

Components

Nesta seção, os três tipos de sensores da estação são descritos em detalhes. O termomêtro é descrito diretamente utilizando-se a SensorML, enquanto que a descrição do pluviômetro e anemômetro são descritos com referências a links externos.

Listing 4.10: Exemplo da definição de componentes de um sistema em SensorML.

```
<components>
<ComponentList>
<component name="rainGauge"
xlink:href="http://mySensorMLregistry.com?object=98765"/>
<component name="anemoneter"
xlink:href="http://mySensorMLregistry.com?object=33333"/>
<component name="thermometer">
<Component>
(Descrição detalhada na seção 4.3.11)
</Component>
</component>
</ComponentList>
</components>
```

Component

Nesta seção a descrição detalhada do termômetro é especificada. Esta descrição se assemelha a descrição do sistema. Contudo, a seção de contatos e posição não são repetidas, em vez disso, estas informações são herdadas da descrição da estação meteorológica.

Listing 4.11: Exemplo da definição de um componente descrito em SensorML

```
<Component>
<keywords>
<KeywordList>
<keyword>weather station</keyword>
<keyword>temperature</keyword>
</KeywordList>
</keywords>
<identification>
<IdentifierList>
<identifier name="uniqueID">
<Term definition="urn:ogc:def:identifier:OGC:uniqueID">
<value>urn:ogc:object:feature:Sensor:IFGI:thermometer123</value>
</Term>
```

```
14 </identifier>
15 <identifier name="longName">
16 <Term definition="urn:ogc:def:identifier:OGC:1.0:longName">
17 <value>OSIRIS Thermometer at weather station 123</value>
18 </Term>
19 </identifier>
20 <identifier name="shortName">
21 <Term definition="urn:ogc:def:identifier:OGC:1.0:shortName">
22 <value>OSIRIS Thermometer 123</value>
23 </Term>
24 </identifier>
25 </IdentifierList>
26 </identification>
27 <classification>
28 <ClassifierList>
29 <classifier name="sensorType">
30 <Term definition="urn:ogc:def:classifier:OGC:1.0:sensorType">
31 <value>thermometer</value>
32 </Term>
33 </classifier>
34 </ClassifierList>
35 </classification>
36 <capabilities>
37 <swe:DataRecord definition="urn:ogc:def:property:capabilities">
38 <swe:field name="status">
39 <swe:Text definition="urn:ogc:def:property:OGC:1.0:status">
40 <gml:description>The operating status of the system.</gml:description>
41 <swe:value>active</swe:value>
42 </swe:Text>
43 </swe:field>
44 </swe:DataRecord>
45 </capabilities>
46 <inputs>
47 <InputList>
48 <input name="atmosphericTemperature">
    <swe:ObservableProperty
49 definition="urn:ogc:def:property:OGC:1.0:temperature"/>
50 </input>
51 </InputList>
```

```
52 </inputs>
53 <outputs>
54 <OutputList>
55 <output name="temperature">
56 <swe:Quantity definition="urn:ogc:def:property:OGC:1.0:temperature">
   <gml:groupName codeSpace="ObservationOffering"> Weather
57 </gml:groupName>
58 <swe:uom code="Cel"/>
59 </swe:Quantity>
60 </output>
61 </OutputList>
62 </outputs>
63 </Component>
```

5 MISSION DESCRIPTION LANGUAGE

Neste projeto explorou-se o conceito de atribuir-se missões aos sensores via uma linguagem de markup inspirada na SensorML, que é designada Mission Description Language (MDL). A MDL é uma linguagem de descrição de missões que utiliza o formato XML. Esta linguagem tem por objetivo descrever todas as etapas de uma missão a serem realizadas por um VANT e sua colaboração com outros sensores do sistema. Os veículos aéreos não-tripulados correspondem aos nós sensores móveis em uma rede de sensores sem fio. O veículo é equipado com sistema de posicionamento global (GPS), sistema de rádio para controle e coleta de dados e câmera de vídeo, além de sensores para aquisição de dados meteorológicos, tais como sensores de temperatura, higrômetros, barômetros ou anemômetros. A especificação das tarefas é feita através de pontos descritos por coordenadas geográficas dados por um sistema de posicionamento global e um comando a ser executado numa determinada posição. A missão é baseada no conceito de *waypoints*, onde todas as informações necessárias para o VANT realizar seu deslocamento e obtenção de dados é descrita. A interface foi desenvolvida em inglês, pois é um idioma de maior alcance científico e portante expande a sua utilização.

5.1 Informações gerais sobre a missão

Uma missão tem os seguintes dados básicos em sua descrição. Nome, descrição sucinta da missão, tarefas a serem realizadas para acompanhamento do usuário, tempo de começo e fim da missão, prioridade, tipo de sensores permitidos, tipo da missão, duração, sigilo da missão e o espaço de cobertura dado em coordenadas geoespaciais.

Name

Este campo define o nome do arquivo o qual será utilizado para salvar a missão em formato XML.

Description

Este campo é destinado à descrição em maiores detalhes da missão para acompanhamento do usuário e possível re-utilização do arquivo da missão caso sejam necessários informações específicas sobre a missão.

Tasks

Este campo define as principais tarefas a serem realizadas na missão.

Latitude, longitude and altitude boundaries

Estes campos definem a área de cobertura da missão para que o VANT não saia da área da missão durante um trajeto entre *waypoints* ou para atender um alarme recebido

de um nó estático. A área de cobertura é delimitada por: latitude norte, sul, leste e oeste; longitude norte, sul, leste e oeste; altitude mínima e máxima.

Priority

Este campo determina a prioridade da missão em caso de outras missões estarem sendo executadas na mesma área de cobertura.

Type of sensor allowed

Este campo é importante pois define o tipo de sensor a ser selecionado para uma missão. Por exemplo, radar ou sonar são sensores ativos pois emitem uma onda e recebem o eco das ondas emitidas, enquanto que uma camêra é um sensor passivo. Sensores ativos são sensores que transmitem algum tipo de energia (microondas, som, luz, etc.) no ambiente para detectar mudanças ocorridas com a energia transmitida e captada. Sensores passivos não transmitem energia e apenas detectam a energia transmitida por alguma fonte.

Type of the mission

Este campo define o tipo da missão que o VANT irá executar. A depender do tipo da missão pode-se escolher o melhor veículo para executá-la. Por exemplo, uma missão para tirar fotos com maior precisão observa-se que a utilização de um quadcóptero é mais adequada, pois este veículo pode parar no ar para tirar as fotos.

Endurance

Este campo caracteriza a autonomia da missão, pois determina quanto tempo a missão terá de duração e portanto pode-se determinar a necessidade de recarga ou abastecimento do veículo.

Secrecy

Este campo determina o sigilo da missão. Por exemplo, em caso de uma missão secreta pode-se determinar a utilização de veículos de menores dimensões e camuflados para não serem detectados.

5.2 Informações sobre o ambiente

As informações sobre o ambiente no qual a missão irá transcorrer são importantes para se determinar os tipos de sensores a serem utilizados e quais veículos são mais adequados. Além disso, determina alguns fatores a serem levados em consideração durante o vôo. Por exemplo, o ambiente for do tipo urbano, deve-se levar em consideração possíveis obstáculos e interferência externa.

Weather

Determina as condições meteorológicas às quais a missão estará submetida. Por exemplo, se o tempo for determinado chuvoso, então o veículo pode ser preparado para suportar a chuva e os sensores protegidos contra água. Também determina o tipo de sensor a ser utilizado.

Time of the day

Este campo determina se a missão será realizada à noite ou de dia. Isto determina qual tipo de sensor poderá ser utilizado. Por exemplo, se for à noite a utilização de uma câmera infravermelha é mais adequada.

Type

Determina o tipo de ambiente no qual a missão transcorrerá. Este campo também é utilizado para se determinar quais sensores e veículos são mais adequados para realizar a missão.

5.3 Sistema de descrição da posição

O sistema de posição é vital para a descrição da missão. Diferentemente de muitas aplicações geoespaciais que apenas requerem conhecimento sobre a localização, o processamento de dados dos sensores geralmente necessita um conhecimento mais complexo sobre o estado completo dos componentes do sistema. Além disso, o estado de vários componentes de um sistema de sensores é geralmente muito dinâmico, alterando sua posição e orientação com o tempo. O sistema de posição é determinado pelos *waypoints* que compõem uma missão. Cada waypoint possui: nome, descrição, prioridade, latitude, longitude, altitude, velocidade linear, tempo de permanência e o comando a ser executado. Os dados sobre a posição podem ser usados como entrada ou parâmetro para outros VANTs e sensores.

Name

Este campo determina a identificação do *waypoint* e portanto deve ser unívoca para uma determinada missão.

Description

Este campo é utilizado para descrever o que será feito no *waypoint* para controle do usuário.

Priority

O campo de prioridade determina a prioridade que este *waypoint* tem para a missão global. É utilizado para se determinar se um *waypoint* deve ser atendido antes de outro, mas depende da política utilizada para determinação da missão pelo sistema de controle.

Latitude, longitude and altitude

A localização do *waypoint* é dada pelo sistema de posicionamento global que é descrito pela latitude, longitude e altitude no *waypoint*.

Speed

Este campo determina a velocidade que o VANT deverá alcançar para chegar neste *waypoint*, normalmente este campo é utilizado para controle e não necessariamente para determinar a velocidade do VANT, já que o veículo funcionará de forma autônoma para atingir os objetivos da missão.

Time

Este campo determina o tempo de permanência do veículo no *waypoint* para realizar o comando determinado.

5.4 Comando

O comando determina a ação que o VANT deverá executar no *waypoint*. Para entender melhor como é descrito um comando, é necessário entender alguns conceitos.

5.4.1 Fenômeno

Um fenômeno pode ser uma propriedade física (como temperatura, tamanho, etc.), uma classificação (como espécies), frequência ou quantidade. Para suportar usos comuns em ciências naturais e engenharia é necessário um sistema que suporte a descrição de fenômenos derivados, ou seja, fenômenos compostos por mais de uma condição. Na MDL, o valor da propriedade observada é importante para classificar a informação reportada em uma observação. Isto requer uma definição de um fenômeno fundamental, que será usado como base para os fenômenos derivados. O fenômeno fundamental é composto por uma condição principal. Para suprir as possíveis necessidades de observação de um fenômeno existem duas especializações suportadas: o fenômeno simples e o composto.

O fenômeno simples modifica o fenômeno fundamental em se adicionando condições simples, cada uma especificando uma propriedade. Por exemplo, a "temperatura da água" possui como base a "temperatura" (i.e. é um tipo de temperatura) que tem como condição a propriedade da "substância" que é a "água", então o valor será a temperatura da água observada. Se por exemplo, o fenômeno for "temperatura da superfície da água" adiciona-se uma nova condição "profundidade" que pode ser "entre 0 e 0.3m".

O fenômeno composto possui vários componentes indicados pela dimensão. Ele é composto por um conjunto de fenômenos componentes. Os componentes não estão relacionados uns com os outros, entretanto um fenômeno composto deve ter uma certa coerência semântica. O fenômeno base permite que os fenômenos compostos sejam gerados adicionando-se componentes à base. Outra possibilidade é a adição de uma lista de condições ao fenômeno base, cada um determinando um conjunto de valores para um fenômeno único. Por exemplo, o "espectro da radiação" tem como base a "radiação" que possui uma lista de "comprimento de onda" especificada.

5.4.2 Propriedade observável

Uma propriedade observável é a propriedade de um fenômeno que pode ser observado e medido, Exemplos: temperatura, força gravitacional, número de indivíduos, etc. A propriedade observada é o valor do fenômeno observado.

5.4.3 Pattern

Um padrão (pattern) é uma definição de algum fenômeno ou objeto de forma a caracterizar o mesmo padrão para observação. Por exemplo, pode-se definir uma *pattern* "fire", para detectar que um fenômeno pode ser caracterizado como fogo é necessário que os sensores definam um intervalo de valores de medidas como temperatura, humidade e luminosidade os quais podem indicar a presença de fogo. A MDL é uma linguagem de descrição em alto nível, portanto a forma como os sensores efetivamente detectarão uma *pattern* específico foge ao escopo deste trabalho. O objetivo é prover uma ferramenta que possibilite tal tipo de definição, mas sem a preocupação de se apresentar as definições efetivamente.

5.4.4 Composição de um comando

Um comando pode ser composto por uma expressão. Uma expressão é formada por um conector, um comando e uma *pattern* ou fenômeno. Os conectores são utilizados para unir comandos. Uma *pattern* pode ser escolhida de uma lista de pattern pré-

definidas. Um fenômeno é composto pela característica principal, características secundárias, lista de caracteristicas e uma condição. Por exemplo, deseja-se definir um fenômeno temperatura da superfície da água, a característica principal é a temperatura e a secundária é superfície. O sensor embarcado possui a função de medir a temperatura da água, e portanto só define-se a característica de como a temperatura da água será medida.

Conectores

São possíveis a utilização dos seguintes conectores para formar um comando: IF THEN, WHILE DO, AND, OR, WHEN THEN e IN.

5.4.5 Comandos

SCAN

Este comando determina a procura por uma determinada *pattern*.

SCAN <HOUSE>: este comando procura por uma casa, a *pattern* casa foi definida pela arquitetura de baixo nível para identificar uma construção.

MONITOR

Este é um comando de monitoramento que pode ser utilizado em uma *pattern* ou um fenômeno.

MONITOR <MOVEMENT>: monitora movimento no *waypoint*.

MONITOR <TEMPERATURE>: monitora a temperatura.

SURVEY

Este comando determina a vigilância de uma *pattern*. Esta pattern pode ser definida como um objeto.

SURVEY <CAR>: efetua a vigilância de um carro.

DETECT

Este comando detecta a ocorrência de uma *pattern* ou fenômero.

IF **DETECT** <LIGHT> **IDENTIFY** <SOURCE>: se detectada luz identifica a fonte.

IDENTIFY

Este commando identifica uma diferença entre *patterns* ou fenômenos de acordo com uma definição de uma *pattern*.

IDENTIFY <TRUCK> IN **DETECT** <MOVEMENT>: identifica um caminhão emu ma região em que está ocorrendo movimento.

LOCALIZE

Localiza um determinado fenômeno ou *pattern*.

IF **IDENTIFY** <FIRE> THEN **LOCALIZE**: se uma *pattern* determinada como fogo é identificada então fornece sua localização.

VISUALIZE

Este commando fornece imagens de uma *pattern* ou fenômeno de interesse.

WHEN **LOCALIZE** <FIRE> THEN **VISUALIZE**: quando localizado a *pattern* fogo então fornece imagens.

ZOOM

Este commando determina que a câmera embarcada foque determinada *pattern* ou fenômeno de interesse.

WHEN **DETECT** <MOVIMENT> THEN **ZOOM<**OBJECT**>:** quando detectado movimento então foca o objeto que está em movimento.

MADE OF

Fornece informações sobre o material que constitui determinada *pattern* de interesse.

IF **IDENTIFY** <TRUCK> THEN **MADEOF:** se identificado a *pattern* TRUCK então determina informações sobre o material que constitui o caminhão.

5.5 Sistema de descrição dos sensores estáticos

A definição de um sensor estático é similar a definição de um waypoint. Ele é composto por um identificador unívoco, uma descrição, latitute, longitude e um comando. Os comandos vistos na seção anterior também são definidos para sensores estáticos. Os sensores estáticos são sensores instalados na área de cobertura da missão para realizar determinado comando. Durante uma missão, estes sensores podem enviar alarmes para sinalizar um VANT para ir até ele e coletar os dados.

5.6 Arquitetura do sistema

A arquitetura do sistema foi construída utilizando o conceito de MVC – Model, View amd Controller (vide ANEXO A). Os dados são armazenados em um package denominado *data*, os componentes da interface gráfica estão no package *gui*, e por fim o package *kernel* é o responsável por realizar a ligação entre a interface gráfica onde ocorre as iterações com o usuário e os dados. As entradas do usuário são definidas na interface gráfica, e quando o usuário aciona a funcionalidade para gerar uma missão estes dados são transferidos para o kernel que armazena estes dados e gera o arquivo em formato xml que descreve a missão.

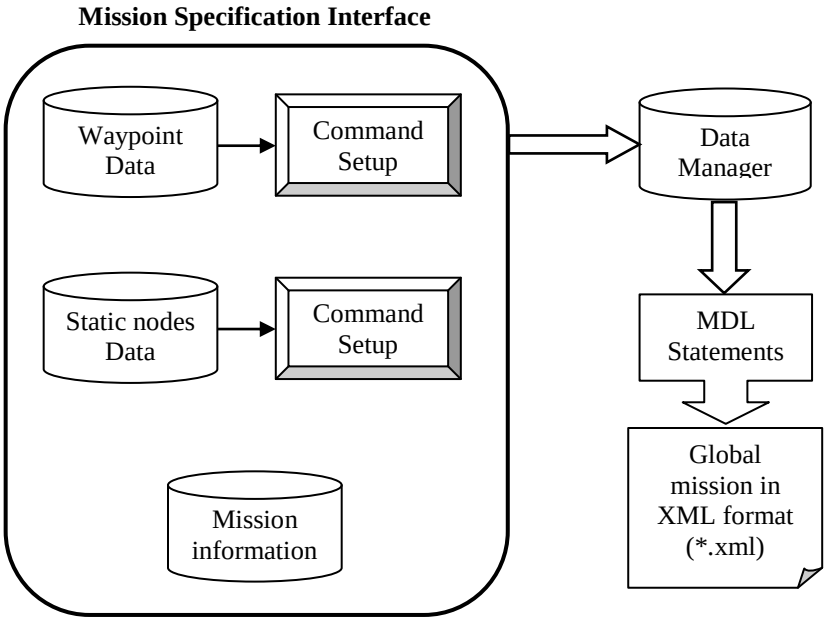


Figura 5.1: Arquitetura do gerador de missões da MDL.

5.6.1 Dados de entrada

Os dados de entrada principais são a descrição da missão por parte do usuário através da interface gráfica, onde é possível definir todos os dados da missão em alto nível, os *waypoints* e os nós sensores estáticos. Algumas opões do programa são sintetizadas através da utilização de arquivos com a extensão .dat presentes no diretório do programa. Estes arquivos são detalhados a seguir.

- commands.dat: descreve os comandos utilizados para compor uma expressão que será passada para a interface de baixo nível que efetivamente executa o comando. Os comandos podem ser: SCAN, MONITOR, SURVEY, DETECT, IDENTIFY, LOCALIZE, VISUALIZE, ZOOM e MADEOF.
- conditions.dat: descreve condições a serem cumpridas para execução de um comando. Os condicionais podem ser: INCREASE OF, DECREASE OF, EQUAL, LESS THAN, GREATER THAN.
- connectors.dat: descreve os conectores disponíveis para se compor um comando que será executado no *waypoints* e pelos nós sensores estáticos. Os conectores podem ser: IF, WHILE, WHEN, THEN, DO, AND, OR e IN. O conector IF é seguido de um conector THEN. O conector WHEN é seguido de um conector THEN e o conector WHILE é seguido de um conector DO. Os conectores AND e OR são utilizados para compor expressões de modo que serja possível realizar mais de uma ação ou no caso de uma opção satisfazer o desejado. O conector IN é utilizado para se descrever uma opção que está presente em um conjunto.
- endurance.dat: descreve a duração da missão em termos de tempo, caso seja necessário re-abastecimento. São eles: Minutes, Hours e Days.

- environment type.dat: descreve o tipo de ambiente no qual a missão será executada. É importante para definir certos cuidados a serem tomados para atingir os objetivos da missão. Os tipos de ambiente podem ser: URBAN ZONE, WAR ZONE, FOREST, DESERT, MOUNTAIN, MOORLAND, TUNDRA, PRAIRIE / STEPPE, SAVANNAH, MEDITERRANEAN SCRUB e POLAR.
- mission type.dat: descreve o tipo da missão global, ou seja seu principal objetivo. O tipo da missão pode ser: Sequential Pictures, Pictures e Videos.
- patterns.dat: descreve as *patterns* que são elementos com uma definição em baixo nível que os identifica. Quando codificadas novas *patterns* pode-se adicioná-las neste arquivo e então selecioná-las na interface. As *patterns* existentes são: FOG, FIRE, MOVEMENT e OBJECT.
- priorities.dat: descreve as prioridades da missão global, dos *waypoints* e dos nós sensores estáticos para utilização no algoritmo de controle da missão. São elas: NORMAL, MEDIUM, HIGH e VERY HIGH.
- secrecy.dat: descreve o tipo de sigilo da missão. Os tipos de sigilo são: Ostensive, Secret e Top-secret.
- sensor type.dat: descreve o tipo de sensor a ser usado na missão. O tipo de sensor pode ser: Active, Passive e Active and Passive.
- time of day.dat: descreve se a missão transcorrerá em um abiente matutino no caso em presença de luz ambiente ou em um ambiente noturno, neste caso em ausência de luz ambiente. Esta opção auxilia na seleção dos sensores a serem usados na missão. São eles: Night e Daylight.
- weather.dat: descreve o clima onde transcorrerá a missão. Esta opção é importante pois auxilia na escolha do sensor e veículo mais adequado para realizar a missão. As opções de condição meteorológica são: Sunny, Cloudy, Stormy, Rainy, Windy, Snowy, Foggy, Hot, Cold, Wet e Dry.

5.6.2 Dados de saída

A interface gera um arquivo de saída que é um arquivo em formato XML que descreve a missão em alto nível. Todos os dados inseridos pelo usuário são descritos neste arquivo que será utilizado na interface de controle da missão para subdividir a missão aos agentes do sistema.

5.6.3 Descrição dos componentes

Package data: contém toda a estrutura de dados utilzada na aplicação. São classes de armazenamento dos dados.

Package gui: contém toda a interface gráfica da aplicação, o console que emite mensagens para o usuário, o sistema de abas móveis e de visualização dos arquivos em formato XML.

Package kernel: contém o kernel que realiza todas as funções da aplicação e a ligação entre a interface com o usuário e a estrutura de dados.

Package main: contém o *launcher* do programa e informações sobre as constantes da aplicação.

Package tools: contém ferramentas que auxiliam outras classes do programa. A classe principal é a classe Tools.java que contém métodos para associar ícones aos botões e à aplicação e um método para ler arquivos do tipo *.dat.

Package xml: contém a classe XmlManager que executa todas as ações referentes ao tratamento do arquivos XML.

5.7 Interface gráfica

Nesta seção são exibidas todas as janelas da interface, descrição dos botões e painéis.

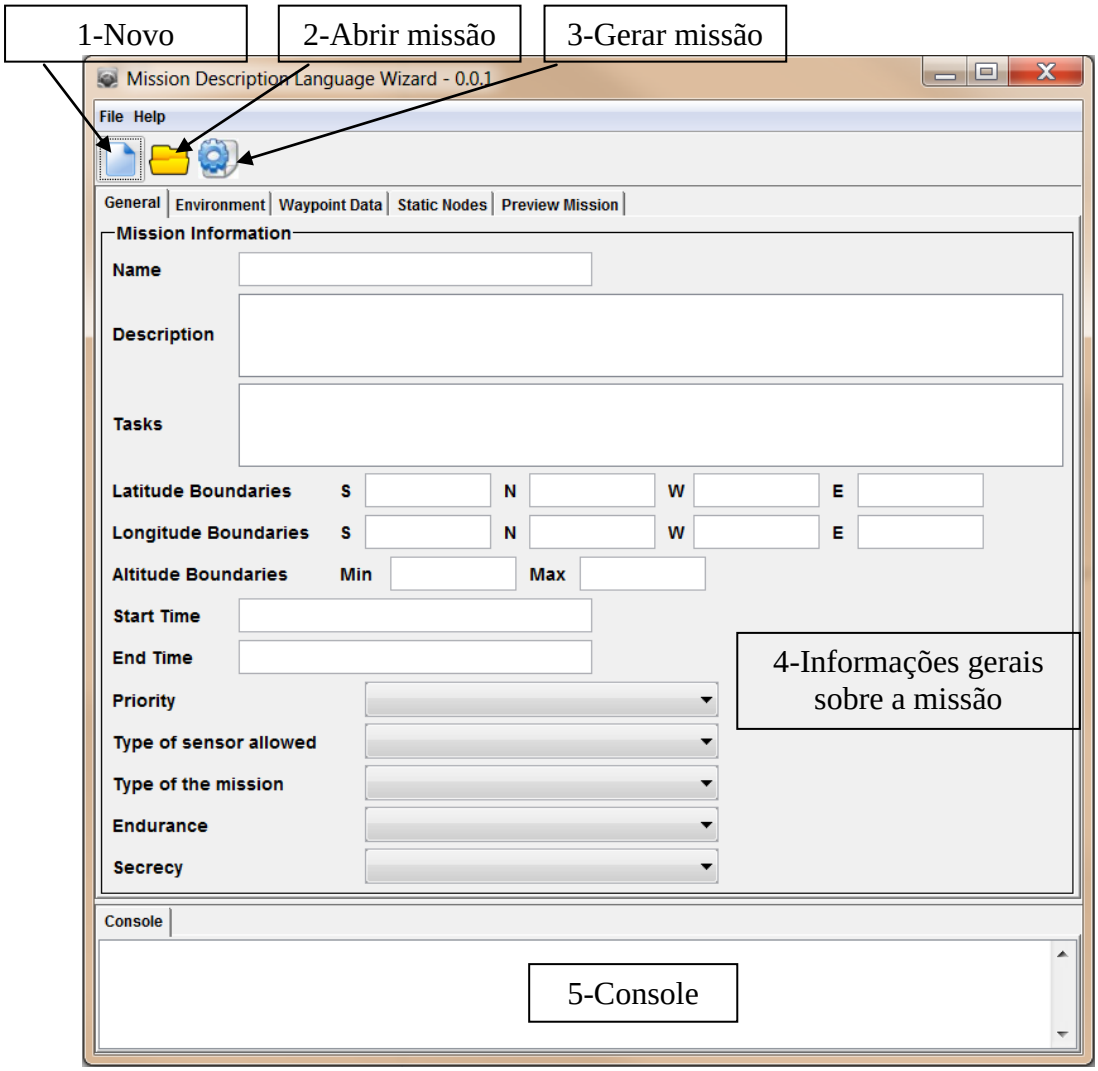


Figura 5.2: Mostra a interface principal, com as informações gerais sobre a missão.

Esta interface contém as seguintes informações e funcionalidades:

1-Novo arquivo de missão.

2-Abrir arquivo de missão descrito na MDL.

3-Gerar arquivo XML que descreve a missão.

4-Informações gerais sobre a missão.

5-Console para emissão de mensagens ao usuário.

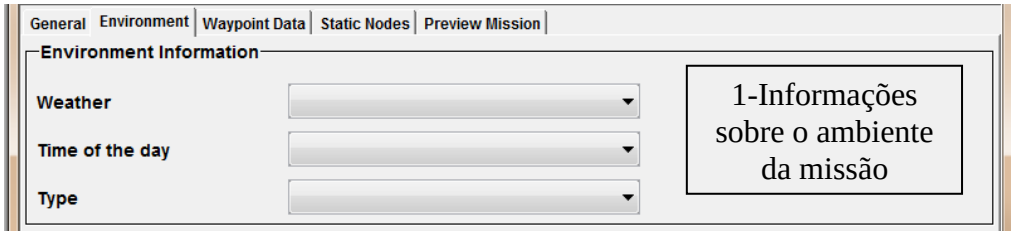


Figura 5.3: Informações sobre o ambiente.

A Figura 5.3 exibe o painel para seleção das informaçõs sobre o ambiente.

1-Informações sobre o ambiente da missão.

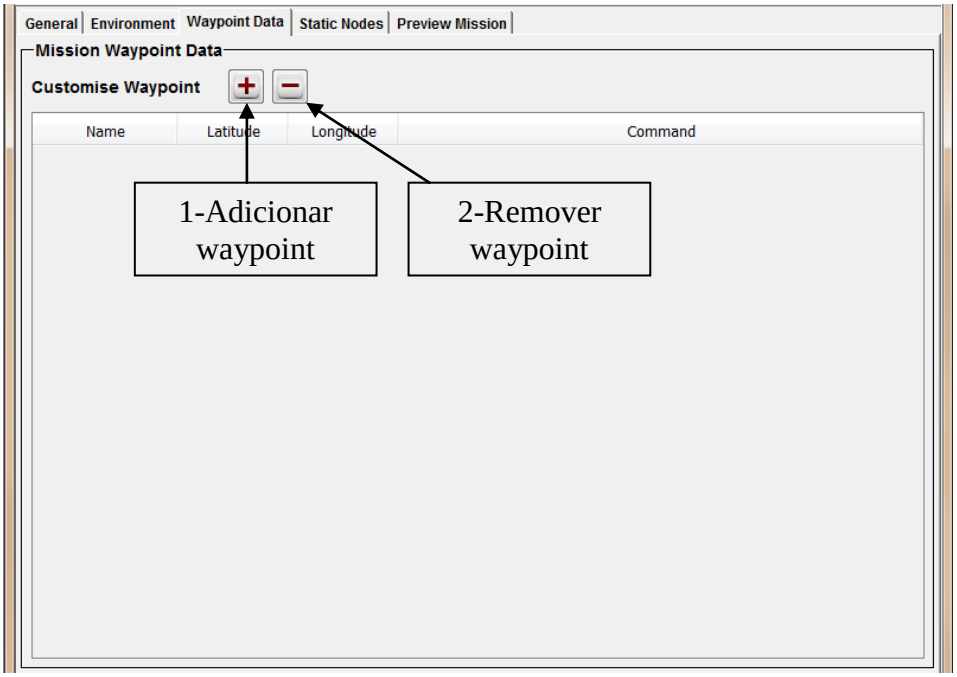


Figura 5.4: Informações sobre os waypoints.

A Figura 5.4 exibe o painel para se adicionar e configurar um *waypoint* à missão.

1-Adicionar um novo *waypoint* à missão. Este botão abre a janela para configurar o comando.

2-Remover *waypoint* selecionado na tabela.

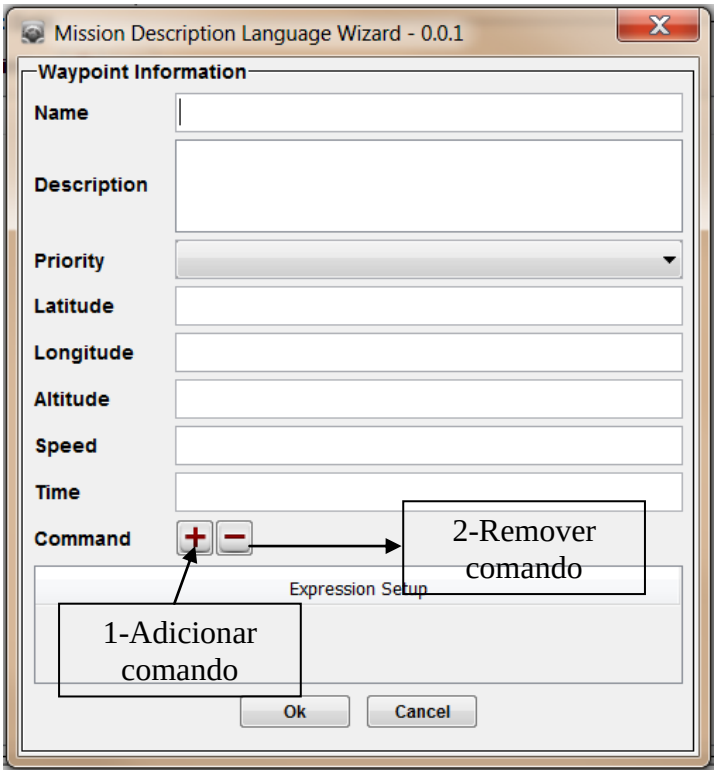


Figura 5.5: Janela para adicionar um waypoint à uma missão.

A Figura 5.5 exibe a janela para definir as informações sobre o *waypoint*.

1-Adicionar um novo comando para montar a expressão que comporá o comando final. Este botão abre a janela para configurar a expressão.

2-Remover comando selecionado na tabela.

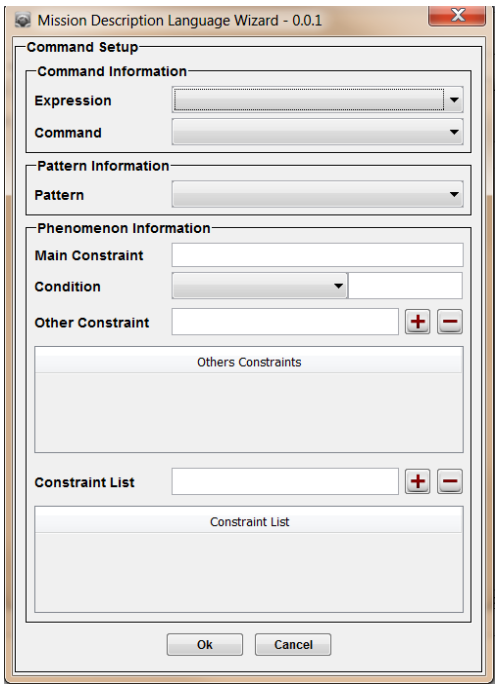


Figura 5.6: Janela para definir uma expressão que compõe um comando.

A figura 5.6 exibe a janela para compor a expressão que formará o comando.

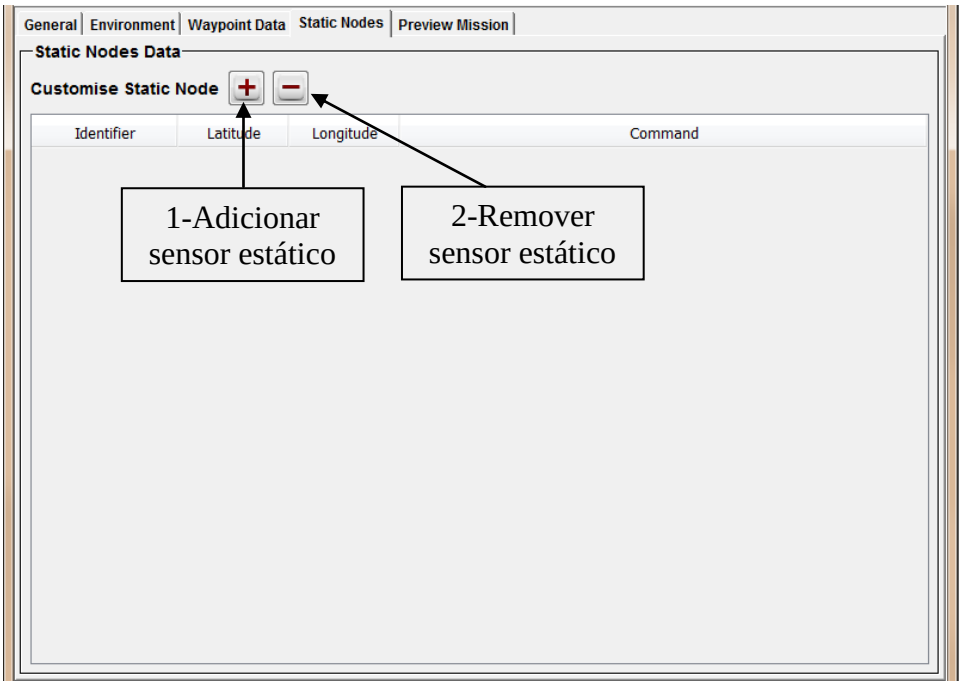


Figura 5.7: Informações sobre os nós sensores estáticos.

A figura 5.7 exibe o painel para se adicionar e configurar um nó sensor estático.

1-Adicionar um novo sensor estático à missão. Este botão abre a janela para configurar o sensor estático.

2-Remover sensor estático selecionado na tabela.

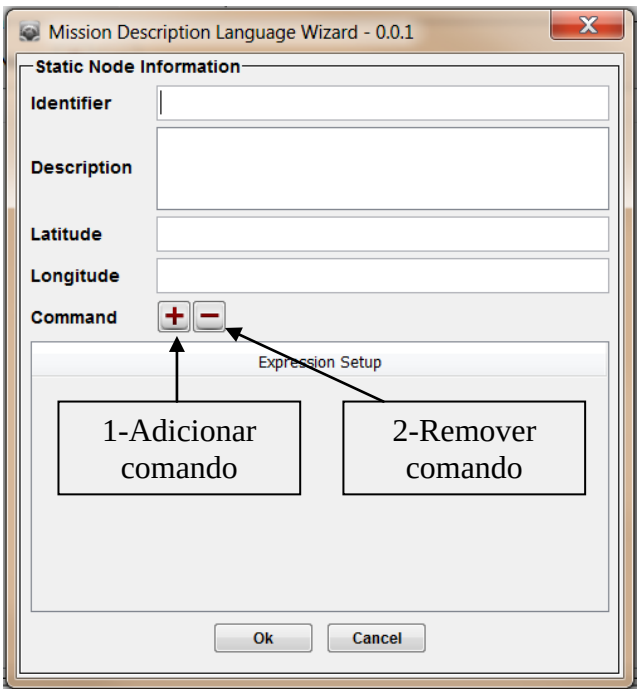


Figura 5.8: Janela para adicionar um sensor estático à missão.

A Figura 5.8 exibe a janela para se definir as informações sobre o nó sensor estático.

1-Adicionar um novo comando para montar a expressão que comporá o comando final. Este botão abre a janela para configurar a expressão.

2-Remover comando selecionado na tabela.

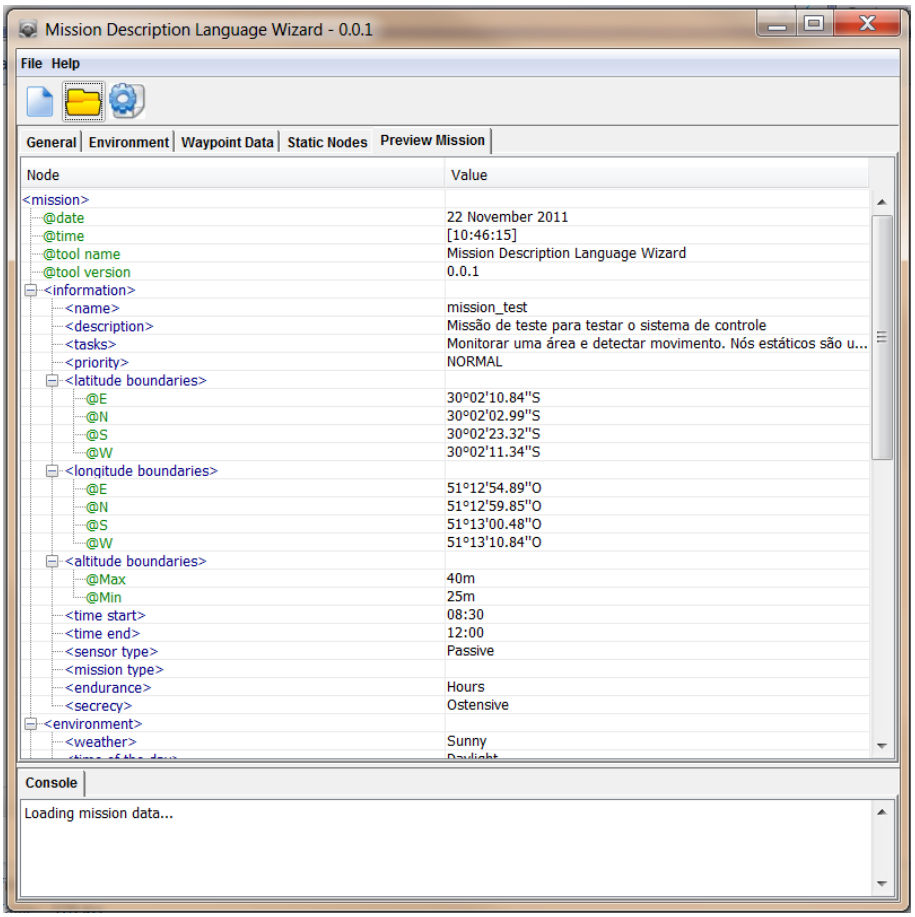


Figura 5.9: Visualizador da missão em formato XML.

A Figura 5.9 exibe o painel que exibe a missão gerada em formato XML.

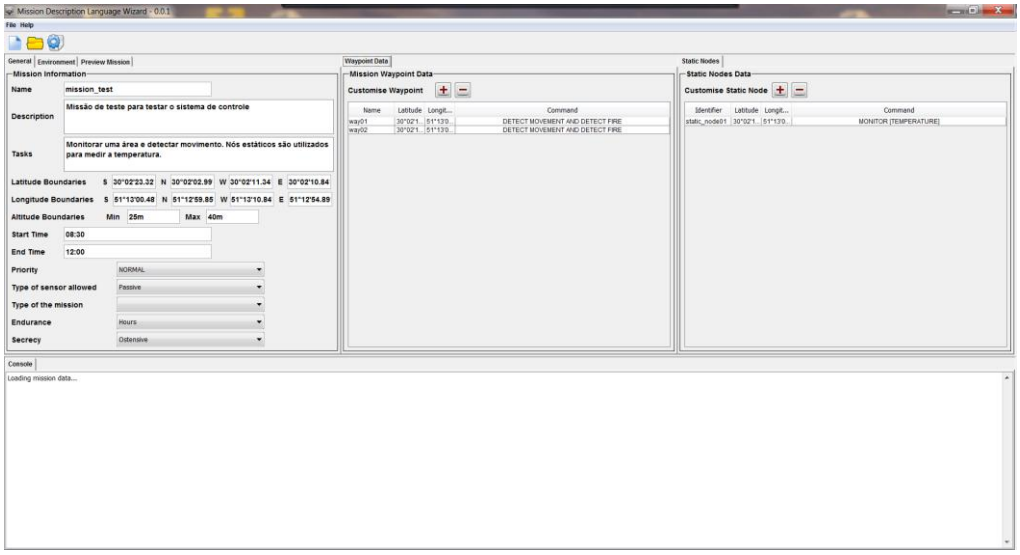


Figura 5.10: Exemplo do sistemas de abas móveis.

A Figura 5.10 exibe o sistema de abas móveis, onde é possível mover as abas para a direita e esquerda e assim exibir o conteúdo de várias abas no mesmo painel.

6 SISTEMAS MULTIAGENTES

Sistema multiagente é um sistema composto por múltiplos agentes inteligentes que interagem entre si de forma a realizar um determinado conjunto de tarefas ou objetivos. Estes objetivos podem ser comuns a todos os agentes ou não. Sistemas multiagentes podem ser usados para resolver problemas que são difíceis de resolver para um agente individual ou um sistema monolítico. Estes sistemas tendem a achar a melhor solução para seus problemas sem intervenção externa, e suas principais características são a escalabilidade e a flexibilidade na construção de sistemas autônomos. Os agentes dentro de um sistema multiagente podem ser heterogêneos ou homogêneos, colaborativos ou competitivos, etc. Ou seja, a definição dos tipos de agentes depende da finalidade da aplicação a qual o sistema multiagente está inserido.

Uma definição bastante aceita é a dada por Wooldridge M., que diz "Sistemas Multiagentes (SMA) são sistemas compostos por múltiplos elementos computacionais interativos denominados agentes. Agentes são entidades computacionais com duas habilidades fundamentais: (1) de decidir por si próprios o que devem fazer para satisfazer seus objetivos de projeto e (2) interagir com outros agentes de forma social (...)" (WOOLDRIDGE M., 2002).

Para se determinar um sistema multiagente pode-se destacar três principais características: a) a comunicação, ou seja, como será realizada a comunicação entre os agentes e que tipo de protocolo será utilizado; b) como ocorrerá a interação, que linguagem os agentes devem usar para interagirem entre si; e c) como será realizada a coordenação dos agentes para atingir os objetivos individuais e globais.

6.1 Agentes

Agentes são entidades que se adaptam, reagem e provocam mudanças no meio ao qual estão inseridos para atingir seus objetivos. Uma definição formulada pela Foundation for Intelligent Physical Agents (FIPA), que define um agente como "uma entidade que reside em um ambiente onde interpreta dados através de sensores que refletem eventos no ambiente e executam ações que produzem efeitos no ambiente. Um agente pode ser software ou hardware puro…" [FIPA].

No geral, agentes seriam entidades de hardware e software autônomas que atuam em determinado ambiente de forma a interagir com este e com outros agentes, além de produzir ações e percepções sem requerer intervenções humanas. Numa abordagem mais aplicada a Inteligência Artificial um agente ideal teria que ser capaz de funcionar indefinidamente e adquirir experiências e conhecimentos acerca do ambiente que está interagindo. Ou seja, ser capaz de "aprender" e tomar decisões a partir das suas experiências.

Existem certas propriedades que caracterizam um agente. Uma delas é a autonomia para tomar decisões e realizar ações importantes para conclusão de uma tarefa ou objetivo, ou seja, ter a capacidade de agir de maneira independente através das informações obtidas pelos seus próprios sensores. Devem operar sem a intervenção humana e ter algum tipo de controle sobre a tomada de decisões. Associado à autonomia está a pró-atividade, que nada mais é do que a capacidade que o agente deve ter de tomar iniciativas e exibir um comportamento baseado em seus objetivos. Outra característica importante é a reatividade e adaptabilidade que é a capacidade de reagir e se adaptar rapidamente a alterações no ambiente. Um agente deve ser robusto para ser capaz de tomar decisões baseadas nas informações adquiridas sendo elas escassas ou incompletas, possuir tolerância a falhas e capacidade de adaptação e aprendizado através das suas experiências. Enfim, os agentes devem apresentar a capacidade de cooperação, ou seja devem interagir entre si e com o ambiente trocando informações para benefício mútuo na execução de uma tarefa complexa. Para ser obter a capacidade de cooperação é necessário que o agente também possua a habilidade de comunicação para trocar informações e interagir cooperativamente com outros agentes para atingir os objetivos individuais e globais. A capacidade de analisar e tomar decisões fundamentado nos seus conhecimentos atuais e passados e das suas interações com outros agentes e o ambiente pode ser baseado em regras ou conhecimento. O primeiro define um conjuntos de condições prévias para avaliar as condições do ambiente externo. O segundo define um grande conjunto de dados sobre cenários anteriores e ações resultantes, dos quais deduz-se seus movimentos futuros.

Para Wooldridge em (WEISS 1999), uma flexibilidade de ações tais como reatividade, pró-atividade e sociabilidade, é suficiente para classificar um agente como inteligente. Contudo, um agente não precisa apresentar todas essas características ao mesmo tempo. Entretanto é a presença de algumas destas características que diferencia um agente de simples programas e objetos.

6.2 Tipos de Agentes

Para se determinar um sistema multiagente é necessário classificar os tipos de agentes para que seja possível configurar os agentes individuais e suas funções. Podemos classificar os agentes quanto à mobilidade, ao relacionamento entre agentes e à capacidade de raciocínio.

- Agentes móveis: são agentes que têm a mobilidade como característica principal. A utilização deste tipo de agente possui uma potencialidade muito grande devido à heterogeneidade cada vez maior das redes e seu grande poder na tomada de decisões baseadas em grandes quantidades de informação.
- Agentes estacionários: são agentes estáticos que não apresentam mobilidade, ou seja, estão fixos no ambiente.
- Agentes competitivos: são agentes que competem entre si para a realização de seus objetivos ou tarefas.
- Agentes coordenados ou colaborativos: são agentes que colaboram entre si com a finalidade de atingir um objetivo global. Realizam tarefas específicas, entretando coordenando suas ações de forma que suas atividades sejam completadas.

- Agentes reativos: são agentes que reagem a estímulos sem possuir memória do que já foi realizado e nem previsão da ação a ser tomada no futuro. Este tipo de agente é incapaz de prever e antecipar suas ações, e portanto geralmente atua em sociedade como uma colônia de formigas por exemplo.
- Agentes cognitivos: são agentes que possuem reciocínio sobre as ações tomadas no passado e planejam as ações a serem tomadas no futuro. Um agente cognitivo é capaz de resolver problemas de forma independente, possui objetivos e planos bem específicados os quais permitem atingir seu objetivo final. Para que isso se concretize, cada agente deve ter uma base de conhecimento disponível, que compreende todos os dados disponíveis para realizar suas tarefas e os dados resultantes de suas interações com outros agentes e com o próprio ambiente. Sua representação interna e seus mecanismos de inferência o permitem atuar independentemente de outros agentes o que garante uma grande flexibilidade na forma de expressão de seu comportamento.
- Agentes de software e físicos: agentes físicos podem ser robôs, pessoas, uma plataforma de hardware mais um software, etc., dependendo do escopo do sistema. Agentes de software são programas com as propriedades básicas de um agente que são executadas em uma determinada plataforma.

6.3 Sistemas multiagentes aplicados em RSSF e controle de VANTs

A utilização do paradigma orientado a agentes em uma rede de sensores sem fio tem como objetivo reduzir a intervenção humana e determinar um comportamento autônomo aos componentes do sistema durante a realização de uma missão. O VANT participa do sistema multiagente como agente físico móvel e os sensores estáticos como agentes físicos estacionários. A tomada de decisão do VANT deve levar em consideração os dados coletados a partir dos seus próprios sensores e das informações recebidas de outros agentes da rede. Este sistema deverá ter duas principais capacidades: primeiramente o sistema deverá ter um comportamento autônomo e tomar a melhor decisão para suprir as necessidades dos agentes; para atingir o primeiro objetivo o VANT deverá interagir com os demais agentes do sistema de forma a negociar e cooperar para atingir os objetivos finais da missão. Para atingir a cooperabilidade entre os agentes, estes devem trabalhar juntos para alcançar os objetivos comuns através de interações e coordenação entre os agentes individuais. O algortimo de controle e tomada de decisão utilizado é descrito em detalhes no próximo capítulo. Por mais interessante que seja a idéia de um sistema completamente autônomo, é necessária a existência de um sistema supervisor que possa intervir em casos críticos.

7 INTERPRETADOR DA MDL UTILIZANDO O PARADIGMA ORIENTADO A AGENTES

Uma maneira natural de se modelar redes de sensores é considerar cada componente do sistema como um agente em um sistema multiagente, com funções bem definidas, autônomas e colaborativas. A linguagem MDL é utilizada para descrever a missão em alto nível e utilizando-se o paradigma orientado a agentes o interpretador define um conjunto de sub-missões independentes que são utilizadas para configurar os agentes que controlam as operações no sistema.

7.1 Problemas com a definição de um comportamente autônomo de um agente dinâmico

A determinação de uma missão deve levar em consideração certos fatores como descrito em (KARIM, S.; HEINZE, C. e DUNN S., 2004).

7.1.1 Tempo de vôo e distância a ser percorrida

O agente dinâmico pode ser operado com bateria ou combustível, para tanto é necessário determinar o tempo de vôo e a distância a ser percorrida pelo VANT. Para missões mais longas ou possíveis re-definições de uma missão pode ser necessário que o VANT retorne para recarregar ou trocar de bateria.

7.1.2 Condições ambientais na zona da missão

Os VANTs são veículos aéreos geralmente de pequeno porte e portanto bastante suscetíveis a variações ambientais como velocidade do vento. Em certas condições o VANT pode não ter condições de prosseguir uma missão, então aciona-se o comando de abortar missão e retornar.

7.1.3 Regulamentações e restrições de vôo

As regulamentações e restrições de vôo exigem que o VANT passe por certificações de segurança. Em especial, veículos autônomos devem exibir um comportamento determinístico, ou seja, o VANT não pode considerar muitas possibilidades, pois isto poderia tornar seu comportamento potencialmente randômico em situações não conhecidas.

7.1.4 Dados provenientes dos sensores com potencial limitado

O VANT possui certos sensores acoplados como GPS, medidor de altura e velocidade. Estes sensores possuem uma determinada precisão e devem ser bem configurados para suprir as necessidades de cada situação.

7.2 Arquitetura do sistema

A arquitetura do sistema de controle da missão é baseada em um algoritmo utilizado em simulações militares chamado OODA (Observe, Orient, Decide, Act). Este modelo de decisão é baseado na premissa que a tomada de decisão em determinada situação é um laço infinito da sequência: observação do ambiente em que o agente está situado, orientação baseada nas observações atuais e passadas, decisão sobre o que será feito e execução da ação escolhida (KARIM, S.; HEINZE, C. e DUNN S., 2004).

A missão global é descrita pela interface da MDL, onde são descritas as informações sobre a missão, o ambiente, os *waypoints* e os nós estáticos. Os *waypoints* são pontos dentro da área limite da missão onde o agente físico móvel deverá ir para executar determinado comando, como por exemplo, captura de imagens. Os nós estáticos também são agentes físicos independentes, eles são sensores estáticos responsáveis por executar alguma ação. Estes sensores estáticos estão localizados na área da missão e estão sempre coletando dados. O interpretador da MDL utiliza o arquivo da missão gerado para particionar a missão global em sub-missões a serem executadas pelo nó móvel, onde cada *waypoint* é considerado como uma sub-missão. As sub-missões são definidas com uma posição, uma prioridade e uma ação. O nó móvel então aplicará o algoritmo de tomada de decisão para decidir qual *waypoint* ele irá tratar primeiro. Esta decisão é feita levando em consideração as informações de posição e prioridade. Inicialmente, apenas os *waypoints* estarão presentes na missão. Entretanto, é possível que um nó estático emita um alarme e seja necessário re-configurar a missão para incluir este novo *waypoint*, que pode incluir, por exemplo, uma parada para coleta de dados de um nó estático.

7.2.1 Dados de entrada

O interpretador possui como entrada um arquivo de missão gerado pela interface da MDL. A missão descrita na MDL é interpretada e gera as sub-missões que serão distribuídas aos agentes do sistema de acordo com o algoritmo. O interpretador possui a função de simulação. Para definir uma simulação é necessário definir o ponto de ínicio e fim da missão, o tempo de cada passo de realização da missão e ao decorrer da missão pode-se enviar alarmes para simular os sensores estáticos.

7.2.2 Dados de saída

Após abrir um arquivo de missão gerado pela MDL o interpretador gera os arquivos de sub-missão para distribuir para os agentes do sistema. Estes arquivos são gerados no formato texto. Inicialmente era previsto que estes arquivos fossem utilizados na interface de controle em baixo nível para controlar o VANT e os sensores estáticos.

7.2.3 Descrição dos componentes

Assim como a interface da MDL, utiliza-se o conceito de MVC (vide ANEXO-A) para particionar o software. A estrutura de dados é a mesma utilizada na interface MDL, assim como o módulo de visualização da missão em XML e o console para emissão de mensagens ao usuário. A interface de simulação é uma thread que é criada no início de uma simulação que realiza a execução da missão.

7.3 Diagramas de blocos do algoritmo de controle da missão

Nesta secão são descritas as 4 fases do algoritmo de controle da missão. Os diagramas em blocos auxiliam na compreensão do algoritmo.

Observe

A fase de observação consiste na compilação dos dados de posicão e velocidade atuais, os dados sobre os *waypoints* que compõem a missão, dados sobre as interações com o ambiente e o recebimento de alarmes dos nós estáticos. Nesta fase apenas observa-se estas variáveis.

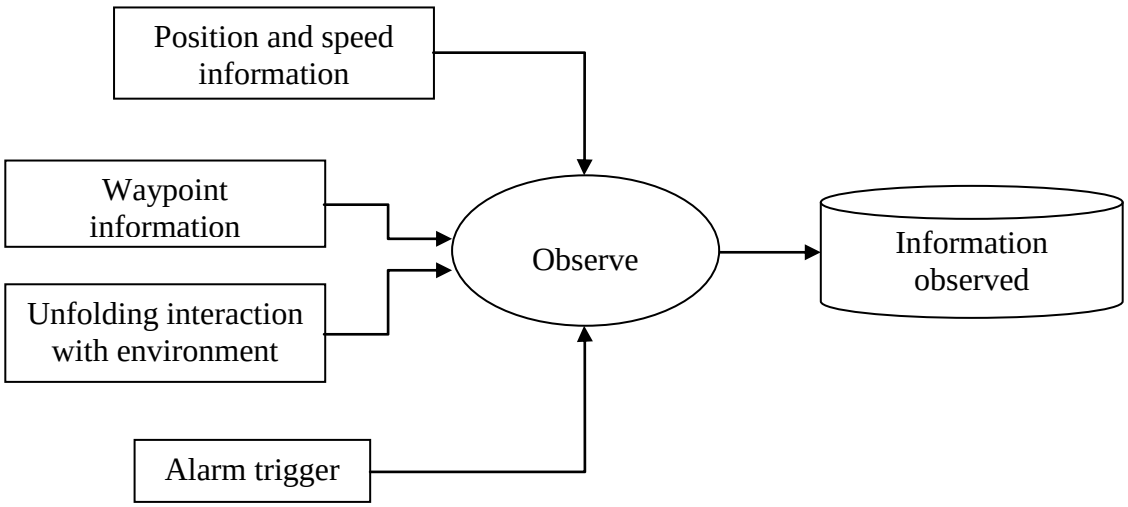


Figura 7.1: Diagrama da fase de observação.

Orient

Na fase de orientação o agente analisa a situação de acordo com as informações observadas e passadas. Nesta fase analisa-se as variáveis de distância e prioridade dos *waypoints* e alarmes recebidos, assim como as informações sobre o ambiente e continuidade da missão.

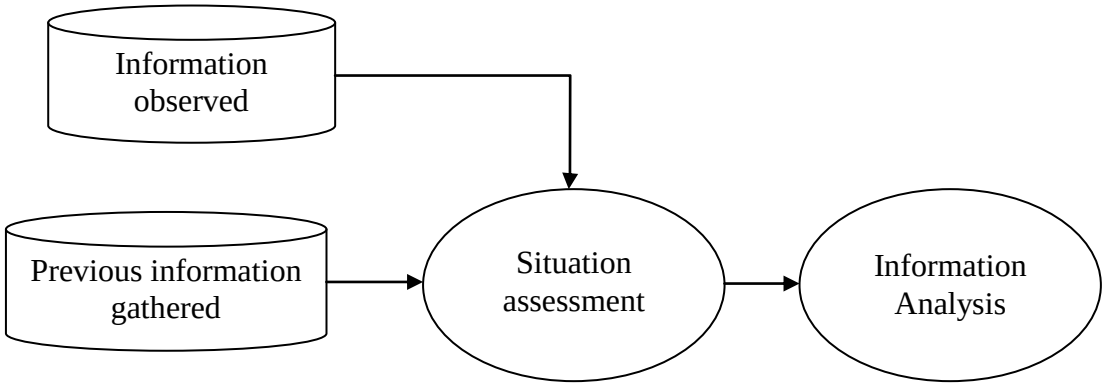


Figura 7.2: Diagrama que descreve a fase de orientação de acordo com as informações observadas e passsadas.

Decide

Nesta fase é onde efetivamente é tomada a decisão sobre a ação que será tomada de acordo com as análises sobre as informações coletadas. Em caso de impossibilidade devido a condições ambientais a missão é abortada com retorno até o ponto de partida. Se um alarme foi recebido, utiliza-se a política de prioridade onde o alarme recebido tem prioridade máxima e deve ser tratado. Caso nenhuma das opções anteriores ocorrerem a missão prossegue para o próximo *waypoint*.

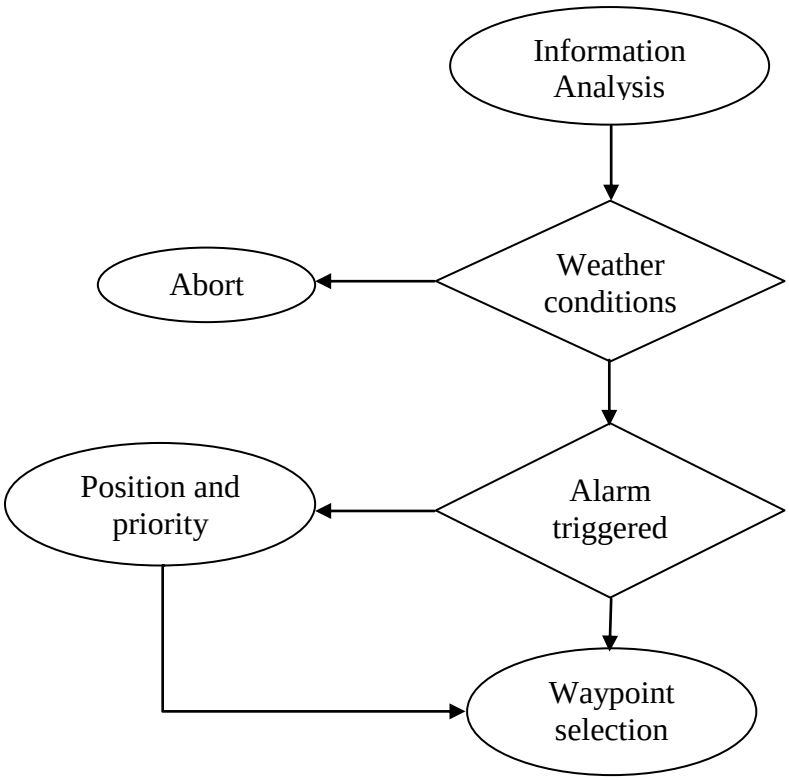


Figura 7.3: Diagrama que descreve a fase de tomada de decisão.

Act

Nesta fase o agente móvel chega no *waypoint* e realiza o comando especificado na missão.

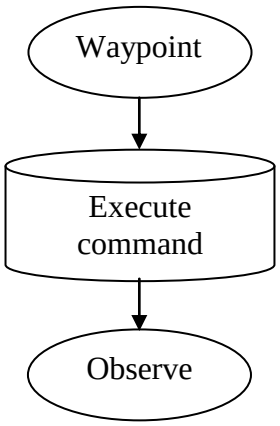


Figura 7.4: Diagrama que descreve a fase de ação após tomada de decisão pelo agente autonômo.

Código em Java dos agentes que executam o algoritmo descrito no diagrama em blocos. O código completo está disponível no Apêndice B para ser utilizado na implementação deste trabalho em uma plataforma real.

Listing 7.1: Código em Java das fases “Observe” e “Orient”.

```
1   public void observe(String alarm) {
2       String currentLat = getCurrentPosLat();
3       String currentLong = getCurrentPosLong();
4       WaypointMap wMap = getWaypointMap();
5       String environmentData = getEnvironmentData();
6       this.alarm = alarm;
7       orient(currentLat, currentLong, wMap, environmentData);
8   }
9
10  private void orient(String currentLat, String currentLong, WaypointMap
                wMap, String envData) {
11      this.waypointMap = wMap;
12      Waypoint w = getNextWaypoint(currentLat, currentLong);
13      decide(w, envData);
14  }
15
16  private boolean checkPriority(String alarm) {
17      if (!alarm.equals("")) {
18          return true;
19      } else {
20          return false;
21      }
22  }
```

Na linha 4 todos os waypoints ainda não visitados são carregados e a variável de alarme é setada na linha 6 caso haja o recebimento de algum alarme. Na linha 10 é executado o método para orientar o VANT na escolha do próximo *waypoint* que será tratado, na linha 12 o método seleciona o próximo *waypoint*.

Listing 7.2: Código em Java da fase “Decide”.

```
24    private void decide(Waypoint w, String envData) {
25        if (checkWeatherCondition(envData)) {
26            if (checkPriority(this.alarm)) {
27                StaticNode sn = getStaticNode(alarm);
28                actAlarm(sn);
29            } else {
30                actWaypoint(w);
31            }
32        } else {
33            abortMission();
34        }
35    }
```

Nesta fase o VANT possui todas as informações necessárias para tomada de decisão. Na linha 26 é checada a prioridade do alarme, como o alarme sempre possui prioridade máxima ele sempre será tratado independente de outras variáveis. Se a linha 26 for verdadeira, os dados sobre o nó estático são carregados e o método para executar é chamado. Caso seja falsa, o VANT executa o método para tratar o próximo waypoint selecionado.

Listing 7.3: Código em Java da fase “Act” em caso de seleção de waypoint.

```
36    private void actWaypoint(Waypoint w) {
37        SystemConsole.append("Going to waypoint: " + w.getName() + "
          Latitude: " + w.getLatitude() + Longitude:  + w.getLongitude());
40        simulationStep(getTimeStep());
41        SystemConsole.append("Arriving at waypoint: " + w.getName() + "
          Latitude: " + w.getLatitude() + " Longitude: " + w.getLongitude());
43        SystemConsole.append("Executing command: " +
          w.getExpression());
44        simulationStep(w.getWaypointTime() * getTimeStep());
45        setCurrentPosLat(w.getLatitude());
46        setCurrentPosLong(w.getLongitude());
47    }
```

Este médoto envia os comandos de controle ao VANT para que ele voe até o waypoint selecionado e ao chegar execute o comando específicado na MDL.

Listing 7.4: Código em Java da fase “Act” em caso de recepção de um alarme.

```
49    private void actAlarm(StaticNode sn) {
50        SystemConsole.append("ALARM RECEIVED: Priority Policy
          Selected");
51        SystemConsole.append("Reconfiguring mission...");
52        SystemConsole.append("Going to static node: " + sn.getIdentifier() +
          Latitude:  + sn.getLatitude() + Longitude:  + sn.getLongitude());
55        simulationStep(2 * getTimeStep());
56        SystemConsole.append("Arriving at static node: " +
          sn.getIdentifier() + Latitude:  + sn.getLatitude() + Longitude:  +
          sn.getLongitude());
59        SystemConsole.append("Getting data from command: " +
          sn.getExpression());
60        simulationStep(getTimeStep());
61        SystemConsole.append("Returning to last point: Latitude: " +
          getCurrentPosLat() + Longitude:  + getCurrentPosLong());
63        resetAlarm();
64    }
```

Este método é acionado em caso de recepção de um alarme de um nó estático, então a missão é re-configurada e o VANT voa até o nó estático e adquire os dados disponíveis no sensor estático. Após o tratamento deste alarme, o VANT retorna a posição de onde partiu e prossegue a missão até visitar todos os waypoints.

7.4 Algoritmo de tratamento de alarmes e tomada de decisão

Uma missão é composta por *waypoints* e nós sensores estáticos que realizam determinada função e emitem alarmes para que o VANT colete os dados. Durante uma missão, o VANT segue a sequência de *waypoints* determinada pela menor distância da posição atual para o *waypoint*, quando então pode receber um alarme e re-configurar a missão para atênde-lo. Inicialmente utiliza-se uma política de prioridade, onde o alarme recebido possui a maior prioridade. Então o VANT decide tratar o alarme recebido, indo até o ponto onde o sensor estático se encontra para coletar os dados e em seguida retornando para o ultimo *waypoint* para prosseguir a missão.

A relação entre distância e prioridade é particulamente complexa, por este motivo foi escolhido atribuir prioridade máxima sem levar em consideração a distância até o sensor estático. Uma outra possibilidade é a atribuição de pesos para prioridade e distância para tratamento do alarme e então decidir se o próximo passo da missão será ir para o próximo *waypoint* ou tratar o alarme recebido indo até o sensor estático.

7.5 Interface gráfica

A interface é composta por três abas e um console que emite as mensagens de andamento da missão e de simulação ao usuário.

Figura 7.1: Interface gráfica principal.

A Figura 7.1 exibe a interface principal para abrir um arquivo de missão descrito em MDL.

1-Abrir arquivo de missão descrito na MDL

2-Executar a missão após abrir o arquivo de missão.

3-Enviar comando específico para o VANT, os comandos não foram implementados já que a interface em baixo nível não foi concluída.

4-Tabela que contém todos os waypoints presentes na missão e o respectivo comando a ser executado.

5-Tabela que contém todos os nós sensores estáticos presentes na missão, com sua localização e comando que está sendo executado para coleta de dados.

6-Console que emite mensagens ao usuário sobre a execução da missão, simulação, e mensagens pertinentes à execução do software.

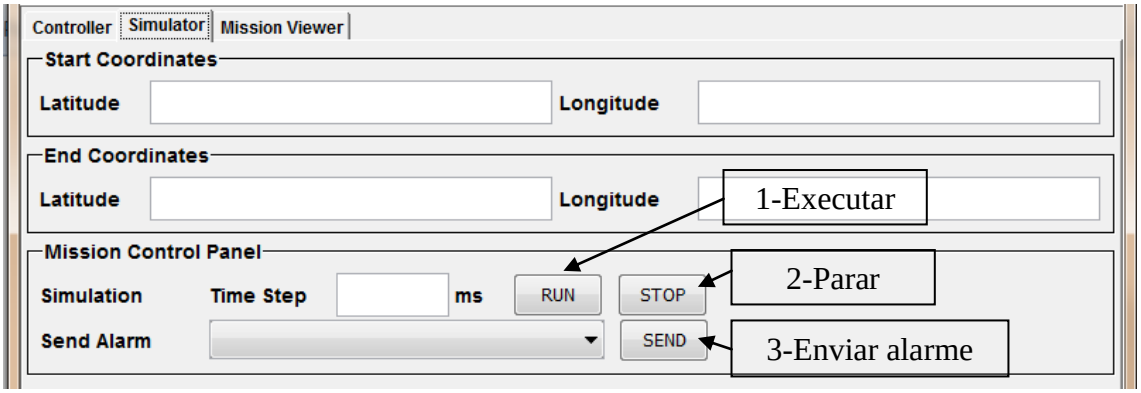


Figura 7.2: Interface de simulação de uma missão.

A Figura 7.2 exibe o painel de simulação de uma missão.

1-Iniciar simulação da missão.

2-Parar simulação da missão.

3-Enviar alarme de um nó estático.

8 EXPERIMENTOS

Neste trabalho foi implementado uma interface de planejamento de missões que descreve os *waypoints* a serem visitados por VANTs e os nós sensores estáticos componentes de uma missão em uma rede de sensores sem fio. A interface gera um arquivo em formato XML, que é utilizado na interface de controle para executar o algoritmo de controle. Inicialmente esperava-se ter uma interface de baixo nível para enviar os comandos para uma plataforma VANT real, a plataforma da Skydrones descrita em 3.2, porém como esta interface em baixo nível não pode ser concluída a tempo para realização da integração e realização de testes, foi incluído uma interface de simulação para verificar o funcionamento do software de controle de missões desenvolvido neste trabalho. O experimento simulado é o planejamento de uma missão fictícia utilizando a interface MDL e simulação utilizando o interpretador MDL para descrever as etapas da missão. O objetivo deste experimento é checar se a missão é bem descrita e corretamente interpretada. As entradas inseridas no planejador de missão foram selecionadas através do Google Maps, onde se determinou uma área de interesse e selecionou-se pontos de coordenadas geográfica em Porto Alegre onde estariam localizados nós sensores estáticos e *waypoints* de interesse. A saída é um arquivo XML que descreve todas as informações sobre a missão. Este arquivo é utilizado no interpretador para visualizar a correta interpretação dos dados e simular a execução do algoritmo de controle com o envio de alarmes pelos sensores estáticos. Os alarmes são mensagens enviadas pelos nós sensores estáticos ao VANT. O arquivo XML da missão pode ser visualizado no apêndice, assim como a visualização gerada pela interface.

8.1 Simulação

Na primeira simulação foi executado a missão sem nenhuma emissão de alarme pelos nós sensores estáticos. Pode-se verificar que o *waypoint* mais próximo foi selecionado e visitado pelo VANT fictício e em seguida executada a ação específicada. Abaixo está o log de simulação da missão. O ponto de ínicio da missão é a latitude 30°02'11.34"S e longitude 51°13'10.84"O, e não foi definido um ponto final para a missão e portante ela termina no ultímo *waypoint*. O passo da simulação foi definido por default como 1000ms. A simulação não leva em consideração fatores ambientais e a velocidade do VANT.

Tabela 8.1: Resultado da simulação de uma missão.

1	Default time step: 100
2	Starting mission...
3	Current position: Latitude: 30°02'11.34"S Longitude: 51°13'10.84"O

5	Arriving at waypoint: way02 Latitude: 30°02'15.52"S Longitude: 51°13'03.50"O
6	Executing command: DETECT MOVEMENT AND DETECT FIRE
7	Going to waypoint: way01 Latitude: 30°02'11.56"S Longitude: 51°13'07.43"O
8	Arriving at waypoint: way01 Latitude: 30°02'11.56"S Longitude: 51°13'07.43"O
9	Executing command: DETECT MOVEMENT AND DETECT FIRE
10	Ending mission...
11	Final position: Latitude: 30°02'11.56"S Longitude: 51°13'07.43"O
12	Time elapsed: 4026 s

Em uma segunda simulação, utilizando os mesmos dados de entrada, o nó estático enviu um alarme para o VANT e a simulação seguiu como descrito abaixo:

Tabela 8.2: Resultado da simulação de uma missão ao recebimento de um alarme.

1	Default time step: 1000
2	Starting mission...
3	Current position: Latitude: 30°02'11.34"S Longitude: 51°13'10.84"O
4	Going to waypoint: way02 Latitude: 30°02'15.52"S Longitude: 51°13'03.50"O
5	Arriving at waypoint: way02 Latitude: 30°02'15.52"S Longitude: 51°13'03.50"O
6	Executing command: DETECT MOVEMENT AND DETECT FIRE ALARM RECEIVED: Priority Policy Selected
7	Reconfiguring mission...
8	Going to static node: static_node01 Latitude: 30°02'13.56"S Longitude: 51°13'00.11"O
9	Arriving at static node: static_node01 Latitude: 30°02'13.56"S Longitude: 51°13'00.11"O
10	Getting data from command: MONITOR [TEMPERATURE]
11	Returning to last point: Latitude: 30°02'15.52"S Longitude: 51°13'03.50"O
12	Going to waypoint: way01 Latitude: 30°02'11.56"S Longitude: 51°13'07.43"O
13	Arriving at waypoint: way01 Latitude: 30°02'11.56"S Longitude: 51°13'07.43"O
14	Executing command: DETECT MOVEMENT AND DETECT FIRE

15	Ending mission...
16	Final position: Latitude: 30°02'11.56"S Longitude: 51°13'07.43"O
17	Time elapsed: 7032 s

Estes testes evidenciaram o correto funcionamento da interface de planejamento da MDL e controle de missão. A simulação, portanto, serve como exemplo de uma execução de uma missão real.

9 TRABALHOS RELACIONADOS

9.1 Sistema de Navegação para um Veículo Aéreo Não-Tripulado Voltado para o Cumprimento de Missões

Esta trabalho descreve um sistema de navegação em baixo nível para controle de veículos aéreos não-tripulados. A interface embarcada controla a navegação e os sensores em um VANT de modo que a aeronave funcione como um nó em uma rede de sensores sem fio (FONTOURA, A., 2011). O sistema é específico para o veículo de quato hélices da Skydrones. Este sistema seria utilizado em conjunto com o sistema de planejamento e controle de missão desenvolvido neste trabalho para controlar o VANT e sua colobaração na rede de sensores. A execução da missão configurada deveria ocorrer de forma autônoma exigindo a mínima intervenção humana.

9.2 Agent-based mission management for a UAV

Este trabalho descreve um design, implementação e teste de um sistema de gerenciamente de missões para um veículo aéreo não-tripulado de pequeno porte, o Codarra Avatar. O sistema foi desenvolvido utilizando-se o paradigma orientado a agentes para determinar um comportamento autônomo ao VANT. A linguagem de programação utilizada foi a JACK Intelligent Agents em conjunto com um sistema de auto-pilotagem e auto-estabilização já existente que realiza o controle básico de vôo. Este projeto foi desensolvido e testado em um teste de vôo, onde o VANT foi manualmente lançado e passado para o controle autônomo onde ele prosseguiu para os waypoints e realizou a missão com sucesso (KARIM, S.; HEINZE, C. e DUNN S., 2004). Este trabalho é importante, pois demonstra a aplicação de um sistema de gerenciamente de um VANT utilizando o paradigma orientado a agentes para se determinar um comportamento autônomo ao veículo para realização de uma missão. Este trabalho utiliza o mesmo conceito de atribuição de um comportamento orientado a agentes que o descrito nesta monografia, entretanto não inclui a utilização de nós sensores estáticos, o que é um diferencial visto o grande número de aplicações mencionadas nesta monografia.

9.3 Investigação de linguagem PDDL no planejamento de missões para robôs aéreos

Este trabalho descreve o planejamento automático de missões para um veículo aéreo não-tripulado utilizando a linguagem de planejamento Planning Domain Definition Language (PDDL). Os domínios desenvolvidos em PDDL foram executados em um simulador de vôo e comparados para se determinar as diferenças entre o que foi

planejado e executado durante uma missão (CANTONI L., CAMPOS M. e CHAIMOWICZ L., 2011). O veículo realiza a missão utilizando o conceito de waypoints e mostra outra forma de se realizar o planejamento de missões para VANTs. Este trabalho difere do planejamento descrito nesta monografia pelo fato de não utilizar o paradigma orientado a agentes, que como visto, é um fator diferencial importante para se atribuir um comportamento autônomo e colaborativo ao sistema. Além disso, a questão dos sensores estáticos não é tratada.

9.4 An Autonomous Autopilot Control System Design for Small-Scale UAVs

Este trabalho descreve o design e implementação de um sistema programável totalmente autônomo de auto-pilotagem para veículos aéreos não-tripulados de pequeno porte. O sistema foi implementado para utilizar a plataforma Exploration Aerial Vehicle (EAV) utilizada pela NASA. A EAV e a estação de solo foram construídos utilizando uma arquitetura baseada em componentes chamada Reflection Architecture. A Reflection Architecture é um protótipo de uma arquitetura para um sistema embarcado plug-and-play de tempo real que fornece a camada de transporte para comunicação em tempo real entre hardware e o software (IPPOLITO, C., 2005). Este trabalho mostra uma abordagem de um sistema de controle para auto-pilotagem de um VANT, o que difere do sistema desenvolvido neste trabalho, pois visa não apenas a auto-pilotagem, mas o controle autônomo completo de um VANT inserido como nó móvel em uma rede de sensores sem fio para a realização de missões. Entretanto, é interessante como exemplo de uma interface de controle de baixo nível necessária para a implementação do sistema de controle. Este design poderia ser facilmente adaptado para aceitar o planejamento descrito na MDL e assim executar missões de forma autônoma e não apenas a auto-pilotagem do VANT.

10 CONCLUSÃO

Neste trabalho, evidencia-se o grande potencial da utilização de veículos aéreos não-tripulados para a realização de diversas aplicações em redes de sensores sem fio. A utilização do paradigma orientado a agentes para definir um comportamento autônomo ao VANT se mostra bastante versátil e importante para o controle de missão. Para controlar as missões e tarefas do VANT utiliza-se uma interface de controle em alto nível que possibilita o controle de uma missão e a tomada de decisões ao recebimento de informações e alarmes. Para implementar este controle, utiliza-se uma interface que gera missões utilizando a linguagem MDL desenvolvida neste trabalho para descrever a missão em alto nível para a observação dos fenômenos de interesse e assim possibilitar a integração entre o VANT e os nós sensores componentes da rede para realizar as aplicações descritas neste texto.

Este projeto entretanto necessita a integração à uma interface de controle em baixo nível para testar o seu potencial e suas funcionalidades. O trabalho relacionado que deveria ser desenvolvido juntamente com este projeto sofreu atrasos e portanto não ficou pronto a tempo de integrar os dois sistemas. Como são trabalhos relacionados, mas independentes, foi possível dar continuidade a este projeto. A integração e os testes em missões reais ficam entretanto como trabalho futuro a ser desenvolvido. Para ao menos suprir a necessidade de teste do algoritmo de controle foi criada uma interface de simulação para visualizar o andamento de uma missão e o comportamente ao recebimento de alarmes.

Para finalizar, este trabalho foi de extrema importância no aprendizado do autor, que inicialmente possuía pouco conhecimento sobre a utilização de redes de sensores sem fio e veículos aéreos não-tripulados para a realização de aplicações. O conhecimento adquirido mostrou-se um adicional importante para a formação do autor que provavelmente será utilizado nas próximas etapas da sua vida profissional.

REFERÊNCIAS

ALONÇO, A. dos S. et al. **Desenvolvimento de um veículo aéreo não tripulado (VANT) para utilização em atividades inerentes à agricultura de precisão.** CONGRESSO BRASILEIRO DE ENGENHARIA AGRÍCOLA, 35., 2005, Canoas. Jaboticabal: Associação Brasileira de Engenharia Agrícola, 2005. 1 CD-ROM.

CANTONI L., CAMPOS M. e CHAIMOWICZ L. **Investigação da linguagem PDDL no planejamento de missões para robôs aéreos.** X SBAI – Simpósio Brasileiro de Automação Inteligente, São João del Rei, Minas Gerais. Brasil, 2011.

ERMAN, A. T.; HOESEL, L. e HAVINGA, P. **Enabling Mobility in Heterogeneous Wireless Sensor Networks Cooperating with UAVs for Mission- Critical Management**, 2008, IEEE Wireless Communications, vol. 15, is. 6, p. 38-46.

FERREIRA, A. M. **Cenário de Aplicação de Veículos Não Tripulados de Interesse de Defesa.** Workshop on Sensor Networks and Applications – WSENA. Gramado, Brasil, 2008.

FIPA**. FIPA Agent Management Specification**. Disponível em: http://www.fipa.org/specs/fipa00023/SC00023K.html#_Toc75950978

FONTOURA A., **Sistema de Navegação para um Veículo Aéreo Não Tripulado Voltado para o Cumprimento de Missões.** Trabalho de Conclusão de Curso (UFRGS), 2011.

FREITAS, E. P.; WEHRMEISTER, M. A.; PEREIRA, C. E.; LARSSON, T. **Real-time Support in Adaptable Middleware for Heterogeneous Sensor Networks**. Proc. of IEEE International Workshop on Real Time Software (RTS'08), 2008.

IPPOLITO, C. **An Autonomous Autopilot Control System Design for Small-Scale UAVs,** 2005. QSS Group, Inc. NASA Ames Research Center. Disponível em: http://cmil.west.cmu.edu/publications/EAV-20051016 20UAV 20autopilot 20design.pdf

KARIM, S.; HEINZE, C. e DUNN S. **Agent-based mission management for a UAV**. 2004. International Conference on Intelligent Sensors, Sensor Networks and Information Processing, 2004, p. 481-486.

Mikrokopter. Disponível em: http://www.mikrokopter.de

OPEN GEOSPATIAL CONSORTIUM INC. **OpenGIS Sensor Model Language (SensorML) Implementation Specification**, 2007. Disponível em: <http://www.ogcnetwork.net/SensorML>.

Oracle Labs. **SunSPOT**. Disponível em: http://www.sunspotworld.com/

Skydrones. Disponível em: http://www.polibiobraga.com.br/skydrones.pdf

Sun Microsystems. **MVC: Model, Viewing and Controller**. http://java.sun.com/blueprints/patterns/MVC-detailed.html

WEISS, G. **Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence**. MIT Press, 1999.

WOOLDRIDGE, M. **An Introduction to Multiagent Systems,** 2002**.** West Sussex. John Wiley. p. 348.

ANEXO A <MODEL, VIEW AND CONTROLLER>

A arquitetura MVC foi introduzida por Trygve Reenskaug, um desenvolvedor de Smalltalk que trabalhava no Xerox Palo Alto Research Center em 1979. Esta arquitetura ajuda a dissociar o acesso aos dados e a lógica da interface de apresentação ao usuário. Mais precisamente a arquitetura MVC pode ser particionada em três elementos.

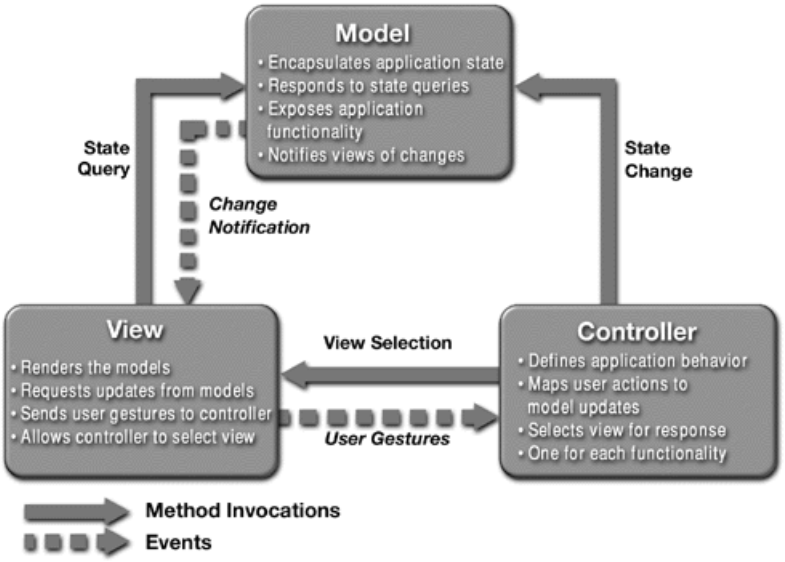


Figura Anexo A: Diagrama em blocos da arquitetura MVC

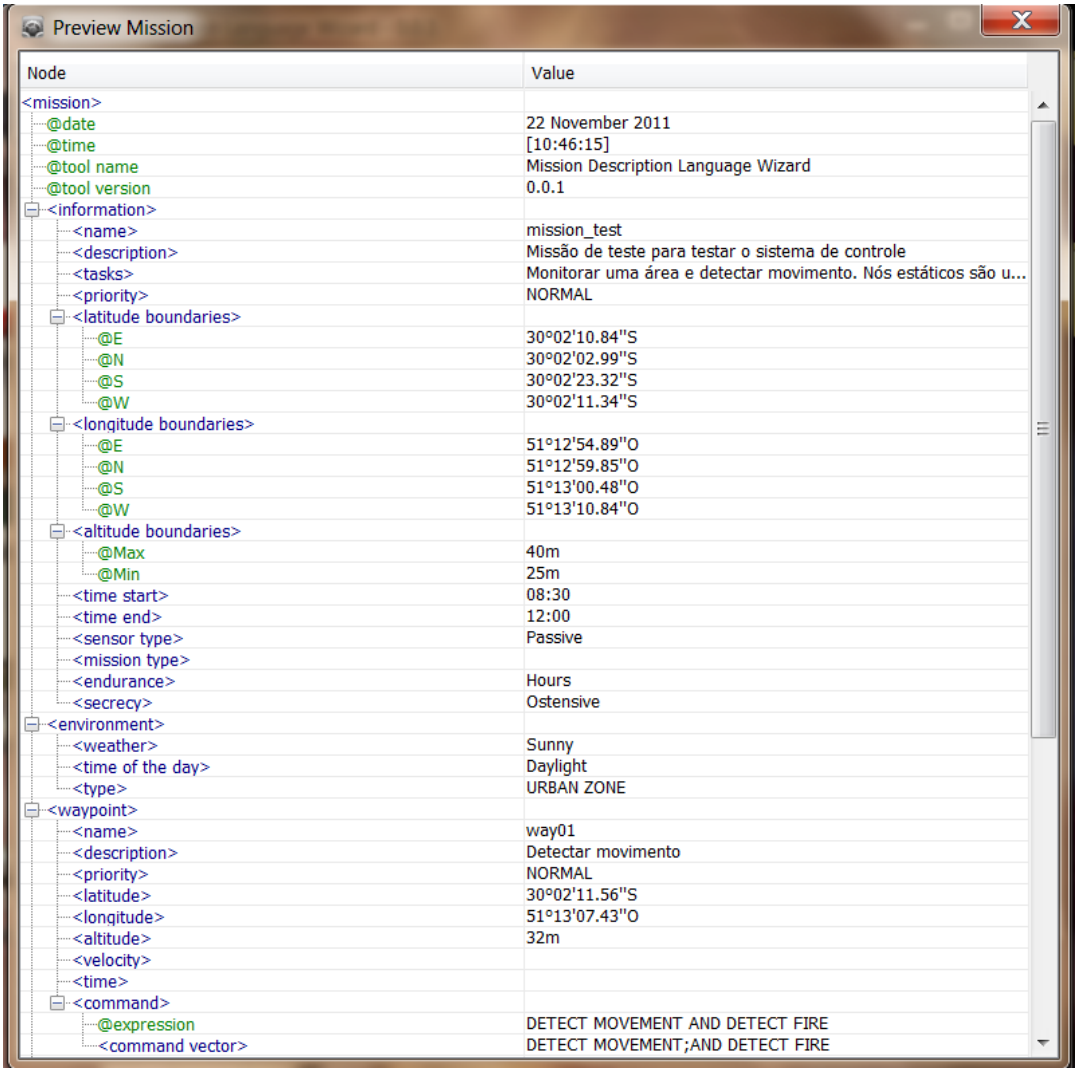
Model – O modelo representa os dados e as regras que definem o acesso e atualização destes dados. Em software corporativo, o modelo serve como uma aproximação do software de um processo do mundo real.

View – O "view" representa a interface gráfica do software. Ele especifica como os dados do modelo devem ser apresentados. Se os dados do modelo são alterados o "view" deve ser atualizado. Em Java, a thread principal é responsável pela atualização da interface gráfica, e portanto é desejável que a aplicação seja baseada em threads que executem as funções do programa para não haver problemas de atualização da interface gráfica.

Controller – O controlador traduz as iterações do usuário com a interface gráfica em ações que realizam modificações no modelo.

APÊNDICE A<ARQUIVO XML DE DESCRIÇÃO DE UMA MISSÃO GERADO PELA INTERFACE MDL>

Este apêndice descreve a missão gerada pela interface MDL e sua visualização na mesma. Esta missão fictícia foi utilizada para testar o funcionamento da interface de planejamento, da interface de interpretação e da execução do algoritmo de controle.



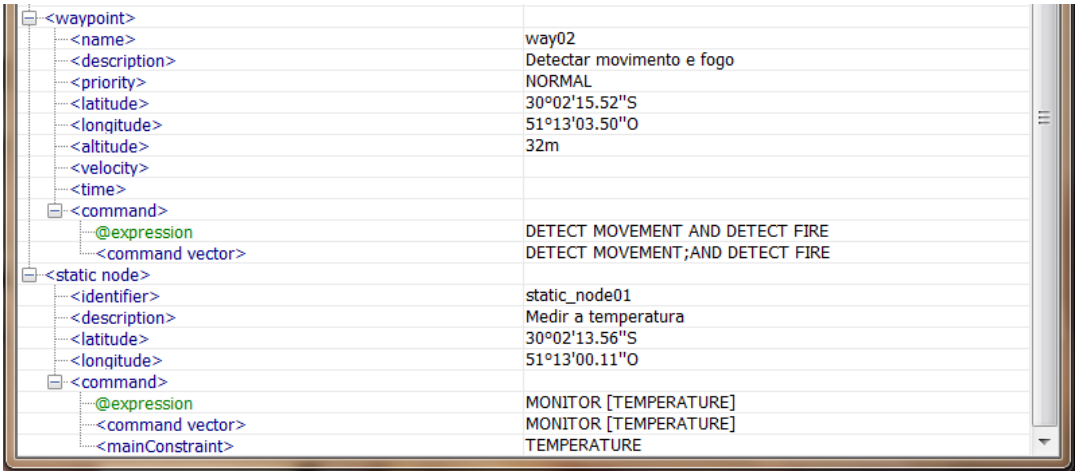


Figura Apêndice A.1: Missão visualizada na interface.

Esta figura exibe a missão utilizada na simulação visualizada na interface.

O código XML para esta missão é descrito abaixo:

<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>

<mission date="22 November 2011" time="[10:46:15]" tool_name="Mission Description Language Wizard" tool_version="0.0.1">

<information>

<name>mission_test</name>

<description>Missão de teste para testar o sistema de controle</description>

<tasks>Monitorar uma área e detectar movimento. Nós estáticos são utilizados para medir a temperatura.</tasks>

<priority>NORMAL</priority>

<latitude_boundaries E="30°02'10.84"S" N="30°02'02.99"S" S="30°02'23.32"S" W="30°02'11.34"S"/>

<longitude_boundaries E="51°12'54.89"O" N="51°12'59.85"O" S="51°13'00.48"O" W="51°13'10.84"O"/>

<altitude_boundaries Max="40m" Min="25m"/>

<time_start>08:30</time_start>

<time_end>12:00</time_end>

<sensor_type>Passive</sensor_type>

<mission_type/>

<endurance>Hours</endurance>

<secrecy>Ostensive</secrecy>

</information>

<environment>

<weather>Sunny</weather>

<time_of_the_day>Daylight</time_of_the_day>

```
        <type>URBAN ZONE</type>
      </environment>
      <waypoint>
        <name>way01</name>
        <description>Detectar movimento</description>
        <priority>NORMAL</priority>
        <latitude>30°02'11.56"S</latitude>
        <longitude>51°13'07.43"O</longitude>
        <altitude>32m</altitude>
        <velocity/>
        <time/>
        <command expression="DETECT MOVEMENT AND DETECT FIRE">
          <command_vector>DETECT MOVEMENT;AND DETECT
FIRE</command_vector>
        </command>
      </waypoint>
      <waypoint>
        <name>way02</name>
        <description>Detectar movimento e fogo</description>
        <priority>NORMAL</priority>
        <latitude>30°02'15.52"S</latitude>
        <longitude>51°13'03.50"O</longitude>
        <altitude>32m</altitude>
        <velocity/>
        <time/>
        <command expression="DETECT MOVEMENT AND DETECT FIRE">
          <command_vector>DETECT MOVEMENT;AND DETECT
FIRE</command_vector>
        </command>
      </waypoint>
      <static_node>
        <identifier>static_node01</identifier>
        <description>Medir a temperatura</description>
        <latitude>30°02'13.56"S</latitude>
        <longitude>51°13'00.11"O</longitude>
```

```
        <command expression="MONITOR [TEMPERATURE]">
            <command_vector>MONITOR [TEMPERATURE]</command_vector>
            <mainConstraint>TEMPERATURE</mainConstraint>
        </command>
    </static_node>
</mission>
```

APÊNDICE B<CÓDIGO COMPLETO DO ALGORTIMO DE CONTROLE DE MISSÃO>

```
package kernel;
import gui.console.SystemConsole;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.Map.Entry;
import data.StaticNode;
import data.StaticNodeMap;
import data.Waypoint;
import data.WaypointMap;

public class Mission {

    private static final String ABORT = "ABORT";
    private String endLat;
    private String endLong;
    private String currentPosLat;
    private String currentPosLong;
    private WaypointMap waypointMap;
    private StaticNodeMap staticNodeMap;
    private int timeStep = 1000;
    private String alarm;
    private Waypoint waypoint;

    public Mission() {
```

```
            this.alarm = "";
        }

        public void setEndLat(String endLat) {
            this.endLat = endLat;
        }
        public String getEndLat() {
            return endLat;
        }
        public void setEndLong(String endLong) {
            this.endLong = endLong;
        }
        public String getEndLong() {
            return endLong;
        }
        public void setWaypointMap(WaypointMap waypointMap) {
            this.waypointMap = waypointMap;
        }
        public WaypointMap getWaypointMap() {
            return waypointMap;
        }
        public void setStaticNodeMap(StaticNodeMap staticNodeMap) {
            this.staticNodeMap = staticNodeMap;
        }
        public StaticNodeMap getStaticNodeMap() {
            return staticNodeMap;
        }

        private float[] coordinate(String coord) {
            float[] ret = new float[3];
            String graus = coord.substring(0, coord.indexOf("°"));
            String minutos = coord.substring(coord.indexOf("°") + 1,
coord.indexOf("'"));
            String segundos = coord.substring(coord.indexOf("'") + 1,
coord.indexOf("'"));
```

```
                ret[0] = Float.valueOf(graus);
                ret[1] = Float.valueOf(minutos);
                ret[2] = Float.valueOf(segundos);
                return ret;
        }

        private float[] calculateDistance(float[] p1x, float[] p2x, float[] p1y, float[] p2y)
{
                float[] ret = new float[3];
                ret[0] = (float) Math.sqrt((p2x[0] - p1x[0])*(p2x[0] - p1x[0])+(p2y[0] -
p1y[0])*(p2y[0] - p1y[0]));
                ret[1] = (float) Math.sqrt((p2x[1] - p1x[1])*(p2x[1] - p1x[1])+(p2y[1] -
p1y[1])*(p2y[1] - p1y[1]));;
                ret[2] = (float) Math.sqrt((p2x[2] - p1x[2])*(p2x[2] - p1x[2])+(p2y[2] -
p1y[2])*(p2y[2] - p1y[2]));;
                return ret;
        }

        /*
         * Compare if d is greater than dOld and then return true
         */
        private boolean compare(float[] d, float[] dOld) {
                if (d[0] > dOld[0]) {
                        if (d[0] == dOld[0]) {
                                if (d[1] > dOld[1]) {
                                        if (d[1] == dOld[1]) {
                                                if (d[2] > dOld[2]) {
                                                        return true;
                                                } else {
                                                        return false;
                                                }
                                        } else {
                                                return true;
                                        }
                                } else {
                                        return false;
```

```
                }
            } else {
                return true;
            }
        } else {
            return false;
        }
    }

    public Waypoint getNextWaypoint(String currentLat, String currentLong) {
        if (waypointMap.getWaypointMap().isEmpty()) {
            return null;
        }
        Set<Entry<String, Waypoint>> enumeration = waypointMap.getWaypointMap().entrySet();
        Iterator<Entry<String, Waypoint>> iterator = enumeration.iterator();
        Waypoint w = null;
        float[] dOld = null;
        while (iterator.hasNext()) {
            Map.Entry<String, Waypoint> entry = iterator.next();
            float[] currentPosLat = coordinate(currentLat);
            float[] currentPosLong = coordinate(currentLong);
            float[] wayLat = coordinate(entry.getValue().getLatitude());
            float[] wayLong = coordinate(entry.getValue().getLongitude());
            float[] d = calculateDistance(currentPosLat, wayLat, currentPosLong, wayLong);
            if (dOld == null) {
                //first distance
                w = entry.getValue();
                dOld = d;
            } else {
                if (!compare(d, dOld)) { //d > dOld
                    w = entry.getValue();
                    dOld = d;
                }
```

```
                }
            }
            //waypoint selected then remove waypoint from map
            waypointMap.delWaypoint(w.getName());
            return w;
        }

        public void observe(String alarm) {
            String currentLat = getCurrentPosLat();
            String currentLong = getCurrentPosLong();
            WaypointMap wMap = getWaypointMap();
            String environmentData = getEnvironmentData();
            this.alarm = alarm;
            orient(currentLat, currentLong, wMap, environmentData);
        }

        private void orient(String currentLat, String currentLong, WaypointMap wMap,
String envData) {
            this.waypointMap = wMap;
            Waypoint w = getNextWaypoint(currentLat, currentLong);
            decide(w, envData);
        }

        private boolean checkPriority(String alarm) {
            if (!alarm.equals("")) {
                return true;
            } else {
                return false;
            }
        }

        private void decide(Waypoint w, String envData) {
            if (checkWeatherCondition(envData)) {
                if (checkPriority(this.alarm)) {
                    StaticNode sn = getStaticNode(alarm);
```

```
                        actAlarm(sn);
                    } else {
                        actWaypoint(w);
                    }
                } else {
                    abortMission();
                }
            }

            private void abortMission() {
                // Abort Mission
                sendCommand(ABORT);
            }

            private void sendCommand(String com) {
                // Send command
            }

            private String getEnvironmentData() {
                return "";
            }

            private void resetAlarm() {
                alarm = "";
            }

            private void actWaypoint(Waypoint w) {
                SystemConsole.append("Going to waypoint: " + w.getName() + "
Latitude: " +
                                     w.getLatitude() +
                                    " Longitude: " + w.getLongitude());
                simulationStep(getTimeStep());
                SystemConsole.append("Arriving at waypoint: " + w.getName() + "
Latitude: " + w.getLatitude() +
                                                    " Longitude: " + w.getLongitude());
```

```
                SystemConsole.append("Executing command: " + w.getExpression());
                simulationStep(w.getWaypointTime() * getTimeStep());
                setCurrentPosLat(w.getLatitude());
                setCurrentPosLong(w.getLongitude());
        }

        private void actAlarm(StaticNode sn) {
                SystemConsole.append("ALARM RECEIVED: Priority Policy
Selected");
                SystemConsole.append("Reconfiguring mission...");
                SystemConsole.append("Going to static node: " + sn.getIdentifier() +
                                " Latitude: " + sn.getLatitude() +
                                " Longitude: " + sn.getLongitude());
                simulationStep(2 * getTimeStep());
                SystemConsole.append("Arriving at static node: " + sn.getIdentifier() +
                                " Latitude: " + sn.getLatitude() +
                                " Longitude: " + sn.getLongitude());
                SystemConsole.append("Getting data from command: " +
sn.getExpression());
                simulationStep(getTimeStep());
                SystemConsole.append("Returning to last point: Latitude: " +
getCurrentPosLat() +
                                " Longitude: " + getCurrentPosLong());
                resetAlarm();
        }

       public void simulationStep(int step) {
         try {
                        Thread.sleep(step);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
       }

         private boolean checkWeatherCondition(String envData) {
```

```
                // Check sensors for weather condition
                return true;
        }

        public void setCurrentPosLat(String currentPosLat) {
                this.currentPosLat = currentPosLat;
        }
        public String getCurrentPosLat() {
                return currentPosLat;
        }
        public void setCurrentPosLong(String currentPosLong) {
                this.currentPosLong = currentPosLong;
        }
        public String getCurrentPosLong() {
                return currentPosLong;
        }
        public void setTimeStep(String timeStep) {
                if (timeStep.equals("")) {
                        this.timeStep = 1000;
                        SystemConsole.append("Defalut time step: " +
Integer.valueOf(this.timeStep));
                } else {
                        this.timeStep = Integer.valueOf(timeStep);
                }
        }
        public int getTimeStep() {
                return timeStep;
        }
        public boolean isWaypointMapEmpty() {
                return waypointMap.getWaypointMap().isEmpty();
        }
        public StaticNode getStaticNode(String alarm) {
                return staticNodeMap.getStaticNode(alarm);
        }
        public void setAlarm(String alarm) {
```

```
            this.alarm = alarm;
    }
    public String getAlarm() {
            return alarm;
    }

    public void setWaypoint(Waypoint waypoint) {
            this.waypoint = waypoint;
    }

    public Waypoint getWaypoint() {
            return waypoint;
    }
}
```