Diogo Filipe Granja Bernardino

Licenciado em Engenharia Informática

Framework-specific DSL para Sensoriamento Participado

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadora : Profª. Maria Cecília, Prof. Auxiliar, Universidade Nova de Lisboa

Co-orientador : Prof. Sérgio Duarte, Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Prof. Doutor Manuel João Toscano Próspero dos Santos

Arguente: Prof. Doutor Alberto Manuel Rodrigues da Silva

Vogal: Profª. Doutora Maria Cecília Farias Lorga Gomes

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2011

***Framework-specific DSL* para Sensoriamento Participado**

À minha Avó.

Agradecimentos

Em primeiro lugar quero agradecer aos meus orientadores, Professora Doutora Maria Cecília e Professor Doutor Sérgio Duarte, pelo papel imprescindível na orientação, disponibilidade, apoio, assim como todos os contributos que permitiram o desenvolvimento desta dissertação.

Os meus agradecimentos dirigem-se também para:

- o Professor Vasco Amaral pelos esclarecimentos de técnicas no desenvolvimento da dissertação;
- todos os meus colegas de faculdade, em particular, Samuel Del Bello e Filipe Gonçalves;
- toda a minha família e amigos pelo suporte incondicional, em especial aos meus pais, Fernando Manuel Nunes Bernardino e Ana Maria Gomes Granja Bernardino.

A todos,
Muito Obrigado!

Resumo

Como suporte ao desenvolvimento ágil de aplicações numa determinada área, tem-se verificado uma grande importância no uso de *DSLs* (*domain-specific languages*). A utilização de abstrações/conceitos específicos de um domínio de aplicação particular faz com que o programador tenha apenas de se focar na lógica da aplicação e não nos detalhes da infraestrutura de suporte.

Por outro lado, *sensoriamento participado* consiste numa área emergente da computação móvel. Esta explora os recentes avanços tecnológicos dos dispositivos móveis pessoais criando redes de sensores móveis avançadas. Com a mobilidade dos utilizadores, podem-se criar aplicações móveis, sem os custos associados à implantação de uma infraestrutura de rede de sensores densa e extensa. Assim, *sensoriamento participado* apresenta ser uma área promissora para o desenvolvimento de aplicações muito relevantes no futuro, que pode ser facilitado com o auxílio de uma DSL.

Esta dissertação trata o desenvolvimento de uma *DSL gráfica* específica para um framework neste domínio de aplicações, tendo partido do levantamento das abstrações e conceitos usados na plataforma designada, *4Sensing*.

O resultado final desta proposta é a possibilidade de desenvolver aplicações em *sensoriamento participado* com base em algumas abstrações da *framework 4Sensing*, através de um editor gráfico. Este gera código de integração a um simulador já desenvolvido, permitindo assim a avaliação das aplicações em ambiente simulado neste contexto particular.

Palavras-chave: Sensoriamento Participado, Linguagem de domínio específico, DSL, Geradores de código, 4Sensing

Abstract

DSLs (*domain-specific languages*) are specially important when used to support agile application development. The usage of specific abstractions/concepts in a particular application domain leads the developer to focus mainly on application logic forgetting structure details.
In other hand, *participatory sensing* is an emergent area of mobile computation due to recently advances of technology in personal mobile devices, creating advance mobile sensors network. With user mobility, one can create interesting mobile applications, without the costs associated to large implementations of density mobile sensor networks. Thus, *participatory sensing* paradigm presents as one of the most promising and innovative areas in future application development.
This dissertation focuses in the development of a *graphic framework-specific DSL* in *participatory sensing*, starting from the analysis of abstractions and concepts used at *4Sensing* platform. The final result of this proposal is the possibility of developing *participatory sensing* applications with *4Sensing framework's* abstractions, with a graphical editor. This editor should generate code to integrate in a previously developed simulator, allowing the evaluation of applications in a simulated environment.

Keywords: Participatory Sensing, Domain-Specific Language, DSL, Framework-specific DSL, 4Sensing, Generative Programming

Conteúdo

1 Introdução **1**

1.1 Objectivo do trabalho . . . 3
1.2 Contribuições . . . 4
1.3 Estrutura do Documento . . . 4

2 Trabalho relacionado **7**

2.1 Sensoriamento Participado . . . 7
2.2 Aplicações em Sensoriamento Participado . . . 9
2.2.1 The Pothole Patrol . . . 9
2.2.2 Cartel . . . 10
2.2.3 BikeNet . . . 11
2.2.4 CenceMe . . . 12
2.3 Infraestruturas de suporte a Sensoriamento Participado . . . 12
2.3.1 4Sensing . . . 13
2.4 Generative Software Development . . . 15
2.4.1 Engenharia do Domínio e Engenharia de Aplicação . . . 15
2.4.2 Mapeamento entre o Espaço de Problema e o Espaço de Solução . . . 16
2.4.3 Diagrama de características . . . 16
2.5 Domain-Specific Languages - DSLs . . . 17
2.5.1 Características Distintivas das DSLs . . . 17
2.5.2 Vantagens e Desvantagens . . . 18
2.5.3 Fases de Desenvolvimento de uma DSL . . . 18
2.5.4 DSLs em sensoriamento participado . . . 20
2.6 Framework-Specific Modeling Languages - FSML . . . 21
2.7 Ferramentas . . . 22
2.7.1 LabVIEW . . . 22
2.7.2 EMF - Eclipse Modeling Framework . . . 24
2.7.3 GMF - Graphical Modeling Framework . . . 24

3 Abstrações do domínio para uma DSL **27**
3.1 *Diagrama de características* . . . 28
3.2 Arquitetura . . . 30
3.2.1 Fontes de dados . . . 30
3.2.2 Dados . . . 32
3.2.3 Manipulação de dados . . . 33
3.3 Operadores . . . 34
3.3.1 GroupBy . . . 34
3.3.2 Classifier . . . 35
3.3.3 Set . . . 35
3.3.4 Processor . . . 35
3.3.5 Filter . . . 35
3.3.6 TimeWindow . . . 35
3.3.7 Aggregator . . . 35
3.4 Metadata: DSL vs 4Sensing . . . 36
3.5 Sumário . . . 36

4 Framework-specific DSL **39**
4.1 Visão geral da solução proposta . . . 39
4.2 Modelo Ecore . . . 40
4.2.1 Fontes e Manipulação de dados . . . 41
4.2.2 Aplicação . . . 42
4.2.3 Transições . . . 43
4.3 Editor . . . 44
4.3.1 Tela . . . 45
4.3.2 Barra de ferramentas . . . 46
4.3.3 Normas de desenvolvimento . . . 47
4.4 Inserção de código . . . 48
4.5 Comparação e coerência entre *inputs* e *outputs* de componentes . . . 49
4.5.1 Métodos de comparação . . . 51
4.6 Ligação Framework/Simulador 4Sensing . . . 53
4.7 Contexto da Simulação . . . 54
4.7.1 Configuração . . . 54
4.7.2 *Outputs* . . . 55
4.8 Limitações . . . 56
4.8.1 Operador *aggregate* . . . 56
4.8.2 Condições . . . 57
4.8.3 Criação e seleção de tuplos . . . 57
4.9 Sumário . . . 57

5 Validação **59**
5.1 Cenário SpeedSense 59
5.2 Aplicação 1 60
5.2.1 TrafficCount 60
5.3 Aplicação 2 62
5.3.1 TrafficSpeed 62
5.4 Sumário 64

6 Conclusões **67**
6.1 Contribuições 68
6.2 Trabalho futuro 68
6.2.1 Importar tabelas virtuais 68
6.2.2 Criação de sensores 69
6.2.3 Interface 69

7 Diagramas **75**

Lista de Figuras

2.1 Arquitectura da aplicação The Pothole Patrol [14] 9
2.2 Arquitectura da aplicação Cartel [19] 10
2.3 Arquitectura da aplicação BikeNet [13] 11
2.4 Arquitectura 4Sensing [15] 13
2.5 Principais processos a ter em conta em Generative Software. Imagem adaptada a partir de [7] 15
2.6 Mapeamento entre o espaço do problema e o espaço de solução 16

3.1 Diagrama de características - Visão geral das 5 características do domínio de *sensoriamento participado*. 28
3.2 Diagrama de características - Elemento dos dados. 29
3.3 Diagrama de características - Elemento da aquisição de dados. 29
3.4 Diagrama de características - Elemento do processamento dos dados. . . . 30
3.5 Diagrama de características - Elemento da distribuição dos dados. 30
3.6 Diagrama de características - Elemento da interação dos dados. 31
3.7 Visão geral dos 3 grupos da arquitectura da DSL 31
3.8 UML das fontes de dados. 32

4.1 Representação da solução proposta 40
4.2 Diagrama do modelo *ecore* - classe principal 41
4.3 Diagrama do modelo *ecore* - sensor e tabela virtual 41
4.4 Diagrama do modelo *ecore* - pipeline e operador 42
4.5 Diagrama do modelo *ecore* - aplicação 42
4.6 Diagrama do modelo *ecore* - transições 43
4.7 Diagrama do modelo *ecore* - transição entre operadores 43
4.8 Diagrama do modelo *ecore* - discriminação dos tipos referentes aos atributos dos vários elementos do diagrama 44
4.9 Visão global do editor. 45
4.10 Representação da tela e seus respectivos elementos. 46

4.11 Representação da barra de ferramentas e seus respectivos elementos. . . . 47
4.12 Exemplo de interface para definir quais os tuplos que constituem o *input/output* duma tabela virtual. 50
4.13 Exemplo de aviso - transição a amarelo - de *input* não utilizado pela tabela virtual ao longo dos seus pipelines. 50
4.14 Exemplos de comparação direta. 51
4.15 Exemplos de comparação interna. 52
4.16 Esquema da ligação framework-simulador. 53
4.17 Visualização da interface de escolha de operações a executar pelo operador *aggregrate*. 56

5.1 Ficheiro .psfd da Aplicação1. 60
5.2 Ficheiro .psfi da Aplicação1. 60
5.3 Ficheiro .psfd da Aplicação2. 62
5.4 Ficheiro .psfi da Aplicação2. 63
5.5 Avisos de validação. 65
5.6 Visualização do simulador 4Sensing em execução. 66

7.1 *Diagrama de características*, completo, no domínio de *sensoriamento participado* 76
7.2 Diagrama completo da arquitetura desenvolvida para a DSL 77
7.3 Diagrama completo do modelo *.ecore* 78

Lista de Tabelas

3.1 Comparação entre características 4Sensing e DSL em *sensoriamento participado* 36

4.1 Parâmetros a configurar 55

Listings

5.1 Código da tabela *TrafficCount*. 61
5.2 Código da tabela *TrafficSpeed*. 64

1

Introdução

As primeiras aplicações de redes de sensores basearam-se na resolução de problemas de pequena escala em domínios científicos e industriais, como por exemplo a monitorização de florestas. A miniaturização dos sensores e o seu baixo custo, levou à introdução de sensores em dispositivos electrónicos populares, como telemóveis, PDAs e leitores mp3 [5]. A capacidade de computação móvel e de monitorização passou a fazer parte do nosso dia-a-dia, andando na nossa mochila, bolsa ou bolso.

Há poucos anos atrás, apenas víamos o telemóvel como forma de comunicar, sendo utilizado intencionalmente e esporadicamente. No entanto essa ideia mudou, e os telemóveis são também vistos como sensores móveis capazes de recolher, partilhar e processar informação silenciosamente e passivamente, durante todo o dia [6]. São inúmeros os dados que estes conseguem captar, desde som, imagens, movimento, localizações, etc, através de sensores como GPS, acelerómetro, câmera, etc. De igual forma, vieram influenciar os sistemas operativos que têm sido desenvolvidos para estes telemóveis, nomeadamente o sistema *iOS* e *Android* [1]. Sistemas operativos esses, com plataformas de auxílio no desenvolvimento de aplicações, suscitando a curiosidade e a motivação de muitos utilizadores na exploração das capacidades destes dispositivos. Por sua vez, surge também a facilidade de partilhar estas aplicações através de repositórios como a *Apple AppStore* e o *Android Market* [22, 3, 2].

Tais sensores móveis, capazes de capturar, classificar e transmitir dados, abriram uma nova porta a um elevado número de aplicações, capazes de nos dar acesso a informação relativa a grandes áreas geográficas e a custos reduzidos. Em comparação com uma rede de sensores estática, uma rede de dispositivos móveis poderá abranger um maior espaço geográfico utilizando menos recursos, deixando de parte elevados custos de implementação e manutenção.

O conceito de *sensoriamento participado* explora a partilha da informação capturada, feita conscientemente a partir de utilizadores participantes numa rede de dispositivos móveis [21]. Um sistema de *sensoriamento participado* é composto por aplicações móveis e web que assistem os utilizadores na *captura*, *interpretação* e *publicação* de informação extraída dos sensores, gerando informação de carácter diferenciado e incentivando a participação de todos. Este é um domínio promissor dado a importância que a informação extraída pode ter em várias áreas, desde política, marketing, desporto, monitorização ambiental, entre muitas outras. Podemos imaginar, por exemplo que, uma marca de refrigerantes decide desenvolver uma aplicação, oferecendo certas regalias aos utilizadores participantes no sistema, fazendo a monitorização das zonas mais movimentadas de Lisboa através de sensores GPS presentes nos telemóveis de cada utilizador. Esta informação pode ser útil caso a marca pretenda executar uma manobra de publicidade acerca de um novo produto a ser lançado no mercado, garantindo um maior número de público alvo.

As aplicações deste domínio podem ser classificadas em três grupos: *pessoal*, relativa a uma monitorização pessoal da atividade de cada utilizador; *social*, onde existe a partilha de informação entre grupos sociais consoante os interesses de cada utilizador; e *pública*, onde a partilha de informação é feita com todos. Cada um destes grupos possui os seus desafios, existindo ainda muito por explorar na forma como os dados são capturados, analisados, visualizados e partilhados com outros [5].

São já várias as aplicações com base neste conceito, como por exemplo a aplicação *Cartel* e a aplicação *CenceMe*. *Cartel* é uma aplicação que se destina à monitorização do tráfego das estradas através de dispositivos móveis colocados nos automóveis. Captar informação como a velocidade, a aceleração e a localização de vários automóveis, possibilita a visualização do tráfego nas estradas, numa determinada zona. Essa informação pode ser visualizada pelos utilizadores do sistema com o auxílio de mapas digitais [19]. A aplicação *CenceMe*, com um carácter mais social, permite reconhecer certas ações de um utilizador (ex. andar, conversar, sentar, etc) através dos sensores do seu telemóvel, partilhando essa informação de forma automática em diversas redes sociais (ex. *Facebook*, *MySpace*) [25].

Embora o número de aplicações com base neste conceito tenha vindo a aumentar, o seu desenvolvimento é ainda limitado sobretudo a pessoas com pouco conhecimento na área da informática pela relativa complexidade das interfaces típicas de programação neste domínio. Dado que estes sistemas geram informação relevante em várias áreas científicas e da engenharia, e não apenas dentro do domínio da informática, é importante proporcionar formas simples de desenvolvimento de aplicações em *sensoriamento participado*. É importante que pessoas de várias áreas profissionais tenham a oportunidade de avançar com as suas ideias e de criar aplicações que se ajustem às suas necessidades, sem que para isso tenham de possuir grandes conhecimentos a nível de desenvolvimento de software ou recorrer a profissionais nesta área. Preferencialmente, devem poder usar

conceitos que lhes sejam familiares e facilmente entendidos por outros profissionais na mesma área.
Este objectivo pode ser alcançado por intermédio de linguagens e interfaces textuais ou gráficas mais simples, com componentes que permitam assim ao utilizador expressar-se de forma natural através de elementos do domínio (ex. sensores, características geográficas ou sociais).

Surge, desta forma, a importância do desenvolvimento de uma DSL (*domain-specific language*) neste domínio de *sensoriamento participado*. Uma DSL é uma linguagem desenvolvida para um domínio de aplicação específico, aproximando o programador das respectivas componentes e abstrações do domínio, possibilitando uma implementação mais fácil e rápida de uma aplicação. Assim, os utilizadores podem desenvolver aplicações usando uma linguagem que lhes é familiar, se esta lhes disponibilizar elementos base correspondentes a conceitos conhecidos nesse domínio [23].

A criação de uma DSL em *sensoriamento participado* aparenta ser vantajosa dado que captura e permite a reutilização de abstrações, componentes e funcionalidades bem definidas e relevantes neste domínio. É frequente a implementação repetida de um mesmo processamento de dados (algoritmos) em diferentes aplicações. O reconhecimento de ações do quotidiano, performance de um utilizador, características ambientais, são exemplos de informação necessária comum a muitas aplicações. Uma vez implementada uma DSL, apenas é necessário *reutilizar* uma solução já existente acelerando o processo de implementação da aplicação. Sendo importante que esta DSL afecte a qualidade de desenvolvimento de aplicações a utilizadores com conhecimentos básicos informáticos, a disponibilização de uma DSL gráfica facilitará a sua aprendizagem.

Concluindo, o desenvolvimento de uma DSL no domínio *sensoriamento participado* faz com que o desenvolvimento de aplicações seja possível e mais simplificado a utilizadores presentes em diferentes áreas profissionais. Para mais, sendo esse desenvolvimento executado de forma fácil e rápida, e levando a um aumento na quantidade e qualidade de informação, relativa a vários interesses sociais (política, saúde, desporto, arte, etc), bem como a disseminação da informação. Esta dissertação pretende portanto contribuir para uma DSL em sensoriamento participado focando numa das suas dimensões, nomeadamente o processamento dos dados. Para tal é necessário realizar a extração e modelação de elementos relevantes nesse domínio e, ao mesmo tempo, desenvolver um protótipo que permita demonstrar as capacidades da DSL gráfica definida.

1.1 Objectivo do trabalho

O desenvolvimento de componentes e implementação de todas as abstrações extraídas das aplicações em *sensoriamento participado* estudadas, levaria a um trabalho extremamente complexo e de longo prazo, não sendo um projeto possível no espaço de tempo disponível para a elaboração desta dissertação. Como tal, o objectivo deste trabalho focase numa *DSL gráfica* específica para uma framework na área de *sensoriamento participado*.

Este é um trabalho, no âmbito de um anterior projeto, *framework 4Sensing*, que consiste numa plataforma focada no processamento de dados para aplicações em *sensoriamento participado*. Esta plataforma desenvolvida no projeto *4Sensing* irá servir de base para a simulação das aplicações criadas pela DSL gráfica (simulador *4Sensing*). Ao longo do projecto *4Sensing* e, como forma de auxílio à sua arquitectura, foram também pensadas abstrações relativamente à manipulação dos dados do sistema. Desta forma, o objectivo desta dissertação passa também por outras duas contribuições: identificação de abstrações numa dimensão particular de sensoriamento participado e, extração de abstrações específicas da plataforma 4Sensing, unindo as duas sob a forma de uma DSL gráfica.

Implementada a *framework-specific DSL gráfica* centrada no processamento dos dados, avaliar-se-á a usabilidade dos elementos que a compõem, seguindo-se uma fase de teste e validação dos seus resultados. Serão criados alguns exemplos de aplicação possíveis, ilustrando o desenvolvimento e o funcionamento das mesmas, seguindo-se um conjunto de execuções e avaliação da prestação de cada uma delas.

Uma continuação deste trabalho permitirá uma DSL mais abrangente para o domínio de *sensoriamento participado*, com a vantagem de ser expressa através de uma framework gráfica.

1.2 Contribuições

- Identificação e modelação dos conceitos e abstrações relevantes no domínio de *sensoriamento participado*
- Extração de abstrações específicas da plataforma *4Sensing*
- *DSL gráfica* específica para a framework *4Sensing* e respectiva integração com a mesma
- Validação da *framework-specific DSL gráfica* através de exemplos de aplicações

1.3 Estrutura do Documento

Neste primeiro capítulo é dada uma introdução ao tema desta dissertação tendo em conta a sua motivação e objectivo, e uma solução proposta para a resolução do problema. A restante estrutura deste documento está organizada da seguinte forma:

- **Capítulo2** dedicado a uma apresentação detalhada dos vários conceitos necessários a esta dissertação;
- **Capítulo3** identifica e estrutura abstrações no domínio de *sensoriamento participado* para uma futura DSL;
- **Capítulo4** processo de desenvolvimento da *framework-specific DSL*;

- **Capítulo5** teste e validação da DSL gráfica a partir de exemplos de aplicações em *sensoriamento participado*;
- **Capítulo6** apresenta conclusões e trabalho futuro relativamente ao trabalho alcançado.

2

Trabalho relacionado

Este capítulo descreve qual a informação importante para o trabalho desenvolvido nesta dissertação. A investigação feita assenta principalmente sobre o conceito *sensoriamento participado* e o desenvolvimento e importância de uma DSL neste domínio (*domain-specific language*), disponibilizada graficamente. É também feito referência às ferramentas tidas em conta na resolução deste trabalho, e que foi necessário compreender/aprender.

Na área de *sensoriamento participado*, para além da sua importância atual e futura, foram analisadas as funcionalidades de algumas aplicações neste domínio (ex. *Bikenet*, *Cartel*). Esta análise pretende identificar as componentes no domínio, e a possibilidade de representar as suas abstrações através da sintaxe de uma linguagem de programação. É também feita uma breve descrição de infraestruturas de suporte a aplicações *sensoriamento participado* com especial ênfase ao sistema *4Sensing*, que será a plataforma utilizada para testes. Na execução deste trabalho, é necessário perceber a importância e as fases de todo um processo de desenvolvimento de uma DSL, conjugando as vantagens e formas de desenvolvimento de software baseado num paradigma *generative programming*.

2.1 Sensoriamento Participado

A área de *people-centric sensitive* e seus sistemas tem tido grandes avanços nos últimos anos, tornando-se nos dias de hoje uma visão interessante do futuro com o uso de sensores, tanto no meio rural, como urbano, interagindo com humanos e vice-versa. Através da introdução destes sensores em dispositivos móveis (ex. *smartphones*), é possível criar uma extensa rede sem que sejam necessários custos enormes.

Existem dois conceitos para as quais a arquitetura de um sistema *people-centric sensitive* poderá divergir, nomeadamente, *opportunistic sensing* e *sensoriamento participado* [21].

Em *sensoriamento participado* cada utilizador tem consciência da sua presença e colaboração num determinado sistema. O utilizador tem a possibilidade de definir o género de informação a adquirir, analisar e partilhar, estipulando ao mesmo tempo o grau de privacidade que pretende que seja aplicado à sua informação pessoal recolhida [5]. Estas funcionalidades são geridas e suportadas através de aplicações e plataformas móveis e web. A divergir do modelo *sensoriamento participado*, temos uma segunda possível arquitetura chamada *opportunistic sensing* que, pode funcionar sem o auxílio de aplicações, sendo os recursos dos dispositivos móveis utilizados aquando uma intercepção e verificação automática de dados e estados específicos por parte do sistema (ex. localização temporal e espacial do dispositivo).

Retomando o modelo *sensoriamento participado*, contexto desta tese, e suas vantagens, com o aumento de dispositivos móveis haverá, com toda a certeza, um acréscimo enorme na quantidade de informação que nos é disponibilizada, tanto a um nível científico como artístico, urbanístico, político e até mesmo de cidadania. Será possível uma inteligente gestão de informação relativa ao nosso quotidiano, bem estar, saúde, desporto, podendo ser partilhada com outras pessoas e, criando técnicas de desenvolvimento social que irão encorajar a adesão de cada vez mais público [22].

Infelizmente, existem problemas de segurança e privacidade, pois torna-se complicado coordenar qual a informação a recolher e partilhar, e qual a informação a ignorar. Uma outra desvantagem consiste na fidelidade e exactidão da informação recolhida (ex. sensor de acelerómetro mal configurado), pondo em risco a integridade dos resultados calculados pelo sistema. Também a necessidade de registo e configuração prévia, por parte de cada utilizador, pode fazer com que se torne mais difícil a adesão a plataformas *sensoriamento participado* [6].

Os desafios numa arquitetura *sensoriamento participado*, predominam sobretudo em volta da segurança e privacidade da informação pessoal dos utilizadores, existindo uma extrema necessidade de lhes transmitir confiança em relação ao sistema. O processamento e armazenamento de um elevado número e diversidade de dados nestas arquiteturas, é mais um desafio a ser ultrapassado. Até ao momento, maior parte desse trabalho é feito através de servidores presentes no sistema, os quais deverão acartar com o maior número de tarefas de forma a libertar os baixos recursos dos dispositivos móveis. Também existem soluções possíveis apoiadas em redes *Ad-Hoc*, onde se partilham recursos entre dispositivos. Contudo, esta aproximação conduz a mais problemas de privacidade e segurança na informação partilhada pelos utilizadores [5].

Nesta dissertação, pretende-se analisar este domínio de *sensoriamento participado*, assim como algumas aplicações dentro desta área, de forma a estruturar e avançar com parte de uma implementação de uma DSL dentro deste domínio.

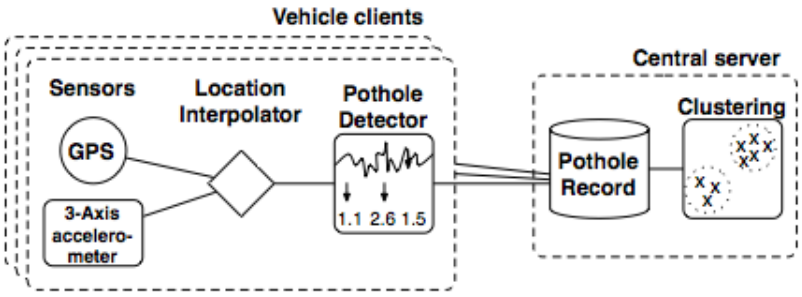


Figura 2.1: Arquitectura da aplicação The Pothole Patrol [14]

2.2 Aplicações em Sensoriamento Participado

Esta secção pretende abordar exemplos de algumas aplicações em *sensoriamento participado*, nomeadamente, *The Pothole Patrol*, *Cartel*, *BikeNet* e *CenceMe*, focando as funcionalidades e vantagens oferecidas por cada uma delas. Pretende-se mostrar exemplos de informação gerada através destas aplicações e, até que ponto ela pode ser relevante para todos. É também descrito o funcionamento, relativo à partilha de informação entre dispositivos móveis, bem como a arquitetura de cada aplicação.

2.2.1 The Pothole Patrol

Aplicação em *sensoriamento participado* que tem como objectivo monitorar o estado das estradas através de sensores instalados em automóveis e com algoritmos especializados, abrangendo de forma fácil e com baixo custo uma área muito extensa. A aplicação ajuda a determinar quais as estradas que necessitam de manutenção, assim como informar os condutores dessas más condições.
O funcionamento do sistema consiste na instalação de sensores num automóvel, registando informação em relação à aceleração nos três eixos xyz e respectiva posição geográfica, através de GPS. Estes sensores interceptam possíveis oscilações referentes a anomalias na estrada, assim como a localização desse evento. Estas detecções são enviadas para um servidor central através de uma ligação sem fios. O servidor irá consumir a informação partilhada por vários carros, filtrando-a e produzindo conclusões acerca das condições de cada estrada [14]. Na figura 2.1 pode ser vista uma representação da arquitetura descrita.

Neste sistema, existe a possibilidade de certos eventos serem mal interpretados, tais como, uma travagem brusca, o bater de portas ou mesmo o atravessamento de obstáculos (ex. caminhos de ferro), sendo combatidos através da recolha e análise de muitas amostras e de algoritmos estatísticos. Outro problema diz respeito às amostras de vários condutores poderem ser diferentes mesmo em locais idênticos, dependendo da forma ou velocidade com que o condutor supera esse local.

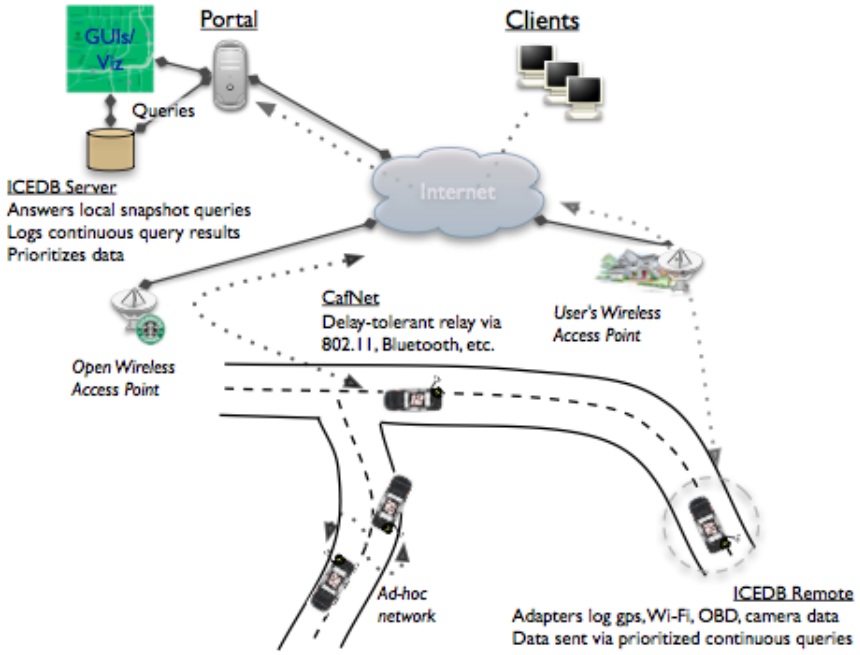


Figura 2.2: Arquitectura da aplicação Cartel [19]

2.2.2 Cartel

Projecto que se destina à recolha, entrega, processamento e visualização de informação extraída a partir de dispositivos móveis colocados em automóveis. Este projecto pretende produzir resultados referentes à monitorização de tráfego nas estradas, através de acelerações e velocidades, assim como a criação de mapas caracterizando zonas da cidade (ex. internet sem fios, zonas poluídas).

A característica principal na definição da sua arquitetura foi a tolerância a problemas de comunicação entre componentes, sendo estes: o *Portal*, *ICEDB* e *CafNet*. O *Portal* é visto como uma zona central do sistema, contendo todas as funcionalidades e aplicações (Web) que disponibilizam ao utilizador informação recolhida pelos sensores. O *ICEDB*, consiste numa base de dados que efectua, de forma sistemática, consultas aos dispositivos móveis, armazenando os resultados devolvidos. Essa comunicação é tratada pela *CafNet*, responsável pela troca de mensagens entre o *Portal*, o *ICEDB* e os dispositivos móveis, podendo ser enviadas "ponto-a-ponto"ou através de intermediários [19]. Para melhor compreensão da arquitectura descrita, ver figura 2.2.

Com o crescimento da utilização deste sistema haverá, igualmente, um enorme crescimento na quantidade de informação ao nosso dispor. Por outro lado, o desempenho do sistema pode tornar-se fraco, caso a média da duração de conexão à internet seja baixa, não existindo ainda perspectivas de evolução neste sentido. Para este sistema funcionar é também necessário que a informação armazenada abranja a maior área possível, sendo

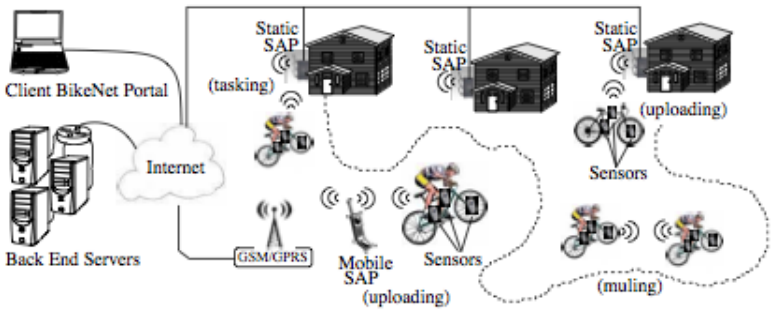


Figura 2.3: Arquitectura da aplicação BikeNet [13]

que se espera um crescimento a nível de dispositivos móveis.

2.2.3 BikeNet

Projeto baseado num paradigma *people-centric*, que consiste na instalação de sensores móveis em bicicletas, recolhendo e armazenando o máximo de informação sobre a utilização da mesma, assim como informação do seu utilizador e seus locais percorridos. Quando analisados os dados recolhidos, é possível retirarmos conclusões referentes ao desempenho do ciclista, aos objectivos que alcançou, ou até sobre a situação ambiental dos locais por onde circulou. O BikeNet oferece ainda funcionalidades de partilha de dados entre utilizadores do sistema, através de uma plataforma Web.

A arquitetura deste sistema, representada na figura 2.3, está organizada em três camadas diferentes, sendo elas: a *camada de sensores móveis*, a *camada SAP* (*sensor access point*) e a *camada Servidor*. A camada de sensores móveis engloba todo um conjunto de sensores, instalados na bicicleta e transportados pelo utilizador, formando uma *BAN* (*bycicle area network*). A SAP está ligada à internet, podendo ser estática (ligação por cabo) ou móvel (GSM e GPRS), funcionando como *gateway* entre a BAN e a camada Servidor. Esta é responsável pela validação, transmissão e segurança da informação que é partilhada entre camadas. A camada Servidor é composta por vários servidores capazes de analisar e armazenar toda a informação recebida, sendo esta informação acedida através de pedidos e consultas feitas por aplicações (Web), que oferecem interfaces para a sua visualização [13].

Neste projeto há que ter em conta o problema do uso de baterias recarregáveis por parte dos dispositivos móveis, pois tendem a possuir uma fraca autonomia. Como solução, existe a possibilidade da bicicleta produzir energia por si mesma, através de movimento (ex. o ato de pedalar). Pode-se também alargar a implementação deste sistema a outros veículos, captando-se informação ambiental de maiores áreas.

2.2.4 CenceMe

A aplicação CenceMe combina um modelo *people-centric* com as novas tendências das redes sociais. Esta é capaz de reconhecer certas ações (ex. andar, sentar, conversar, conduzir, correr), através dos sensores de um telemóvel transportado pelo utilizador, assim como saber a sua localização e fazer a captação de imagens desses locais, partilhando-as automaticamente em redes sociais (ex. *Facebook*, *MySpace*). O utilizador pode também visualizar essa informação, podendo concluir, por exemplo, o local onde despende mais tempo do seu dia-a-dia, ou se a sua atividade física é baixa comparada com a de outros utilizadores.

Uma parte deste sistema é composto por telemóveis e seus sensores, capazes de captar e analisar dados através de classificadores. Os classificadores são utilizados para classificar e interpretar os dados, definindo a ação e movimento em execução. Estes dados são, quando possível, enviados para servidores *Backend*. Esta infra estrutura *Backend* apoia-se sobre servidores que executam algoritmos mais complexos na classificação dos dados, sendo também responsável pelo respectivo armazenamento [25].

Sendo uma aplicação que utiliza constantemente os recursos do telemóvel, será complicado a gestão dos mesmos, como é o caso da bateria. Acresce o facto da aplicação não poder interferir com a execução das funcionalidades normais do telemóvel.

O factor *reutilização* é uma das várias vantagens de uma futura DSL na área *sensoriamento participado*. Por isso é importante estudar o maior número de aplicações possível, de forma a abstrair as diversas funcionalidades de cada uma. Quanto maior o número de abstrações alcançado, maior a taxa de reutilização de código por parte dos programadores.

2.3 Infraestruturas de suporte a Sensoriamento Participado

As aplicações *sensoriamento participado* são suportadas por infraestruturas responsáveis pela troca e gestão da informação dentro do sistema, podendo estas ser centralizadas, em que um servidor é responsável por toda a gestão da informação do sistema; ou descentralizadas, consistindo numa rede distribuída de dispositivos móveis partilhando recursos e informação entre si. Uma arquitetura centralizada leva a uma implementação bastante mais simples, com melhor capacidade de gestão dos dados e segurança do sistema. Uma desvantagem deve-se ao facto da informação de todos os utilizadores ser gerida pelo sistema, originando problemas de privacidade. Esta arquitetura implica também custos bastante mais elevados em comparação a uma arquitetura distribuída, devido à necessidade de implantação de toda a infra-estrutura e respectiva manutenção. Num sistema descentralizado, cada membro do sistema gere a sua própria informação, eliminando a falha na privacidade da informação.

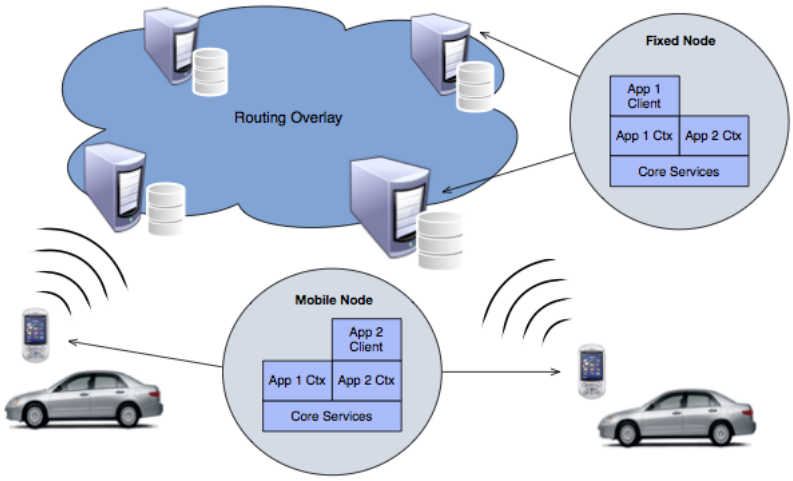


Figura 2.4: Arquitectura 4Sensing [15]

2.3.1 4Sensing

Dado o objectivo desta tese, e o seu tempo limitado, o estudo centrou-se na plataforma 4Sensing que definiu o contexto do trabalho desenvolvido. O 4Sensing descreve uma arquitectura em sistema distribuído capaz de suportar aplicações *sensoriamento participado*. Arquitectura assenta sobre uma infra-estrutura descentralizada, oferecendo uma maior escalabilidade e distribuindo o processamento e o armazenamento, uniformemente, por todos os participantes do sistema [15].

Esta arquitectura é composta por dispositivos móveis e fixos. Estes dispositivos móveis, tendo fracos recursos de computação, encontram-se ligados a uma infra-estrutura de nós fixos, que criam uma rede distribuída e organizada, capazes de operar tarefas mais complexas (ver figura 2.4). Esta rede é também responsável pela gestão do encaminhamento de mensagens entre todos os componentes do sistema. A distribuição das consultas é feita entre nós fixos, utilizando uma de três possíveis estratégias: QTree, RTree e NTree. Cada uma destas estratégias distingue-se pela sua forma de aquisição de dados e disseminação e processamento de pedidos. Informação pormenorizada sobre estas estratégias pode ser vista em [15].

Após testes a esta implementação, concluiu-se que a sua eficiência melhora quando existe um número elevado de nós na rede. Caso contrário, quando a densidade de nós na rede é baixa, este sistema torna-se pouco eficiente. Por outro lado, com um elevado número de nós, o número de mensagens trocadas dentro da rede tem um aumento enorme. No futuro pretende-se explorar formas eficientes na entrega dos resultados das consultas bem como a optimização no processamento de múltiplas consultas [15].

No projeto 4Sensing surgiu a necessidade de focar exemplos de aplicações *sensoriamento participado*, de forma a ilustrar os aspectos propostos por esta arquitetura, tendo

sido criadas algumas abstrações interessantes no âmbito das aplicações escolhidas. Foram três as aplicações tidas em conta neste projeto: *SpeedSense*, *NoiseMap* e *PotholePatrol*. A aplicação *SpeedSense* conclui quais as condições do tráfego em dado momento e localização, assim como a velocidade média dos veículos, planeamento de rotas e estimativas do tempo de deslocação. A aplicação *NoiseMap* determina uma estimativa do ruído em determinadas zonas de uma cidade. *PotholePatrol*, é uma aplicação que permite localizar possíveis anomalias nas estradas.

Nesta plataforma foram também definidos componentes a nível de *middleware*, nomeadamente as tabelas virtuais, responsáveis por especificar e manipular os dados recolhidos pelos sensores, ou partilhados por outra tabela virtual. Cada tabela virtual especifica os dados e as respectivas transformações que estes sofrerão. Cada uma delas requer uma fonte que fornece os dados necessários ao seu processamento. Essa fonte poderá ser *sensores* ou *tabelas virtuais* com informação já calculada anteriormente. Cada tabela virtual deve executar os devidos operadores, também definidos pelo o autor deste projeto, disponibilizando o resultado pretendido. A utilização de um armazenamento e distribuição dos dados de forma descentralizada exige a divisão das tabelas virtuais em dois modelos: *data sourcing* e *global aggregation*. *Data sourcing* refere o processo de produção de dados executado por cada nó, tendo como fonte de dados sensores ou informação armazenada anteriormente. *Global aggregation* constitui uma fase posterior de agregação das várias amostras de dados, produzidos pelos diversos nós do sistema, por parte do nó que efectuou o pedido de informação. Na sequência disto foram pensadas abstrações necessárias às diversas funcionalidades de cada aplicação, sendo cada uma destas abstrações uma tabela virtual.

Em relação à aplicação SpeedSense, as abstrações tidas em conta foram "TrafficSpeed", capaz de obter qual a velocidade média e densidade de veículos em determinada área através de amostras recolhidas por GPS; "TrafficHotspots", para saber quais as zonas de congestionamento de acordo com um determinado nível de "confiança", e "TrafficTrends"que prevê a situação do tráfego de uma determinada zona, segundo informação armazenada anteriormente. As duas últimas necessitam dos dados processados pela "TrafficSpeed". Para PotholePatrol apenas foi definida uma tabela que indica as condições da estrada numa determinada zona, através de informação recolhida e armazenada. Para as funcionalidades da aplicação NoiseMap foram criadas as tabelas "NoiseLevel"e "NoiseTrends"que especifica e prevê o nível de ruído numa determinada zona, respectivamente; ambas precisam da informação processada pela tabela virtual "TrafficSpeed". Estas abstrações relacionam-se entre si através da partilha de informação e reutilização de funcionalidades entre aplicações, simplificando as suas implementações. Pode ser dado o exemplo da necessidade dos resultados da tabela virtual "TrafficSpeed"no processamento de dados da tabela virtual "NoiseLevel". A expressividade dada a estas abstrações, através do seu nome e dos recursos necessários à sua execução, permite aproximar o programador do domínio das aplicações focadas neste projeto, distanciando-o de todo o processo que estas implicam (código necessário à sua execução). Grande parte dos

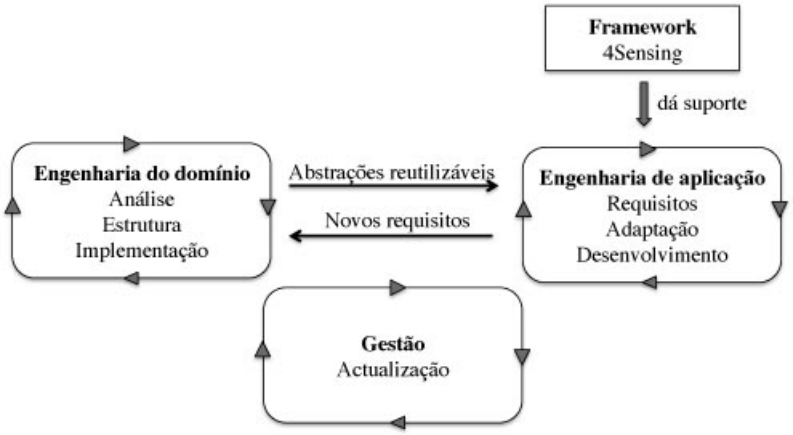


Figura 2.5: Principais processos a ter em conta em Generative Software. Imagem adaptada a partir de [7]

componentes definidos no projeto 4Sensing foram utilizados e adaptados na criação de uma possível DSL na área de *sensoriamento participado*.

2.4 Generative Software Development

Generative software development é um paradigma de computação automática na criação de aplicações através de um conjunto de abstrações. O desenvolvimento de *generative software* pretende focar o vocabulário, conceitos e componentes de determinada área disponibilizando-os de forma clara e automática, textualmente ou graficamente [7]. Nas próximas secções são descritos alguns método/ideias, mais relevantes no âmbito deste trabalho, relativamente a este paradigma *generative software development*. Para mais informação, consultar [7].

2.4.1 Engenharia do Domínio e Engenharia de Aplicação

No desenvolvimento e existência de *generative software* devem ser distinguidos três processos: *engenharia do domínio*, *engenharia de aplicação* e *gestão*. Em *engenharia do domínio* é feita uma análise do domínio tendo em conta as abstrações que podem ser reutilizadas na criação de aplicações nesse mesmo domínio, proporcionando uma estrutura e eficiente implementação do software. *Engenharia de aplicação* consiste no desenvolvimento de aplicações nesse domínio tendo em conta os seus requisitos, através da reutilização das abstrações implementadas anteriormente. Estes dois processos descritos cooperam entre si na evolução e desenvolvimento de novas abstrações importantes no domínio, sendo necessário um constante processo de *gestão* dos seus recursos e componentes. A figura 2.5 representa a execução dos processos referidos, agora adaptada ao trabalho desta dissertação, acrescentando o auxílio da plataforma *4Sensing* no desenvolvimento das aplicações.

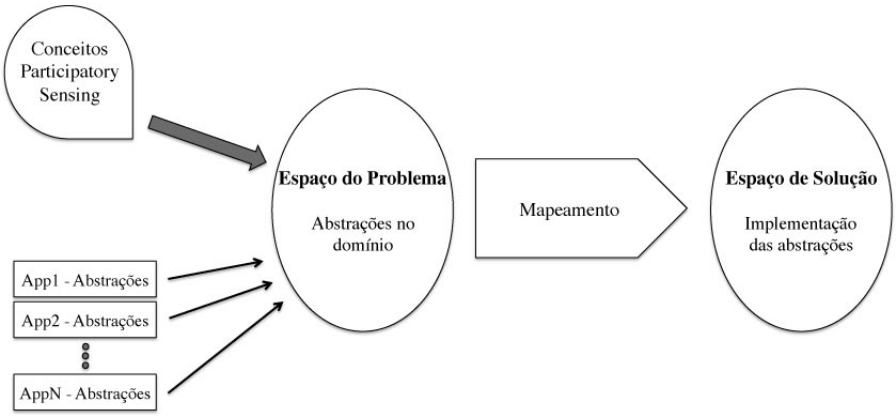


Figura 2.6: Mapeamento entre o espaço do problema e o espaço de solução

2.4.2 Mapeamento entre o Espaço de Problema e o Espaço de Solução

A ideia essencial para a criação deste género de software baseia-se no *mapeamento* das abstrações de um domínio entre o *espaço do problema* e o *espaço de solução*. O *espaço do problema* refere a investigação e recolha de informação de um determinado domínio obtendo-se as suas abstrações principais e mais relevantes. Estas abstrações são mapeadas dando origem ao *espaço de solução*, que define a estrutura e implementação das mesmas. O mapeamento pode ser feito a partir de um ou mais espaços de problema ou originar um ou mais espaços de solução.
Uma primeira estrutura dos vários recursos, dados e processamentos podem ser demonstrados e organizados através de diagramas, sendo um bom ponto de partida para a criação de uma arquitetura de uma DSL.
Pretende-se que utilizadores experientes num determinado domínio consigam exprimir as suas necessidades de forma natural e o mais próximo possível desse domínio aquando o desenvolvimento de uma aplicação. Conclui-se que Generative Programming engloba vários conceitos importantes no desenvolvimento de DSLs.

O trabalho desta dissertação, como ilustrado na figura 2.6, passa por definir um *espaço de problema* referente ao domínio *sensoriamento participado*, através da análise de aplicações nesse domínio, captando as várias abstrações presentes. Posteriormente deve-se estruturar uma potencial arquitetura de forma a ser preenchido o *espaço de solução* com respectivas implementações dos componentes para a criação de uma *framework-specific DSL*.

2.4.3 Diagrama de características

É frequente a utilização de *diagrama de características* ao longo de todo um processo de desenvolvimento de software. Estes diagramas consistem numa notação gráfica introduzida como uma componente da metodologia *Feature Oriented Domain Analysis* [4]. Desde então foram surgindo várias extensões com a intenção de melhorar a sua expressividade

[4]. O seu objectivo é capturar, estruturar e documentar as características necessárias a uma linguagem ou aplicação de um determinado domínio. Esta modelação de características de um domínio e suas respectivas dependências faz com que seja útil a utilização destes diagramas em *generative software development*. Como consequência, este modelo de características criado através de uma análise de domínio pode ser o ponto de partida no desenvolvimento da arquitetura de uma DSL [7].

Diagrama de características baseia-se numa árvore que conduz às possíveis opções que podem ser tomadas determinando as características de um determinado sistema, facilitando a reutilização de componentes que implementarão estas características. Este diagrama é composto por:

- um conceito inicial, ou raiz, que identifica todo o sistema que está a ser modelado;
- por nós que podem ser obrigatórios (sem nenhuma representação ou com um circulo preenchido) ou opcionais (com um círculo não preenchido), assim como atómicos, não podendo ser subdivididos em outros subnós, ou compostos, contendo subnós;
- relações entre eles que podem conter, ou não, restrições de exclusividade ou não exclusividade.

Um exemplo desta notação pode ser visto em anexo, 7.1. Um dos problemas na utilização destes diagramas deve-se ao facto da análise de domínio feita por este poder mostrar-se superficial, sem a devida complexidade [29]. Em Secção 3.1 é mostrado e explicado em detalhe o diagrama respectivo ao domínio de *sensoriamento participado*.

2.5 Domain-Specific Languages - DSLs

Domain-specific languages são linguagens desenvolvidas para um domínio de aplicação específico, elevando o nível de abstração de linguagens mais genéricas. Assim, as DSLs tentam aproximar o programador do domínio e da semântica de uma determinada área, trabalhando diretamente sobre os seus conceitos [27]. Os programadores podem assim usar conceitos e abstrações que lhes são familiares.

2.5.1 Características Distintivas das DSLs

De forma a serem eficazes, as DSLs devem respeitar certas características que fazem destas linguagens um elemento importante na área de engenharia de software [10] (esta referência trata-se apenas de um estudo teórico e exploratório).

Uma DSL deve ser *pequena*, identificando apenas os conceitos e funcionalidades do seu domínio, criando uma notação restrita. Esta notação oferece assim soluções específicas para esse domínio, excluindo uma necessidade de programação extra.

Estas linguagens são geralmente direcionadas, não a utilizadores com algum conhecimento de programação, mas a utilizadores experientes no domínio em questão. Como tal,

devem ser eficientes, proporcionando uma compreensão e aprendizagem a curto prazo. No desenvolvimento de aplicações, o utilizador focar-se-á assim em aspectos do domínio, esquecendo detalhes de implementação.

2.5.2 Vantagens e Desvantagens

Todas as características descritas na secção 2.5.1 implicam várias vantagens e desvantagens em relação a DSLs. Por exemplo, o facto destas linguagens serem direcionadas a utilizadores experientes no domínio faz com que também seja possível a sua participação no desenvolvimento e evolução destas, aproximando de forma rigorosa a semântica do domínio ao resultado final da DSL.

A forte *expressividade* e abstração na notação de uma DSL melhoram a *flexibilidade*, *produtividade* e *usabilidade* no desenvolvimento de aplicações, tudo isto devido à sua eficiência na *compreensão* e *aprendizagem*. Estas características devem-se também à existência de documentação partilhada, normalmente, pelos programadores.

A manutenção de uma DSL, em comparação às GPLs (general-purpose programming languages), é bem mais simplificada dado a fácil compreensão da sua sintaxe. As abstrações de uma DSL estimulam a *reutilização* de soluções, acelerando o processo de desenvolvimento de uma aplicação. Todas estas vantagens fazem minimizar os recursos necessários no desenvolvimento de aplicações reduzindo assim os seus custos. O facto de haver uma DSL num determinado domínio, faz avançar e motivar a criação de novas ideias e ferramentas dentro do mesmo.

Dado as abstrações de alto nível numa DSL, poderão surgir limitações aos programadores em implementações de maior detalhe. O facto de uma DSL estar focada num só domínio, caso a adesão à mesma não se dê por parte de um considerado número de utilizadores, esta irá fracassar rapidamente. O mesmo é mais complicado de acontecer numa GPL, pois podem abranger um elevado número de domínios e por sua vez de utilizadores.

2.5.3 Fases de Desenvolvimento de uma DSL

A escrita que se segue refere várias fases necessárias no desenvolvimento de uma DSL, composto pela fase de *decisão*, *análise*, *desenho* e *implementação* [24]. Contudo, este não é um processo sequencial das fases mencionadas, estando todas envolvidas entre si. O processo de decisão pode implicar uma prévia análise do domínio, que por sua vez, poderá ter de definir critérios no processo de desenho. O desenho da DSL pode ser influenciado por possíveis limites na implementação. Segue-se, portanto, uma breve abordagem a cada uma das fases referidas.

2.5.3.1 Fase de Decisão

A decisão da criação de uma DSL nem sempre é fácil, devendo ser feito um balanceamento entre as suas vantagens e os custos envolvidos no seu desenvolvimento. Este

processo de decisão pode ser definido identificando certos padrões de decisão, referidos nos parágrafos seguintes.
Em relação à sua futura notação, de forma a proporcionar uma fácil compreensão ao utilizador, deve ser possível a transformação visual de elementos desse domínio para uma notação textual (ex. notação textual para representação de um sensor GPS) e vice-versa (útil na criação de ambientes gráficos).
Por outro lado, a *análise*, *verificação*, *optimização* e *paralelização* de aplicações num domínio específico, nem sempre é praticável numa GPL (*general-purpose programming language*) devido à sua complexidade. Quando criada uma DSL, estas 4 características devem ser proporcionadas pela mesma[24].

A programação em GPL requer normalmente a programação repetida de certos algoritmos, consumindo tempo desnecessário aos programadores. Uma DSL deve eliminar estas tarefas repetidas gerando, automaticamente, e reutilizando código. Adicionalmente, a utilização complexa e combinada de várias estruturas de dados deve ser expressa de melhor forma e com maior segurança numa DSL.

Por fim, uma DSL deve facilitar também outros aspectos como a criação de interfaces gráficas capazes de ilustrar o domínio em questão, levando a uma maior compreensão e simplicidade na programação. Deve-se analisar se a DSL a desenvolver, conseguirá obter todos estes requisitos, compensando os custos associados ao seu desenvolvimento.

2.5.3.2 Fase de Análise

A fase de análise, no desenvolvimento de uma DSL, constitui a identificação do domínio do problema e a recolha de informação sobre o mesmo [24]. Pode tratar-se de informação, implícita ou explícita, de documentos técnicos; documentos referentes ao domínio, escritos por profissionais na área; ou mesmo inquéritos feitos a utilizadores deste domínio. O objectivo desta investigação consiste em obter a semântica deste domínio representando-a de uma forma mais abstracta. É estudado o vocabulário, as componentes e tarefas do domínio, estabelecendo-se uma notação capaz de representar todas essas características. Neste trabalho, seguiu-se a abordagem de estudar uma *framework* particular, abstraindo os conceitos mais relevantes. A *Framework DSL* resultante deste trabalho, é o primeiro passo em direção à construção futura de uma DSL em *sensoriamento participado*. Para mais informação sobre requisitos na construção de uma DSL consultar [11].

2.5.3.3 Fase de Desenho

O desenho de uma DSL pode ser caracterizado em dois aspectos diferentes, a forma como esta está, ou não, relacionada com outra linguagem, e o método de especificação no seu design [24].

Uma DSL pode ser desenvolvida através de uma linguagem já existente, sendo esta

mais fácil de implementar e de ser aceite pelos utilizadores, caso estes sejam conhecedores da mesma. Através deste método, podem ser analisadas três perspectivas de implementação da DSL, *piggyback*, onde são usadas partes de uma linguagem existente; *especialização*, restringindo a linguagem existente ao domínio do problema; e *extensão*, que consiste numa extensão da linguagem existente atribuindo-lhe novas características e funcionalidades. Por sua vez, uma DSL também pode ser desenvolvida a partir do zero, sem qualquer semelhança a outras linguagens. No entanto, este processo pode ser bastante complicado e demorado.

Em relação ao método de especificação para o design de uma DSL, este pode ser *informal* ou *formal*. No método informal a especificação da linguagem, da sua sintaxe, é feita de formal natural, sem quaisquer regras ou modelos para tal. O método formal segue um conjunto de regras utilizando modelos de definição de semânticas disponíveis, para o desenho da DSL (ex. Slonneger and Kurtz).

2.5.3.4 Fase de Implementação

A fase de implementação diz respeito à construção da DSL em concreto, utilizando padrões e aproximações definidos na fase de desenho, anteriormente descrita. Na fase de implementação quando criada uma DSL do zero, é necessário implementar todo o processo de tradução da sintaxe da DSL para código máquina. São duas as possíveis abordagens na resolução desta necessidade, a *compilação* e a *interpretação* [10].
Na compilação a linguagem definida na DSL é analisada e resolvida em código máquina ou em código de uma GPL, sendo executado posteriormente pela máquina. A interpretação distingue-se da compilação no sentido que a linguagem é reconhecida, mas não executada na máquina de seguida, em alternativa esta é interpretada. De forma a aplicar estas aproximações, existem ferramentas de auxílio, capazes de compilar qualquer tipo de linguagem, ou mesmo ferramentas especializadas na criação de compiladores e interpretadores para DSLs (ex. *Draco*) [10].
No caso da DSL ser desenvolvida a partir de uma GPL, a tradução para código máquina pode ser feita por *embedding compiler/interpreter* ou *extensible compiler/interpreter* [10]. No primeiro caso, todo o trabalho de compilação é feito pelo compilador da GPL, havendo a vantagem de não ser necessário qualquer conhecimento a nível de compiladores. Em *extensible compiler/interpreter* o compilador da GPL é estendido de forma a englobar os aspectos da DSL.

2.5.4 DSLs em sensoriamento participado

Com o aumento de dispositivos móveis, o conceito de *sensoriamento participado* tem ganho popularidade, estando o desenvolvimento de aplicações nesta área em constante crescimento. Assim, o desenvolvimento de aplicações neste domínio é feito por GPLs (ex. Java), distantes de um vocabulário que a área de *sensoriamento participado* trata, surgindo a necessidade dessa aproximação. É também frequente nestas aplicações a recorrência de

objetos e tarefas comuns a todas elas (ex. reconhecimento de movimentos e ações, desempenhos de um utilizador, características ambientais, etc), consumindo bastante tempo aos seus programadores, quando poderiam simplesmente fazer uma reutilização de soluções já existentes. Através da análise de várias aplicações de *sensoriamento participado*, é possível desenvolver uma DSL, gerando um nível de abstração capaz de resolver estes problemas. Devem ser agrupados os elementos e funcionalidades predominantes neste domínio, englobando e fornecendo todos estes numa só linguagem, através de uma sintaxe apropriada. O objectivo desta dissertação destina-se ao desenvolvimento de uma *framework-specific DSL* específica para uma framework no contexto da plataforma 4Sensing, desenvolvida a partir da linguagem de programação Java, focando o domínio de aplicações *sensoriamento participado*.
Em [30] são descritas metodologias e ferramentas de suporte à implementação, e em [28] é inclusivamente descrito uma abordagem para o desenvolvimento sistemático de *DSLs*.

2.6 Framework-Specific Modeling Languages - FSML

Framework-Specific Language (*FSML*) é uma linguagem de domínio específico desenhada e adaptada a uma framework específica, ou seja, parte dos objectivos desta dissertação. Consiste numa sintaxe abstracta seguido do seu mapeamento com a API da framework. Uma framework apenas disponibiliza uma API para os seus elementos, sem qualquer implementação para os mesmos. A implementação destes elementos deve ser feita na FSML, comunicando, posteriormente, com a API da framework. Esta deverá especificar certas regras de utilização e implementação de cada elemento.

O programador de *FSML* faz uma análise do domínio de forma a identificar os elementos disponibilizados pela framework. De notar que o programador de FSML não tem necessariamente que ser o programador da framework, havendo a necessidade de distinção entre o código relativo à framework e suas funcionalidades, e código que utiliza a framework para a criação de instâncias a partir da FSML.

É cada vez mais comum a criação de aplicações baseadas em frameworks. No seu desenvolvimento é frequente o uso de modelos. A criação de um modelo consiste na criação de instâncias de elementos atribuindo-lhes funcionalidades e atributos. Estes descrevem como as abstrações oferecidas por uma framework podem ser usadas ou implementadas através de código. Uma FSML captura, através do modelo, os elementos e funcionalidades de uma framework adaptando-os a uma linguagem de sintaxe abstracta. Essa sintaxe engloba todo um conjunto de configurações válidas estipuladas e implementadas na framework. Neste processo é necessário um constante mapeamento entre os elementos do modelo e o código da framework. Adicionar ou remover funcionalidades e atributos ao modelo implica alterações ao código da framework, e vice versa [9].

Uma FSML pode permitir *round-trip engineering*. O objectivo é manter, tanto o modelo como o código da framework, consistentes. Em *round-trip engineering*, o modelo e o código atual são comparados de forma a identificar as modificações que ocorreram desde

a última sincronização. Encontradas as modificações, estas são propagadas entre modelo e código de acordo com as decisões do programador. Esta propagação pode ser feito em ambas as direções, permitindo a criação de um novo modelo através do código (*reverse engineering*) ou de um novo código através do modelo (*forward engineering*) [8].

Uma *FSML* pode ser desenvolvida incrementalmente, sendo estendida com novos elementos e opções, dando liberdade e continuidade a qualquer projeto.

2.7 Ferramentas

Existem ferramentas de suporte com ambientes apropriados ao desenvolvimento de software com o objectivo de facilitar a implementação de uma *DSL*. As principais ferramentas consideradas no desenvolvimento deste trabalhado foram: LabVIEW, EMF e GMF.

2.7.1 LabVIEW

O *LabVIEW* consiste numa plataforma de programação gráfica para o desenvolvimento de sistemas de cálculo, teste e controle. Esta programação gráfica aborda várias linguagens de programação, modelos de computação e níveis de abstração para as várias áreas da engenharia. O *LabVIEW* possui um enorme número de ferramentas capazes de adquirir, analisar, exibir e armazenar informação através de mecanismos de filtragem, interpretação e manipulação disponibilizados graficamente.

O ambiente de programação gráfico dispõe de objetos e ligações gráficas construindo-se, através destas, um diagrama que ilustra o funcionamento do sistema desenvolvido. A sua interface divide-se em duas janelas que constituem uma *VI* (*Virtual Instrument*); a *block diagram*, onde temos o código, e suas respectivas funções, representado através de um diagrama de fluxo; e a *front panel*, com controladores (ex. caixas de texto, botões) e indicadores (ex. gráficos e leds) que representam os mecanismos de input e output, respectivamente.

Esta programação gráfica feita através de uma utilização *drag and drop* de elementos, por oposição à implementação de diversas linhas de código num formato de texto, é simples e intuitiva. Podem ser criadas *subVIs*, vistas como funções, métodos ou sub-rotinas, contendo código que define o seu comportamento, capazes de serem reutilizadas dentro de outras *VIs*. Estas características facilitam e aceleram o processo de aprendizagem e desenvolvimento de aplicações. Permite a integração de hardware no funcionamento dos sistemas desenvolvidos, como por exemplo a utilização de um dispositivo capaz de medir a temperatura ambiente, incluindo essa informação no sistema. Existe também a possibilidade de serem adicionados *add-ons* estendendo as capacidades das funções, controladores e indicadores do *LabVIEW* a casos mais específicos, assim como a integração de outras linguagens de programação (ex. ActiveX, C) [31]. A comunidade *LabVIEW* que tem sido criada é outro ponto forte desta ferramenta, proporcionando um melhor

suporte e feedback na resolução de possíveis problemas, assim como no constante desenvolvimento de novas extensões e produtos [20].

Em contrapartida, a programação gráfica pode trazer certas dificuldades. Esta tanto pode simplificar implementações mais complicadas, como fazer com que implementações simples se tornem complicadas, pois podem tornar-se enormes e, desta forma, confusas. No *LabVIEW* uma implementação pode implicar a utilização de diversas *subVIs* e, por sua vez, diversas janelas em simultâneo, tornando difícil a visualização e a compreensão da mesma. Este problema acresce com a impossibilidade de aplicar *zoom in* ou *zoom out* aos diagramas. A constante utilização de *subVIs* faz com que haja um enorme consumo de memória, destruindo a performance de execução de todo o projeto. Muitos utilizadores partilham a ideia que a utilização do *LabVIEW* tem uma aprendizagem um pouco demorada e difícil devido à sua complexidade, necessitando de muito tempo de pesquisa e interação com a comunidade online. Por estas razões a utilização do *LabVIEW* é desaconselhada caso o objectivo do projeto vá para além de uma simples aquisição, análise e visualização de dados [16].

A utilização desta ferramenta na dissertação teria o objectivo de desenvolver uma *framework DSL* através de novas funções, controladores e indicadores, associados ao domínio *sensoriamento participado*. Existe uma aplicação criada através do *LabVIEW*, chamada, *LEGO MINDSTORMS NXT* [1], que desenvolveu os seus próprios elementos para o desenvolvimento de software. Esta aplicação pretende proporcionar a qualquer utilizador a possibilidade de programar robots, vendidos pela empresa LEGO, através de ações *drag and drop* de objetos, de forma simples e divertida. O utilizador é capaz de definir o funcionamento do robot através de blocos relacionados entre si especificando ações que interagem com os seus motores e sensores [26].

No âmbito da dissertação e para o desenvolvimento da *framework DSL* seria necessário integrar código Java nos respectivos elementos criados para a framework, sendo este um ponto fraco do *LabVIEW*. A capacidade de integração de outras linguagens de programação mostra-se limitada nesta ferramenta. A integração de Java no *LabVIEW* necessita da união de várias ferramentas mostrando-se pouco segura e ao mesmo tempo complexa. Esta integração é pouco aconselhada e com pouco suporte por parte da ferramenta e da comunidade *LabVIEW*. Outra possível solução pensada para a utilização desta ferramenta passaria pela criação de componentes necessários à esquematização do funcionamento de aplicações *sensoriamento participado* que, posteriormente, salvado num ficheiro, poderia ser feito o seu *parser*. No entanto, apenas é possível salvar ficheiros em formatos internos à ferramenta sendo extremamente complexo a sua leitura e interpretação.

[1] http://mindstorms.lego.com

2.7.2 EMF - Eclipse Modeling Framework

Em *Model-Driven Software Development* (*MDSD*) existem várias abordagens e tecnologias à nossa disposição. A plataforma Eclipse tem vindo a desenvolver e integrar diversos projetos focados nestas abordagens, sendo um deles a framework EMF. EMF consiste num plugin para a plataforma Eclipse, já de longa existência, com diversas capacidades e potencialidades no desenvolvimento de linguagens de domínio específico (DSLs). Em MDSD é normal, numa fase inicial do desenvolvimento de uma DSL, a utilização de um metamodelo representando as características do domínio que se pretende modelar. No caso do EMF, o respectivo metamodelo é representado pelo Ecore através da simples especificação de classes, respectivos atributos, e relações entre elas.

Para definir e gerir estes modelos esta ferramenta utiliza informação *XMI* (XML Metadata Interchange), formato utilizado para a troca de informação metadata através de XML, partilhado entre as várias ferramentas de MDSD da Eclipse. Existem várias formas de obter o modelo nesta forma, nomeadamente; importando um conjunto de classes Java previamente anotadas; através de um diagrama Ecore (modelado através de UML) disponibilizado pelo projeto EMFT [2]; importando um ficheiro XSD; ou importando um modelo UML2 (Unified Modeling Language - Version2) [17].

Esta ferramenta traz aos utilizadores acostumados a uma modelação orientada por objectos a possibilidade de transformação de um modelo num eficiente, correto e fácil código Java. Esta ferramenta pode gerar um conjunto de classes Java representando todas as entidades e relações presentes no nosso modelo, podendo estas classes ser modificadas pelo utilizador. É também possível a utilização de OCL (Object Constraint Language) no melhoramento da consistência da estrutura e semântica da DSL através da execução de transições, pesquisas e validações ao nosso modelo. Posteriormente e, em conjunto com outras ferramentas, também discutidas neste capítulo, é possível através da modelação e geração de código, a produção de novas ferramentas e aplicações baseadas num modelo [12].

Esta ferramenta foi utilizada no desenvolvimento de um modelo para a framework DSL em *sensoriamento participado*. A utilização de código Java em EMF é também uma mais valia da sua utilização neste projeto, pois toda a dissertação assim como trabalho prévio foi trabalhado em torno da linguagem Java.

2.7.3 GMF - Graphical Modeling Framework

O projecto GMF surge pela necessidade de completar a framework EMF, possibilitando a criação de ambientes gráficos associados aos modelos desenvolvidos, baseados numa plataforma Eclipse. Utilizando esta framework é possível desenvolver uma notação gráfica para uma linguagem de domínio específico (DSL) gerando um editor gráfico capaz de construir diagramas. Permite representar os seus elementos e respectivas características, como por exemplo a atribuição de ícone, nome ou comportamento. O GMF é visto

[2] http://eclipse.org/modeling/emft/

muitas vezes como uma plataforma de mais alto nível que faz a ligação entre EMF e GEF. O projeto GEF [3] oferece uma plataforma para a construção de editores gráficos enquanto que o EMF modela e gere os dados em causa através de um modelo. A função do GMF consiste em ligar estas duas tecnologias de forma a coordenar o modelo através de EMF e produzir uma visualização gráfica através de GEF [12].

O framework GMF é composta por duas componentes base: runtime e tooling framework. A componente runtime é responsável pela ligação entre EMF e GEF disponibilizando uma API para que seja possível uma comunicação e desenvolvimento o mais completo possível. A componente tooling framework permite definir os elementos gráficos e a barra de ferramentas do editor, mapeando os dois posteriormente. Este conjunto de passos é constituído por 4 modelos: *Graphical Definition Model* (.gmfgraph file), usado para definir as figuras, nós, links, compartimentos, etc. possibilitando a sua visualização no canvas do nosso editor; *Tooling Definition Model* (.gmftool file), que permite especificar os nós e links mas desta vez na barra de ferramentas do nosso editor; *Mapping Model* (.gmfmap file), sendo este o mais importante de todos os modelos pois é ele quem faz o mapeamento dos elementos definidos nos dois modelos antes descritos em conjunto com o modelo Ecore criado no EMF, gerando o *Generator Model* (.gmfgen file); por fim este *Generator Model* tem apenas a função de adicionar informação que será usada na geração de código necessário à execução do editor [17].

GMF é uma ferramenta bastante completa, sendo possível a construção de editores com uma visualização rica e de grande funcionalidade. No entanto, e por isso, torna-se uma ferramenta bastante complexa e de difícil aprendizagem, podendo levar algum tempo até ser compreendida.

Esta ferramenta, em conjunto com EMF, mostra ser estável, coerente e completa. Foi portanto, a escolhida para o desenvolvimento do editor gráfico feito nesta dissertação. Mais detalhes da sua implementação serão descritos na secção 4.3.

2.7.3.1 EuGENia

EuGENia faz parte do projeto Epsilon [4] composto por ferramentas e linguagens de programação com tarefas específicas orientadas a uma engenharia dirigida por modelos (MDE - Model Driven Engineering) e que interagem e completam ferramentas, como EMF e GMF. EuGENia consiste num plugin de auxílio a utilizadores de GMF para gerar um editor completamente funcional através de anotações, escritas em EOL [5], no modelo Ecore, gerando automaticamente os modelos *.gmfgraph*, *.gmftool* e *.gmfmap*.

O principal objectivo deste plugin é proporcionar a entrada de novos utilizadores GMF, possibilitando uma forma rápida e fácil de desenvolver uma primeira versão de um editor. No entanto, depois da primeira experiência e da criação do primeiro editor,

[3] http://www.eclipse.org/gef/
[4] http://www.eclipse.org/gmt/epsilon/
[5] http://www.eclipse.org/gmt/epsilon/doc/eol/

esta aplicação torna-se limitada quando existe necessidade de explorar outras funcionalidades, pois torna-se uma tarefa impossível a edição dos ficheiros modelo (*.gmfgraph*, *.gmftool*, *.gmfmap* e *.gmfgen*). Apesar disto, o plugin nunca poderia dar suporte a muitas funcionalidades pois tornar-se-ia igualmente complexa em comparação ao GMF.

Este plugin foi utilizado numa primeira interação entre um editor, criado através das ferramentas acima mencionadas, e um simulador de redes. No início do capítulo 4 será novamente mencionado esta implementação.

3

Abstrações do domínio para uma DSL

Quando se pretende criar um novo modelo para determinado produto é por vezes vantajoso fazê-lo através de algo já desenhado, com sucesso, anteriormente. O risco de começar do zero é muitas vezes elevado, sendo seguro desenvolver e aprofundar conhecimento alcançado no passado.

Uma DSL pode derivar de duas formas, *bottom up* e *top down*. Numa perspectiva *bottom up* o desenvolvimento é feito com a recolha de abstrações através de algo já implementado ou pensado anteriormente. Existe o risco de desenvolvimento de uma linguagem direccionada apenas ao que já foi feito, não havendo oportunidade no surgimento de novas ideias. Em *top down* o desenvolvimento da linguagem é feito somente a partir dos conceitos e regras no domínio, podendo haver problemas na complexidade de implementação das abstrações. O ideal passa por desenvolver uma DSL equilibrando estas duas prespectivas.

Este capítulo pretende identificar e estruturar abstrações no domínio de *sensoriamento participado* para uma futura *DSL*. Pretende-se identificar todo um conjunto de características associadas a este domínio.

Por outro lado, ao longo do projeto *4Sensing* houve necessidade de produzir a manipulação de dados num sistema de *sensoriamento participado*, tendo sido criados componentes para tal, cada um deles com a sua respetiva função. Através da aplicação destes componentes foi possível refletir processamentos de dados feitos por aplicações nesta área. Foi então que surgiu a ideia/intenção de extrair e modelar estes componentes, integrando-os com as restantes características do domínio de *sensoriamento participado*,

estruturando uma futura *DSL*. Todo este processo será feito no decorrer deste capítulo.

3.1 *Diagrama de características*

Como ponto de partida no desenvolvimento de uma arquitetura para a *DSL* em *sensoriamento participado* estruturou-se um *diagrama de características* de forma a identificar, clarificar e ilustrar características comuns entre as várias aplicações neste domínio. São 5 as características principais e, obrigatórias, nesta área de aplicações: *dados* em si, *aquisição* de dados, *processamento* de dados, estratégia de *distribuição* dos dados, e a forma como a aplicação *interage* com os dados presentes no sistema (ver figura 3.1).

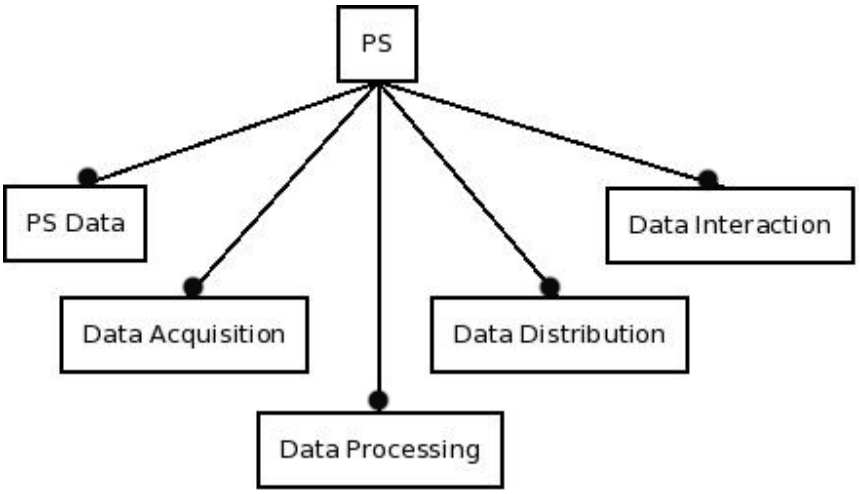


Figura 3.1: Diagrama de características - Visão geral das 5 características do domínio de *sensoriamento participado*.

Assim como em todos os domínios de aplicação, um sistema de *sensoriamento participado* necessita obrigatoriamente de dados (ver figura 3.2). Estes podem conter determinados tipos relacionados com os diversos sensores disponíveis. Para que estes dados tenham relevância no sistema é sempre necessário possuírem características relativamente ao espaço e, por vezes, tempo.

O processo de monitorização e aquisição de dados é feito através de diferentes tipos de sensores, que monitorizam e produzem diferentes tipos de dados, podendo estes serem móveis ou estáticos (ver figura 3.3). Como exemplo, podemos referir a aplicação The Pothole Patrol que faz a sua aquisição de dados através de sensores GPS e acelerómetro embutidos em carros que produzem dados sobre a localização e oscilações sofridas devido a anomalias na estrada.

O processamento de dados, também indispensável a todas as aplicações, consiste no tratamento dos mesmos de forma a poderem ser interpretados e, assim, extraídas conclusões. Este processamento inclui uma sequência de operações de *classificação*, *transformação*, *junção* e *acumulação* nos dados capturados (ver figura 3.4). No caso da aplicação Cartel, e como exemplo deste processamento, é necessário efetuar uma junção dos dados

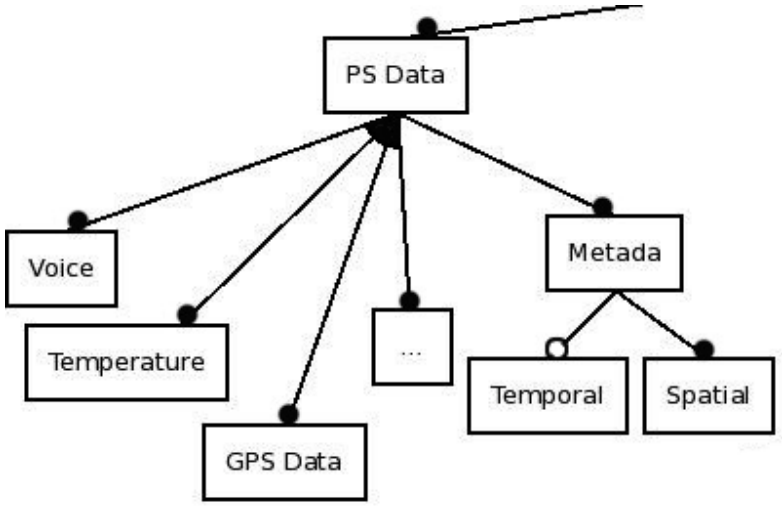


Figura 3.2: Diagrama de características - Elemento dos dados.

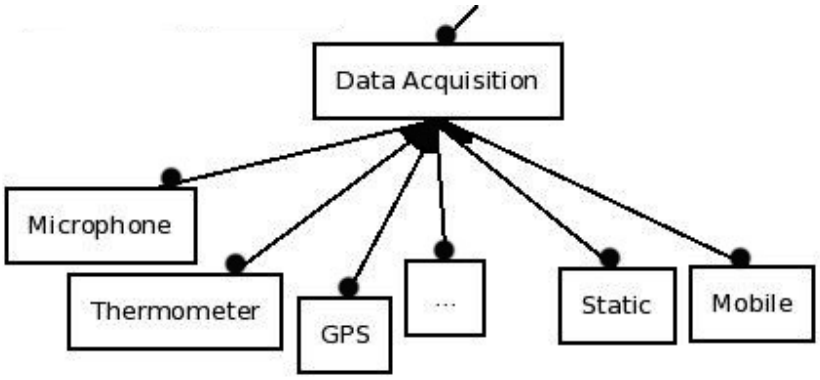


Figura 3.3: Diagrama de características - Elemento da aquisição de dados.

capturados numa determinada área geográfica, classificando-os de seguida como possíveis áreas de congestionamento, ou não.

Em qualquer aplicação é necessário definir se a distribuição dos dados será feita de forma *centralizada*, por servidores do sistema destinados apenas ao armazenamento e distribuição dos dados, ou *descentralizada*, por parte de todos os dispositivos intervenientes no sistema, já tendo sido descrito em detalhe cada uma delas na secção 2.3, assim como as suas respetivas vantagens e desvantagens (ver figura 3.5).

A interação dos dados entre os componentes de um sistema de *sensoriamento participado* pode ser feita de forma *contínua*, sendo a informação entregue à aplicação no momento que é captada pelos sensores, ou *descontínua*, onde os dados são armazenados e utilizados posteriormente (ver figura 3.6).

Estas são as características base que a instância de uma aplicação tem de gerir para que a mesma funcione. De relembrar que nesta primeira abordagem ao domínio, através de um *diagrama de características*, não se pretende que haja uma complexidade excessiva, mas sim manter uma visão superficial e clara do sistema. O diagrama de características

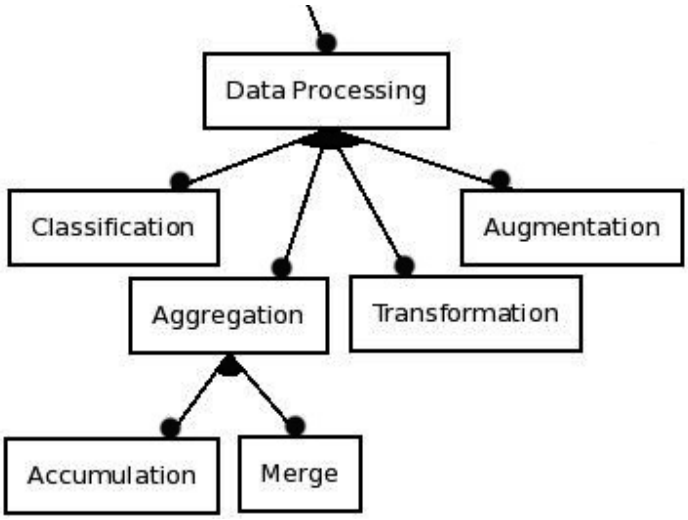


Figura 3.4: Diagrama de características - Elemento do processamento dos dados.

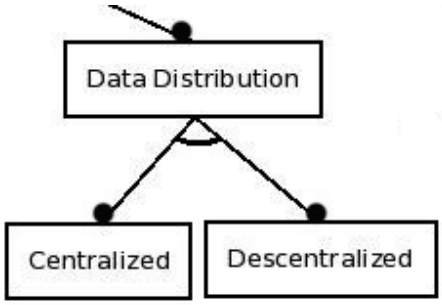


Figura 3.5: Diagrama de características - Elemento da distribuição dos dados.

completo pode ser visto em anexo, figura 7.1.

3.2 Arquitetura

A arquitetura da DSL consiste numa estruturação de todo o conjunto de características do domínio de *sensoriamento participado* relacionando e reutilizando ao mesmo tempo os componentes definidos no sistema *4Sensing*, através de um diagrama UML. Desta arquitetura resultaram 3 grupos: *fontes de dados*, *dados* e *manipulação de dados*. A figura 3.7 mostra uma visão geral dos 3 grupos.

3.2.1 Fontes de dados

Qualquer aplicação tem a necessidade de especificar a forma como é feita a aquisição de dados necessária ao funcionamento da mesma. Este grupo representa todos os elementos que possibilitam essa aquisição.

A monitorização de acontecimentos relevantes à aplicação é feita através de um, ou mais, *sensores* de diferentes tipos, nomeadamente, GPS, Termómetro, entre muitos outros, cada um deles captando diferentes tipos de dados. Um sensor pode ter uma vertente

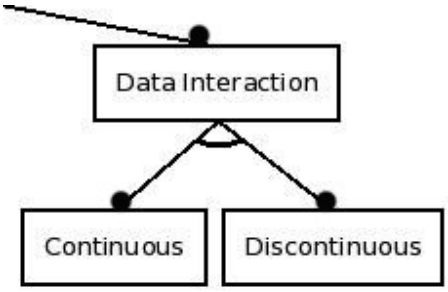


Figura 3.6: Diagrama de características - Elemento da interação dos dados.

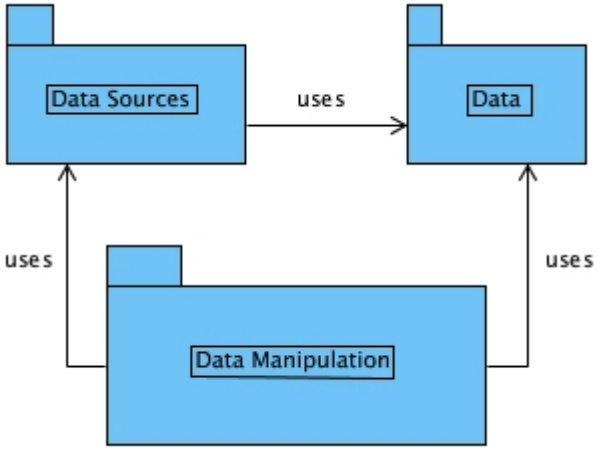


Figura 3.7: Visão geral dos 3 grupos da arquitectura da DSL

móvel, sendo implícito a sua localização através de GPS. Uma pessoa pode também ser vista como um sensor no sentido que é capaz de captar e partilhar informação que lhe é acessível e importante à aplicação. Um sensor possui ainda atributos referentes à forma como irá captar e transmitir os dados ao sistema:

- *Qos*, refere a latência na aquisição dos dados por parte dos sensores. Esta pode optar por dois modos: *online*, latência baixa, os dados estão aptos a ser utilizados pouco tempo depois de terem sido capturados; ou *deferred*, associado a modelos de sistemas com uma conectividade inconstante ou um funcionamento offline;

- *sampling*, representa a forma como são recolhidas as amostragens por parte do sensor. Pode ser caracterizado como: *periodic*, definindo uma captura e partilha de dados de forma periódica; ou *event driven*, em que os dados monitorizados entram no sistema apenas se forem identificados como sendo relevantes;

- *tasking*, caracteriza o processo sobre o qual as aplicações irão, ou não, controlar a actividade dos sensores. Existem 2 formas de controlo: *query-driven*, onde faz sentido capturar informação apenas mediante a existência de consultas feitas pelos nós num dado momento; ou *application-driven*, sendo os dados considerados relevantes

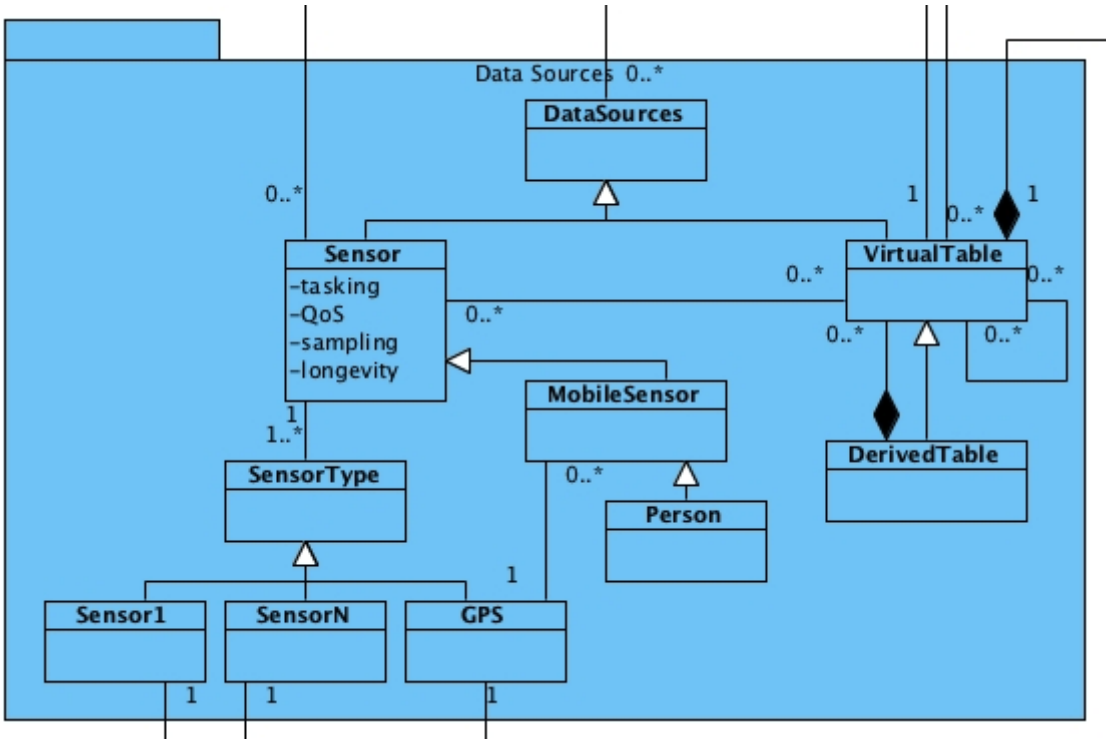


Figura 3.8: UML das fontes de dados.

para o sistema em geral, uma posterior utilização destes, e não necessariamente e somente para consultas que se estão a fazer de momento (utilizado normalmente em aplicações que operam em modo deferred);

- *longevity*, definindo a informação capturada como sendo, *persistent*, em termos de permanência no sistema, ou *volatile*, sendo a informação apenas consumida imediatamente depois de ter sido capturada.

Para uma análise mais pormenorizada de cada atributo ver [15].

As aplicações modelam os dados de acordo com *tabelas virtuais*. Estas recebem um *stream* de dados através de um ou mais sensores, ou mesmo de outras tabelas virtuais, especificando uma transformação desse stream através de uma sequência de componentes (definidos num *pipeline*) criados para esse efeito - componentes de *manipulação de dados*. Uma tabela virtual pode ainda ser construída a partir de outra tabela virtual definida anteriormente, ou seja, ser uma redefinição de uma tabela já existente.

3.2.2 Dados

Numa DSL existe a necessidade de especificar os tipos de dados que a mesma manipula e quais as características de cada um deles.

Uma aplicação coleciona informação capturada pelos sensores. No início esta é vista como informação *crua*, sem, ainda, qualquer tratamento. O formato desses dados é vasto dependendo do sensor pelo o qual são capturados. Cada um destes dados é modelado

pelo sistema através de um *tuplo* com um determinado tipo e composto por um conjunto de atributos. Os tuplos são posteriormente agrupados num *stream* produzindo um fluxo de dados constante a ser consumido pelos diversos componentes da DSL. Um *stream* pode ser *livestream* ou *storedstream*. *Livestream* consiste numa sequência de tuplos sem limite que é criada à medida que os dados são capturados pelos sensores e entregue de imediato ao sistema. Num *storedstream* a sequência de tuplos é finita e representa o passado de um *livestream* utilizado mais tarde pelo sistema.

Quaisquer que sejam os tipos de dados manipulados nesta DSL têm uma característica em comum, nomeadamente a sua *metadata*. Todas as leituras a partir de sensores deverão refletir um momento no tempo e uma localização no espaço, esta característica contextual dos dados é abrangida pela *metadata*. Todos os tipos de dados presentes na DSL deverão também possuir determinados atributos de qualidade. Na área de aplicações *sensoriamento participado* os atributos tidos em conta para um funcionamento consistente do sistema foram:

- *modificabilidade*, possibilidade de modificação de um tipo de dado do sistema;
- *prioridade*, por vezes existe a necessidade de definir prioridades nos dados que entram no sistema. Como exemplo, numa aplicação como Cartel, aquando uma análise do tráfego numa determinada área geográfica, pode ser dado prioridade à entrada de dados referentes a segmentos no centro dessa mesma área;
- *durabilidade*, relativamente ao destino que os dados terão e qual a sua durabilidade no sistema, ou seja, se apenas existe relevância nos dados no momento da captura ou também numa posterior utilização;
- *continuidade*, se os dados são transmitidos de forma continua , ou não, tendo sido este referido na secção anterior (secção 3.1 como uma característica da interação dos dados;
- *confiança*, a existência de erros na captura de fenómenos por parte dos sensores leva a deduções erradas por parte da aplicação, sendo por isso importante garantir um certo grau de confiança nos dados recolhidos. Na aplicação The Pothole Patrol é descrito a possibilidade de erros na captura de dados por parte dos sensores relativamente a fenómenos mal interpretados, como travagens bruscas ou o bater de portas do veículo, havendo necessidade de criar métodos de deteção e correção destes erros.

3.2.3 Manipulação de dados

Como o nome indica este último grupo diz respeito a todo um conjunto de manipulação de dados, e suas fontes. Esta manipulação engloba a *aquisição* de dados, feita através das fontes de dados, referidas anteriormente; o *processamento* de dados; as *estratégias de distribuição* dos dados; e por último a definição de um *modelo de interação*.

Como dito, um *stream* alimenta um *pipeline* composto por uma sequência de operadores. A transformação de um stream representa a aplicação dessa sequência de operadores, cada um com a sua respectiva função. Cada pipeline deve ser descrito como *data sourcing*, contendo uma sequência de operadores a ser executada na recolha e produção de informação, feita por cada nó do sistema; ou *global aggregation*, tendo este último o objectivo de agregar toda a informação disponibilizada por cada nó do sistema. Estas duas componentes foram já referidas na secção 2.3.1.

Em relação a *estratégias de distribuição*, estas podem consistir num sistema *centralizado* ou *descentralizado*, já discutidos anteriormente. Aquando uma estratégia descentralizada devem ser tidos em conta ambos os modelos de um pipeline, *data-sourcing* e *global aggregation*. *Data sourcing*, de forma a que cada nó possa produzir os seus dados quando lhe é feito um pedido; *global aggregation*, para que o nó que efectuou a consulta execute um processo de agregação de toda a informação que lhe foi entregue. Em caso de estratégia centralizada o processo de agregação global é dispensável.

A *interação de dados* na aplicação entre os vários intervenientes é feita a partir de *queries*. Estas acedem aos dados a partir de tabelas virtuais definindo transformações e restrições nos dados que pretendem receber. Uma *query* poderá executar de forma *contínua*, normalmente especificando uma restrição no espaço, ou seja, uma área de interesse, sendo os dados utilizados mal entrem no sistema através de um *livestream*; ou *histórica*, através de um *storedstream*, acrescendo uma restrição temporal a este último.

3.3 Operadores

Um *operador* é utilizado para articular os dados monitorizados de forma a conseguirmos extrair informação relevante dos mesmos. Cada *operador* recebe um stream (sequência de tuplos) aplicando a sua função a cada tuplo do stream, produzindo assim um novo stream que será o input do próximo operador. Os operadores dividem-se em 4 grupos distintos: *classificação*, classificam os dados mediante determinada característica; *transformação*, para processamentos e transformações específicas nos dados; aumentação, realçando a relevância dos dados; e *agregação*, estando este último dividido em *acumulação* e *junção*.

3.3.1 GroupBy

Operador de classificação que particiona um stream, em sub-streams, tendo em conta uma condição. Cada sub-stream pode ser processado de forma independente, sem influenciar todos os outros, suportando um processamento de dados em paralelo. Este operador define uma nova sequência de operadores, através de um sub-pipeline, que será aplicada a cada sub-stream. De forma a simplificar, e adoptando a decisão tomada no sistema *4Sensing*, um sub-pipeline não poderá conter outro operador do tipo *groupBy*.

3.3.2 Classifier

Operador de classificação usado de forma a gerar uma conclusão em relação a uma sequência de dados, na qual é aplicado. Este operador aplica uma determinada condição a cada tuplo de um stream, reencaminhando, caso a condição se verifique, o respetivo tuplo mas agora classificado. Este operador é normalmente aplicado após uma agregação de valores de forma a gerar uma conclusão sobre esse resultado.

3.3.3 Set

Este, também de classificação, cria uma sequência dos elementos mais recentes, recebidos, de acordo com um critério. Como exemplo, uma sequência dos tuplos mais recentes de cada nó do sistema, sendo o atributo neste caso o *Id* dos nós. Por sua vez este operador poderá encaminhar esta sequência para o próximo operador de dois modos. Em modo *change*, a sequência é encaminhada sempre que houver alterações, caso contrário, se definido o modo *eos*, a sequência será encaminhada apenas quando for interceptado um sinal de "fim de stream"relativamente ao seu input.

3.3.4 Processor

O operador de transformação, *processor*, permite desenvolver implementações para o processamento de dados. Este tem um carácter mais livre em relação aos outros operadores, oferecendo ao utilizador a possibilidade de aplicar funções mais concretas e complexas sobre os dados do sistema.

3.3.5 Filter

O operador *filter*, de aumentação, reencaminha os tuplos recebidos apenas se um determinado atributo desse tuplo não variar significamente relativamente aos tuplos que o precederam. Pode ser usado para restringir um stream a certos resultados, ou filtrando transições irrevelantes ou erradas num stream contínuo.

3.3.6 TimeWindow

Na existência de uma query contínua existe frequentemente a necessidade de ir recebendo actualizações com determinada dimensão e frequência monitorizando o estado corrente do fenómeno que se pretende inferir resultados. Este operador de agregação e, por sua vez, acumulação, é utilizado para este efeito definindo-se, em segundos, a frequência com que se pretende receber dados e quanto tempo se pretende estar a receber dados, delimitando assim a dimensão do conjunto recebido.

3.3.7 Aggregator

O operador *aggregator*, de junção, executa operações de agregação - máximo, ,mínimo, contagem, somatório e média - a cada elemento da sequência devolvendo o respectivo

resultado. Caso haja necessidade de executar operações mais complexas que as mencionadas, é possível desenvolver implementações no interior deste operador. A sequência de tuplos que este operador recebe tem de ser finita.

3.4 Metadata: DSL vs 4Sensing

Nesta secção é feita uma análise relativamente aos atributos dos dados tida em conta no diagrama UML, percebendo quais as características que se verificam no sistema 4Sensing, relativamente aos dados que este manipula, e como se exprimem. Esta análise é descrita pela tabela 3.1 acompanhada de uma legenda explicando de que forma é exprimida cada característica no respectivo tipo de dado.

	RawData	Stream	Tuplo
Modificabilidade	-	X a)	-
Continuidade	-	X b)	-
Durabilidade	-	X c)	X d)
Prioridade	-	-	-
Confiança	-	X e)	X f)

Tabela 3.1: Comparação entre características 4Sensing e DSL em *sensoriamento participado*

a) Um *stream* pode ser modificado através da aplicação de uma sequência de operadores originando um *derived stream*.

b) Existência de *live stream*, sequência de tuplos sem limites e contínuo.

c) Existência de *stored stream*, sequência finita de tuplos terminada através de um sinal "*fim de stream*". Trata-se de informação persistente.

d) Um *tuplo*, quando associado a um *stored stream*, é armazenado sendo conservada a sua informação de forma persistente.

e) Aplicando operadores de classificação aos respectivos tuplos de um stream seguido de uma filtragem, podemos obter, por consequência, um *stream* com valores de confiança mais ou menos altos.

f) Como dito, é possível aplicar operadores de classificação a cada tuplo, revelando determinados valores de confiança em cada um deles (caso da tabela *TrafficHotspots*).

3.5 Sumário

Este capítulo pretende contribuir com a modelação dos vários conceitos no domínio de *sensoriamento participado*, e respetiva extração e integração dos componentes do projeto *4Sensing*. Foi feita uma primeira análise ao domínio, identificando-se as várias características presentes neste, e de que forma se relacionam entre si. Para isso, utilizou-se

um *diagrama de características* (figura 7.1) estruturando-se as abstrações partilhadas pelas diversas aplicações. A partir das abstrações capturadas e uma extração dos componentes do projecto *4Sensing*, modelou-se uma possível arquitectura para uma *DSL* através de um *diagrama UML*, ilustrado pela figura 7.2. Aos dados desta arquitetura estão associados certos atributos de qualidade, sendo que alguns já preenchidos pela plataforma *4Sensing*. A tabela 3.1 mostra quais são esses atributos.

4

Framework-specific DSL

O 4º Capítulo retrata todo um processo de desenvolvimento da DSL gráfica e suas características, de forma a cooperar com a plataforma *4Sensing* e assim, proporcionar um desenvolvimento simples e correto de aplicações em *sensoriamento participado* no contexto desta plataforma. Todo este processo de desenvolvimento do editor passa pela criação do modelo *ecore* - em parte equivalente à arquitetura descrita na secção 3.2 - geração das suas componentes gráficas e respetiva barra de ferramentas, e, finalmente, a implementação do código que fará a ponte entre este e o simulador 4Sensing.

Como forma de experimentação e garantia que as ferramentas estudadas (secções 2.7.2 e 2.7.3) seriam as mais indicadas para o desenvolvimento de todo o sistema, foi desenvolvido uma primeira amostra de um editor que permite a simulação de uma rede de nós a partir dum sistema distribuído. Foi feita uma primeira ligação, menos complexa, entre editor e simulador, servindo de primeira abordagem nesta matéria. Este primeiro desenvolvimento serviu também como início de uma aprendizagem nas ferramentas EMF e GMF, tendo sido neste caso utilizado o plugin EuGENia descrito na secção 2.7.3.1, produzindo um editor básico de forma simples e rápida.

4.1 Visão geral da solução proposta

Antes de prosseguir com o desenvolvimento de cada uma das componentes de toda a framework que será criada, é importante explicar como será feito o processo de desenvolvimento de uma aplicação a partir da mesma. A solução proposta, de forma a atingir os objectivos do trabalho, gira em torno de três componentes principais: desenvolvimento da aplicação através de um *editor gráfico*, *compilação*, e *simulação* da aplicação desenvolvida.

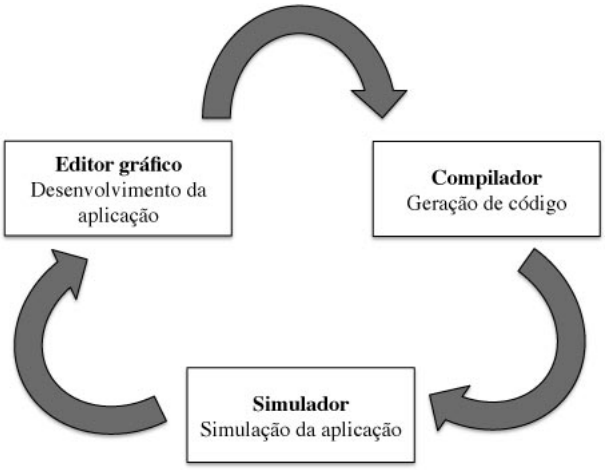


Figura 4.1: Representação da solução proposta

No desenvolvimento da aplicação, a interface do editor deverá disponibilizar graficamente objetos que representem as várias componentes definidas na plataforma *4Sensing*, manipulados através de ações *drag and drop*. Estes devem ser selecionados e relacionados entre si, ilustrando o funcionamento da aplicação. Uma vez definida a estrutura de funcionamento da aplicação, esta deverá ser compilada. O processo de compilação interpreta a estrutura dada à aplicação e gera código de integração ao simulador *4Sensing*. Por sua vez, este simulador é responsável pela execução da aplicação criada, num ambiente simulado através de uma rede de sensores. Quando termina a sua execução, são analisados os seus resultados levando, ou não, a uma alteração na estrutura da aplicação, novamente através do editor gráfico. Gera-se portanto, um ciclo em volta destes três processos (ver figura 4.1). Cada exemplo de aplicação desenvolvido para testes e validações exigirá a formação deste ciclo.

4.2 Modelo Ecore

O modelo *ecore*, formato partilhado por diversas ferramentas da plataforma Eclipse no desenvolvimento de software, consiste num meta modelo composto pelas várias características e componentes do domínio que se pretende modelar. Foram estruturados os elementos necessários ao funcionamento de uma framework no domínio *sensoriamento participado*, a partir das características e componentes anteriormente captados neste domínio de aplicações. Este modelo está diretamente adaptado às necessidades e requisitos do editor e plataforma 4Sensing, não sendo por isso necessariamente idêntico à arquitetura proposta na secção 3.2.

Este modelo é definido através de classes e respetivas relações, atributos, operações, tipos, etc. A classe *principal*, que irá conter todos os elementos presentes no editor - desde objetos, respetivas propriedades, transições, etc - irá corresponder mais tarde à tela do editor, onde será desenhada a arquitetura da nossa aplicação. Neste caso esta classe

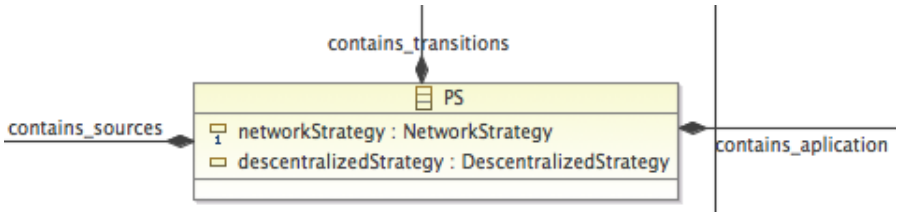


Figura 4.2: Diagrama do modelo *ecore* - classe principal

representa o domínio *sensoriamento participado* em geral, contendo diretamente *fontes* e *manipulação* de dados, *aplicação* e *transições* (figura 4.2). Nela existem também todo um conjunto de atributos que permitirá parametrizar a execução do simulador (tema discutido em detalhe na secção 4.7). Para uma visão mais clara de todo o diagrama deste modelo ver 7.3.

4.2.1 Fontes e Manipulação de dados

Assim como na arquitetura da DSL, podem ser adicionadas várias fontes de dados à nossa aplicação, variando entre *sensor*, tendo como sua herança os diversos tipos de sensores disponíveis pela aplicação, e *tabela virtual*. Cada sensor tem associado os atributos: *qos*, *sampling*, *tasking*, *longevity* (secção 3.2.1) e *sensorType* definindo-o como sensor *móvel* ou *estático*. As tabelas virtuais possuem um *nome* e um ou mais *tipos de tuplo* associados ao seu input e output (ver figura 4.3). Estas têm de conter, pelo menos, um *pipeline* especificando uma sequência de operadores e, podem ou não conter outras tabelas virtuais dentro delas. Em relação aos pipelines, estes contém elementos do tipo *operador* (no mínimo um), podendo ser, como descrito na arquitetura da DSL: *processor*, *classifier*, *set*, *groupby*, *filter*, *aggregator* e *timewindow* (ver figura 4.4).

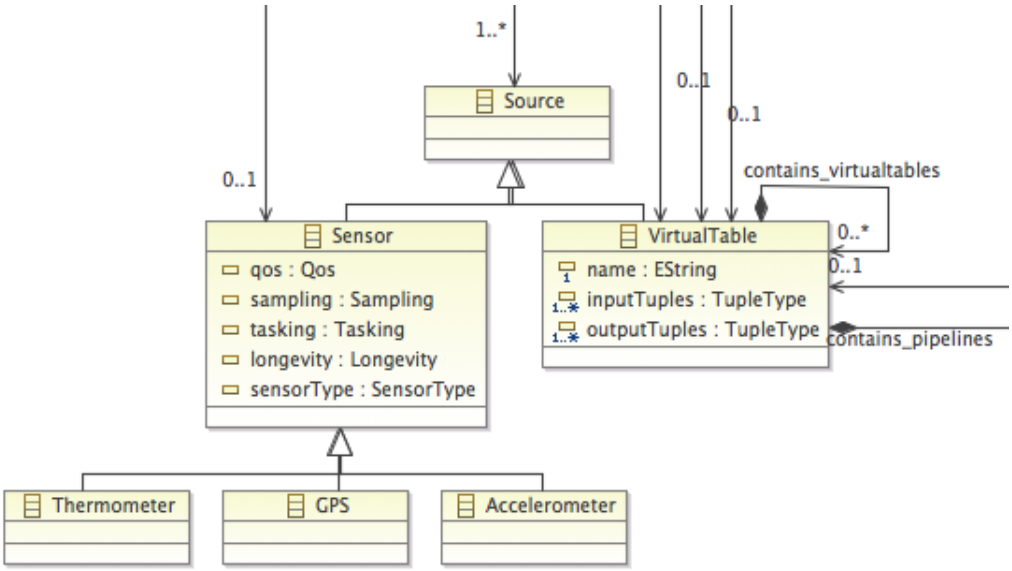


Figura 4.3: Diagrama do modelo *ecore* - sensor e tabela virtual

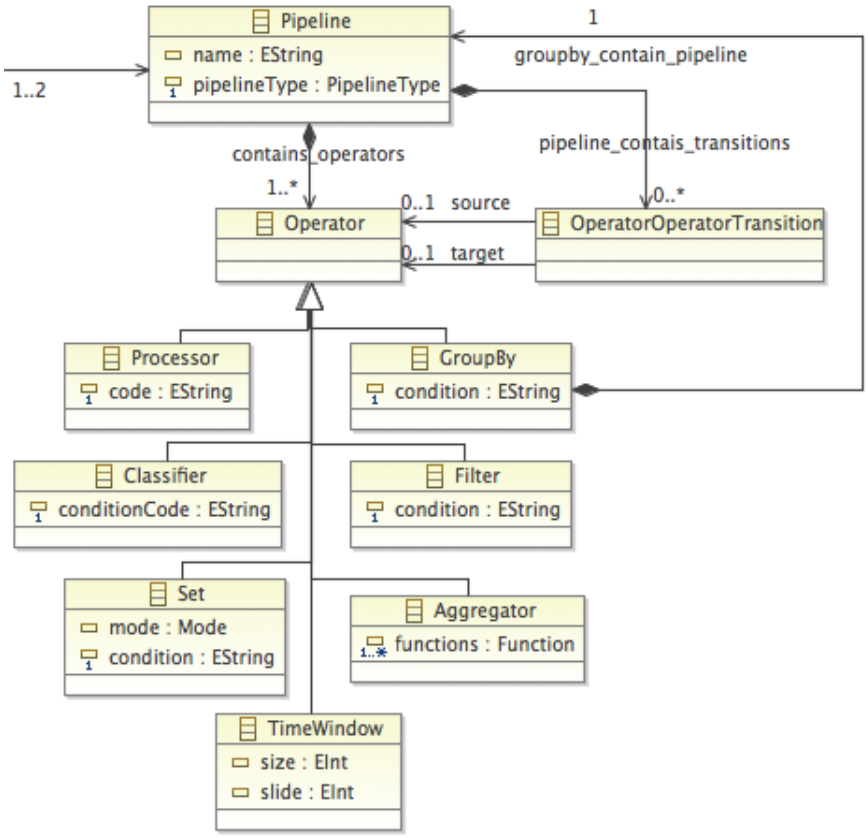


Figura 4.4: Diagrama do modelo *ecore* - pipeline e operador

4.2.2 Aplicação

O elemento *aplicação* (figura 4.5) representa o componente do sistema que efetuará partilhas e consultas à rede através das tabelas virtuais. Qualquer instância desenhada no editor terá de possuir este elemento ligado a uma tabela virtual. De momento poderá ser um pouco irrelevante a sua presença no sistema, mas no futuro poderão haver possíveis parametrizações no mesmo, definindo, por exemplo, formas de *output* para a aplicação.

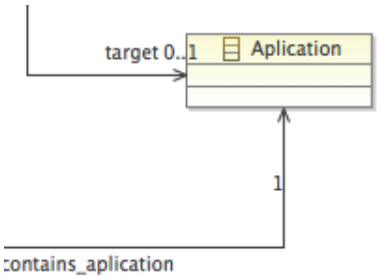


Figura 4.5: Diagrama do modelo *ecore* - aplicação

4.2.3 Transições

A interação de todos os elementos mencionados é feita através de transições representando um fluxo de dados numa determinada direção. Para tal este elemento é parametrizado com uma *origem*, de onde provêm os dados, e um *destino*, para onde são encaminhados os dados. Para além dos vários tipos de transições contidas na classe "principal", existe também um tipo de transição apenas associado aos pipelines, com o objetivo de possibilitar a ligação entre operadores (ver figura 4.6 e 4.7). As transições restringem a possibilidade de se ligar quaisquer outros componentes que não os descritos no modelo, ou seja, a transição de um sensor para uma tabela virtual poderá ser concretizada, no entanto de uma tabela virtual para um sensor não.

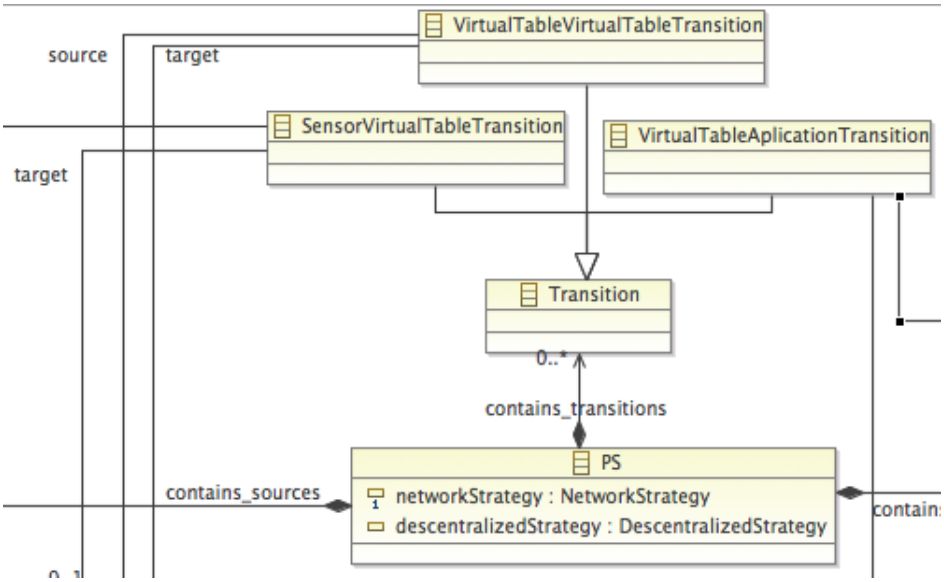


Figura 4.6: Diagrama do modelo *ecore* - transições

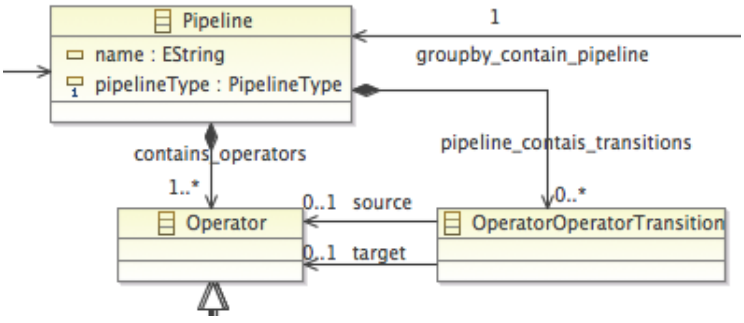


Figura 4.7: Diagrama do modelo *ecore* - transição entre operadores

As enumerações dos vários tipos de dados referentes aos atributos dos componentes do modelo, como exemplo os atributos do sensor (ex. qos, sampling, tasking, etc), são

também eles descritos no modelo *ecore*. A figura 4.8 enumera cada uma das possibilidades permitidas por cada tipo.

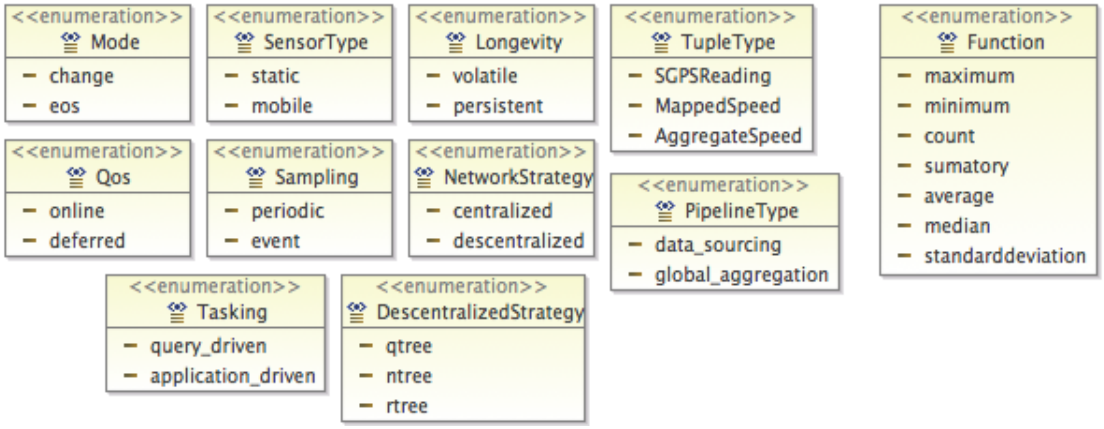


Figura 4.8: Diagrama do modelo *ecore* - discriminação dos tipos referentes aos atributos dos vários elementos do diagrama

Posteriormente, quando finalizada toda a estrutura de classes do modelo *ecore* são gerados os ficheiros para o desenvolvimento da parte gráfico do editor (*.gmfgraph*, *.gmfmap* e *.gmftool*) assim como um pacote com todo um conjunto de classes java correspondentes ao modelo. Estas classes serão utilizados mais tarde na ligação framework-simulador para instanciar elementos do ficheiro *XMI*. Os ficheiros *XMI* irão guardar a estrutura de toda a aplicação desenhada pelo utilizador. Mais à frente veremos que um modelo simples e consistente trará facilidades na leitura e interpretação destes ficheiros. Na secção 4.3.3 será dado ênfase às várias obrigatoriedades e restrições presentes no modelo para validação do ficheiro *XMI*.

4.3 Editor

Os utilizadores da aplicação deverão ter uma noção do seu funcionamento, sabendo quais os elementos disponíveis mas também como cada um deve ser instanciado de forma a conseguir o resultado esperado. Aquando a utilização desses elementos, o utilizador deverá ter noção do que é obrigatório e opcional, e suas compatibilidades/restrições. Nesta secção é descrito: uma visualização geral de todo o conteúdo e disposição da framework, os vários elementos disponibilizados, de que forma estes se apresentam, onde podem ser utilizados, quais as normas de utilização, etc.

O editor gerado por estas ferramentas, tem todo um aspecto da ferramenta Eclipse IDE. Portanto, para quem já estiver familiarizado com este IDE, torna-se mais fácil lidar com o editor, de forma intuitiva. Do lado esquerdo do editor temos uma janela que nos permite gerir os vários projetos. Dentro de cada projeto teremos os dois ficheiros responsáveis pela implementação que está a ser desenvolvida. Um ficheiro com o diagrama que ilustra o funcionamento da aplicação e um ficheiro com informação textual

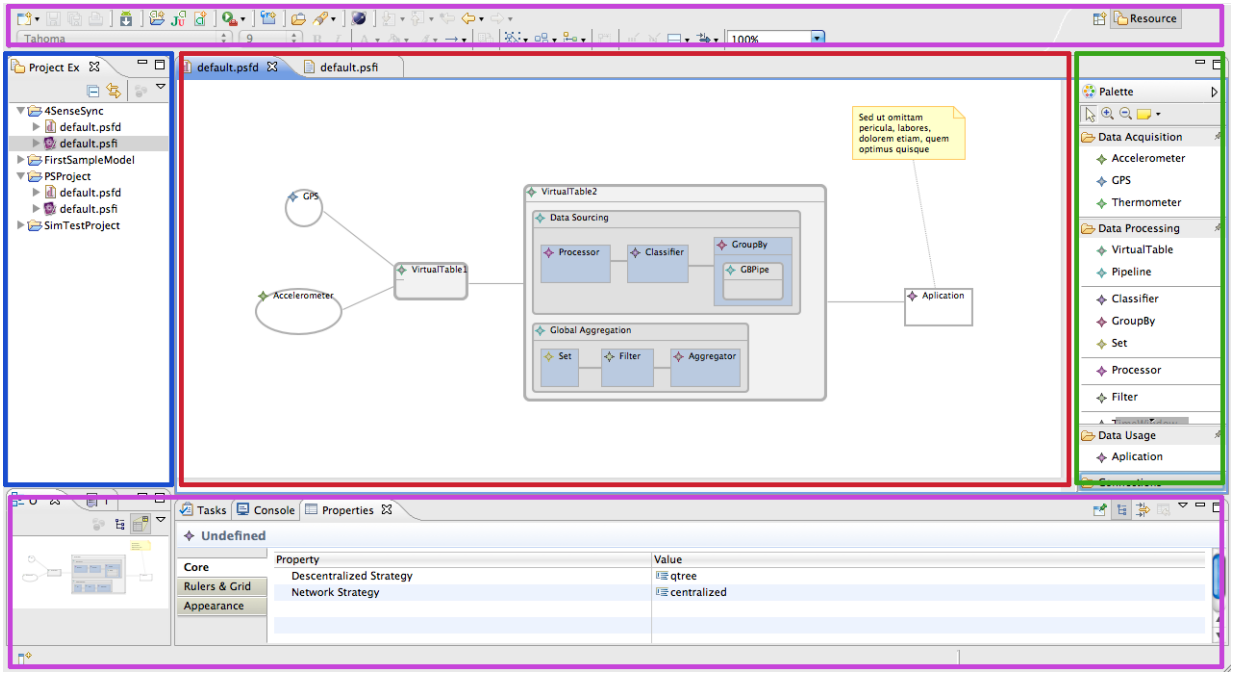


Figura 4.9: Visão global do editor.

do diagrama num formato *XMI*, com extensões *.psfd* e *.psfi*, respectivamente. Sempre que ocorre alguma modificação no diagrama, as alterações serão automaticamente reflectidas no ficheiro *XMI*. Do lado direito da framework temos a *barra de ferramentas* com os vários elementos disponíveis, estando estes separados por grupos. Ao centro temos a *tela* onde será feito a implementação gráfica da nossa aplicação a partir dos elementos disponibilizados pela *barra de ferramentas*. O framework permite ainda diferentes visualizações do diagrama, definição de propriedades dos elementos, utilização de atalhos na execução de ações frequentes, etc. A posição destas janelas não tem necessariamente de ser esta, podendo ser modificado mediante o gosto de cada utilizador. A figura 4.9 mostra uma visão global do editor, dividida segundo as janelas descritas.

4.3.1 Tela

Como dito, a tela é identificada como a classe principal do modelo *ecore* (classe *PS* deste modelo *sensoriamento participado*), onde serão adicionados os vários elementos do editor. Na propriedades da tela poderemos parametrizar todos os atributos declarados na classe principal, neste caso associados à parametrização do simulador.

Através dos ficheiros *.gmfgraph* e *.gmfmap* são atribuidos tipos de objetos a cada componente do modelo *ecore*. O tipo de objecto que é atribuído a cada componente do modelo, estabelece todas as suas propriedades quando aplicado à tela, nomeadamente a sua representação e comportamento. Os 3 tipos de objeto utilizados na composição do editor foram: *ligações*, *nós* simples e *compartimentos*.

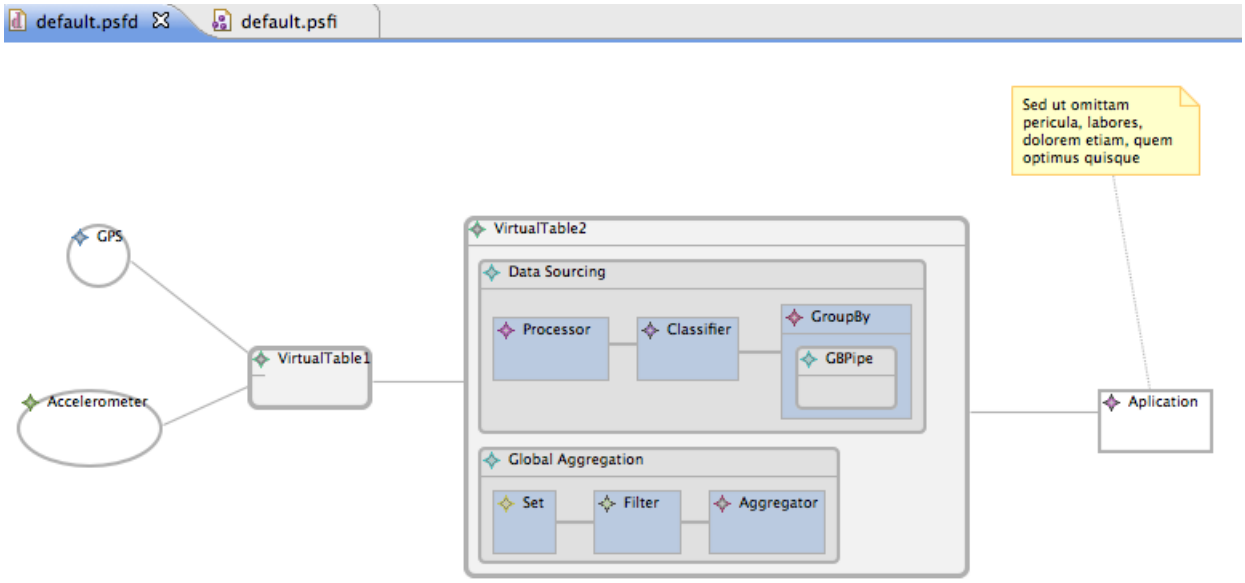


Figura 4.10: Representação da tela e seus respectivos elementos.

Todas as transições presentes no modelo *ecore* foram identificadas como *ligações*, conectando *nós* e *compartimentos*. Em relação aos outros componentes, à exceção da tabela virtual, do *pipeline* e do operador *groupby*, todos foram identificados como *nós*. Entre estes distingue-se apenas a representação gráfica de cada um na tela (elipse para sensores e retângulos para operadores e aplicação). A sua função consiste apenas numa representação gráfica, básica, da instancia do componente. Mais complexos são os objetos do tipo *compartimento* associados às tabelas virtuais, ao *pipeline* e ao operador *groupby*. Estes podem conter outros objetos no seu interior, nomeadamente, *nós*, *ligações* ou até mesmo outros *compartimentos*, formando um *subdiagrama*. No caso de uma *tabela virtual*, podemos ter *pipelines* no seu interior, um máximo de dois, correspondentes ao *data sourcing* e *global aggregation*. Por sua vez, um *pipeline* pode conter *ligações*, *nós* e *compartimentos*. Os operadores *groupby* devem novamente definir um *subpipeline* no seu interior. A figura 4.10 ilustra (apenas como exemplo, criado de forma aleatória) todos estes elementos e funcionalidades mencionadas. A ferramenta GMF, quando gerando um editor, adiciona automaticamente a possibilidade de associar notas/comentários a cada elemento da tela. As propriedades de qualquer um dos elementos colocados na tela, incluindo os seus atributos, podem ser definidas selecionando o respectivo elemento e editando os diversos campos exibidos na janela das propriedades.

4.3.2 Barra de ferramentas

A *barra de ferramentas* da framework é criada através do ficheiro *.gmftool*, definindo-se grupos para os vários elementos que deverão ser disponibilizados ao utilizador. A intenção consiste em dar a entender ao utilizador de forma intuitiva a principal funcionalidade

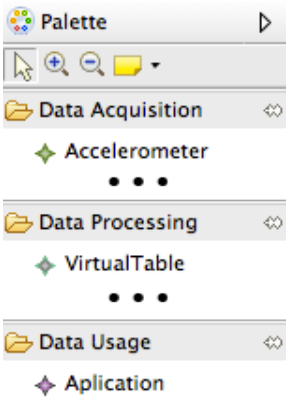


Figura 4.11: Representação da barra de ferramentas e seus respectivos elementos.

dos elementos que se encontram em cada grupo. A separação dos elementos disponibilizados pela framework foi feita em 3 grupos:

- *aquisição de dados*, composta por elementos que se destinam à aquisição de dados através da monitorização do exterior. Ou seja, baseia-se nos diversos sensores presentes no modelo *ecore*, por agora, GPS, Acelerómetro e Termómetro;

- *processamento de dados*, onde se encontram todos os elementos capazes de manipular dados, diretamente (operadores) ou indiretamente (tabelas virtuais e pipelines);

- *utilização dos dados*, composta apenas pelo elemento *aplicação*, sendo neste momento o único objeto da framework capaz de consumir informação capturada e processada pelo sistema, através da execução de consultas.

A *barra de ferramentas* dispõe também, por defeito, de ferramentas de ampliação e adição de notas.

O mapeamento entre um elemento da barra de ferramentas e o respectivo elemento na tela é feito no ficheiro *.gmfmap*, por exemplo, de maneira a que o editor saiba que o objecto "Virtual Table"do grupo "Data Processing"da barra de ferramentas, quando adicionado à tela, corresponderá a um compartimento.

4.3.3 Normas de desenvolvimento

Existem certas *normas* e *restrições* na utilização dos componentes que devem ser tidas em conta quando interagindo com o framework. Muitas delas são estabelecidas pelo modelo *ecore*. Outras podem ser alcançadas através da adição de OCL no respectivo modelo. A *validação* do diagrama da aplicação pode ser feita a partir da framework, selecionando a opção de validação no ficheiro *XMI* (formato .pdfi), verificando a consistência do mesmo relativamente às normas impostas. De momento algumas das restrições estão a ser verificadas directamente no código que faz a ligação editor-simulador. No entanto, estas

poderão, no futuro, ser verificadas através da inserção de OCL no modelo *ecore*. As normas a seguir pelo utilizador da framework no desenvolvimento gráfico de aplicações são:

- haver sempre forma de captura de dados, ou seja, pelo menos um elemento do tipo *sensor* presente na plataforma *sensoriamento participado* desenvolvida;
- qualquer plataforma necessita ter um elemento *aplicação*;
- caso se verifiquem dois pipelines e, não mais que dois, estes deverão ser de tipos diferentes, *data sourcing* e *global aggregation*;
- caso a aplicação seja definida como *descentralizada*, será obrigatório haver um *pipeline* do tipo *global aggregation*, para além de *data sourcing*;
- necessidade de se especificar os tipos de tuplo relativos ao input e output de cada *tabela virtual*;
- deve ser tido em conta a não existência de *loops* entre *tabelas virtuais*. Imaginemos uma transição de uma tabela virtual, VT1, para outra, VT2, e novamente outra transição de VT2 para VT1; este cenário iria causar um loop entre estas tabelas.
- a todas as tabelas virtuais adicionadas à tela tem de ser dado um nome, único, de forma a poderem ser posteriormente identificadas e criadas as classes necessárias a cada uma das tabelas por parte da implementação que liga o editor ao simulador;
- uma tabela virtual tem de conter todos os inputs identificados como obrigatórios;
- os operadores que contém condições como atributo, assim como o atributo *code* do operador *processor*, devem ter estes preenchidos obrigatoriamente. Todos os outros, caso não especificados, tomaram valores *default*;

Existem ainda outras normas a ter em conta relativamente à coerência entre os *inputs* e *outputs* entre fontes de dados. Estas serão referidos na secção 4.5.

4.4 Inserção de código

Poderá haver a necessidade da introdução de código por parte do utilizador em certos componentes durante o desenvolvimento da aplicação. Um exemplo desses componentes é o operador *processor*, utilizado para processamentos de dados específicos que o utilizador pretenda executar. Quando for detectado a intenção de implementar código por parte do utilizador, deverá ser apresentado uma janela para esse efeito. No entanto, deve ser dado contexto ao utilizador para que este se enquadre no código que está a ser desenvolvido graficamente por ele, percebendo o que é possível implementar naquele específico segmento de código. O utilizador deverá ter noção do tuplo a que tem acesso de momento, que informação este guarda e, como é normal algum conhecimento a nível de

programação (neste caso Java/Groovy), mesmo que básico .Este contexto pode ser dado fazendo parte de uma interpretação ao ficheiro XMI e assim criar uma estrutura para o código que está a ser desenvolvido graficamente.

4.5 Comparação e coerência entre *inputs* e *outputs* de componentes

É importante salientar que não houve implementação nenhuma nesta matéria e a intenção deste raciocínio não é oferecer uma solução precisa no que diz respeito a comparação de tipos mas sim expor o problema em causa, a sua importância nesta *framework* e algumas ideias. Esse trabalho pormenorizado deve ser feito por profissionais na área de sistemas e comparação de tipos. A importância de uma análise mais pormenorizada nesta área passa também pela prevenção da ocorrência de erros de execução da aplicação, trazendo um desenho regular e ortogonal a uma linguagem.

Para que os componentes do *framework-specific DSL gráfica* possam processar de forma correta a informação que recebem, existe a necessidade de manter a compatibilidade e a coerência no fluxo de dados partilhado entre eles. Imaginando um componente A que se pretende ligar a um componente B, o utilizador deverá definir qual o *output* do componente A e, qual o *input* do componente B, caso estes sejam compatíveis a ligação entre eles será possível. Para que haja compatibilidade não é necessário que o componente A faculte a B, todos os tipos de tuplo que este necessita para executar. O componente A poderá apenas preencher certos requisitos definidos no *input* de B, podendo ser os outros preenchidos por outros componentes do sistema. Ou seja, para ser feita a ligação entre A e B, o conjunto de tipos de tuplo do *output* de A tem de estar, pelo menos, contido no conjunto de tipos de tuplo do *input* do componente B.

Inputs e *Outputs*

Os componentes, como por exemplo as tabelas virtuais, devem definir o seu *input* e o seu *output* para que seja possível determinar quais os dados que este consome e, quais os dados que este gera. No *input* de um componente o utilizador deverá definir todos os tipos de tuplo necessários à execução do respectivo componente. No caso do *output* o utilizador pode optar por definir tipos de tuplo já existentes no sistema ou, criar um novo tipo de tuplo - tipo de tuplo *complexo* (ver figura 4.12). A criação deste tuplo é feita através de outros tipos de tuplo já existentes – tipos de tuplo *base* ou também *complexos*. Por exemplo, existindo um tipo de tuplo "GPSReading", base, e outro "AggregatedSpeed", complexo, o utilizador poderá criar um novo tipo de tuplo que consiste na junção destes dois.

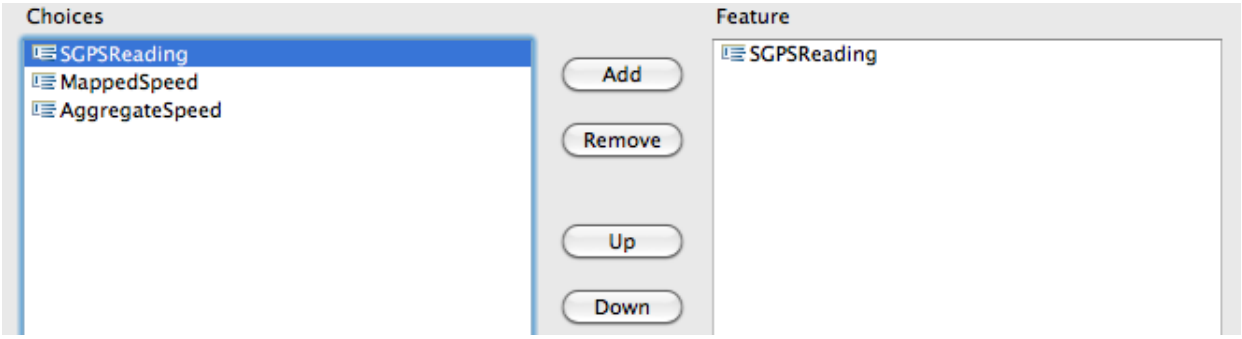


Figura 4.12: Exemplo de interface para definir quais os tuplos que constituem o *input*/*output* duma tabela virtual.

Tipos de tuplo *base* e *complexo*

A ideia é existir duas noções de tipos de tuplo no sistema: tipos de tuplo base e tipos de tuplo complexo. Tipos de tuplo *base* são tipos instanciados pelo sistema, de início, que provêm dos vários sensores disponíveis na *framework*, assim como outros tipos de dados comuns, como, *double*, *int*, etc. Tipos de tuplo complexos, são tipos gerados através de tipos de tuplo base, definindo novos tipos de tuplo no sistema. Por sua vez, tipos tuplo complexos também podem ser constítuidos por tipos de tuplo complexos e assim sucessivamente.

Inputs do tipo *obrigatório* e *opcional*

Os *inputs* definidos num componente podem ser *obrigatórios* ou *opcionais*. Como o nome indica, um *input* obrigatório é estritamente necessário à execução do respectivo componente. Caso haja *inputs obrigatórios* não conectados ao componente este não poderá ser executado. Por outro lado, se algum *input obrigatório* não estiver a ser utilizado durante todo o funcionamento interno do componente, deve ser exibido um aviso por parte da *framework*, alertando o utilizador. Em relação aos *inputs opcionais*, estes podem, ou não, ser adicionados ao componente alterando o seu comportamento consoante os dados que lhe forem facultados.

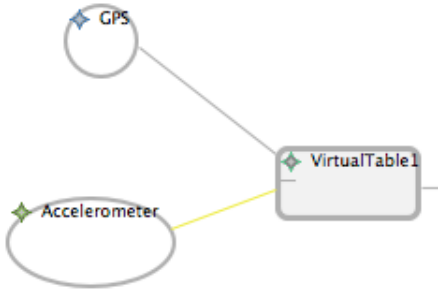


Figura 4.13: Exemplo de aviso - transição a amarelo - de *input* não utilizado pela tabela virtual ao longo dos seus pipelines.

(a) Caso de comparaçao direta satisfazível

(b) Caso de comparaçao direta não satisfazível

(c) Caso de comparaçao direta não satisfazível

Figura 4.14: Exemplos de comparação direta.

Casos de comparação

1º Caso – Comparação Directa

No primeiro caso, o *output* do componente que se pretende conectar, consiste num tipo de tuplo presente nos tipos de tuplo definidos no *input* do componente alvo, sendo a comparação direta e simples. Basta apenas verificar se esse mesmo tipo de tuplo base está contido no conjunto de tipos de tuplo do *input* do outro componente (ver figura 4.14(a).

2º Caso – Comparação Interna

No segundo caso de comparação, o *output* do componente consiste num tipo de tuplo complexo. Caso esse tipo esteja presente no conjunto de tipos de tuplo do *input* do outro componente, a execução será idêntica à do 1º caso de comparação (ver figura 4.15(a)), caso contrário, implica uma análise interior do tipo do *output*, analisando os seus *subtipos*, sucessivamente, verificando se este contém algum subtipo compatível com algum tipo de tuplo presente no *input* (ver figura 4.15(b)).

4.5.1 Métodos de comparação

O segundo caso de comparação exige uma comparação elaborada, para tal foram pensados dois métodos para verificação de compatibilidade interna de um tipo. Um método a partir de atributos <chave, valor> de uma tabela de dispersão e outro através de herança múltipla de classes. Como os tipos de tuplo complexos terão normalmente uma quantidade moderada no sistema, as complexidades destes métodos nunca atingirão proporções elevadas.

(a) Caso de comparaçao interna satisfazível

(b) Caso de comparação interna satisfazível

(c) Caso de comparação interna satisfazível

(d) Caso de comparação interna não satisfazível

Figura 4.15: Exemplos de comparação interna.

4.5.1.1 1º Método

O primeiro método, mais simples, consiste numa tabela de dispersão, em que a chave corresponde a um tipo de tuplo complexo e o valor aos respectivos subtipos que o constituem. Recorrendo ao exemplo anteriormente dado, podemos pensar que o componente A pretende ligar-se a B que possui um tipo de tuplo T no seu *input*. Na altura de comparação procede-se com o primeiro caso de comparação, verificando se o *output* de A é do tipo de tuplo T. Caso seja, fica resolvido e faz-se a ligação, caso contrário, verifica-se na tabela se o valor, sendo este um conjunto, correspondente à chave do *output* de A, contém algum tipo de tuplo T. Caso não haja e, o conjunto contenha outros tipos de tuplo complexos, aplica-se a mesma ação a cada um deles. Assim sucessivamente até se encontrar, ou não, um tipo de tuplo T. Este é um método de fácil implementação mas, por outro lado, pode exigir muitas iterações e chamadas à estrutura de dados. O número de entradas na tabela pode variar consoante o processo de inserção utilizada.

4.5.1.2 2º Método

O segundo método apoia-se sobre a funcionalidade de herança múltipla entre classes. A criação de um novo tipo de tuplo complexo implica a criação de uma nova classe que irá estender os subtipos de tuplo que o constituem. Um tipo de tuplo T constituído pelos subtipos T1 e T2, implica uma classe T estendendo T1 e T2. Na altura de comparação, quando um componente se pretende conectar a outro, é necessário verificar se o seu *output* herda algum dos tipos de tuplo presentes no *input* do segundo.

Mais uma vez, imaginando um componente A que se pretende ligar a um componente B que possui um tipo de tuplo T no seu *input*, terá de se verificar se o *output* de A é igual a T e, caso não o seja verifica-se se é uma instância de T, ou seja, se herda T. Este método apresenta uma implementação mais complexa que o anterior. Ainda em relação a herança múltipla, esta é uma possível característica de uma classe, numa linguagem orientada por objetos, que estende/herda comportamentos e características de mais do que uma SuperClass. Linguagens como por exemplo C++, OCaml e Python suportam a utilização de múltipla heranças. O mesmo não acontece em Java, em que os seus designers sentiram que implicaria uma maior e desnecessária complexidade. No entanto é possível contrariar esta falha através de herança múltipla de interfaces. Na verdade quando se diz que Java não suporta herança múltipla, estamos a dizer que esta apenas não suporta herança múltipla de implementações [18].

4.6 Ligação Framework/Simulador 4Sensing

A ligação entre framework-simulador consiste numa sequência de passos desde a aquisição do ficheiro *XMI* criado pela framework, até à criação das várias classes que irão compor o cenário de simulação do sistema 4Sensing. O esquema de classes utilizador neste processo é ilustrado pela figura 4.16.

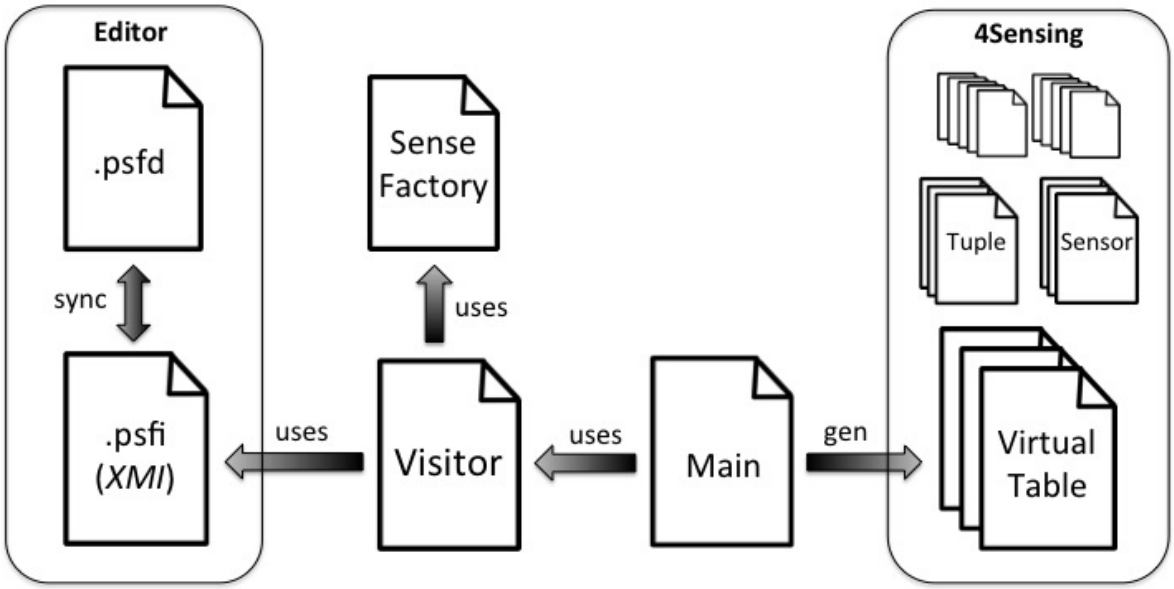


Figura 4.16: Esquema da ligação framework-simulador.

Enquanto é feito o desenvolvimento gráfico de toda a aplicação *sensoriamento participado*, está a ser gerado um ficheiro *XMI*, equivalente ao diagrama mas de forma textual. Após validação do diagrama , o ficheiro é utilizado para termos acesso a toda a estrutura da aplicação e assim, saber o conteúdo das classes que serão criadas para completar o simulador 4Sensing. A classe que inicia este processo, *Main.java*, começa por criar um

objecto do tipo *Visitor*. Este quando instanciado começa o seu processo de interpretação do ficheiro *XMI*. Vai analisar quais as tabelas virtuais existentes e qual o input, ouput e conteúdo de cada uma. À medida que é feita esta análise vão sendo instanciados os elementos do ficheiro, através do pacote gerado anteriormente a partir do modelo *ecore* (referido em 4.2), e gerados novos objetos (do tipo *Sense*) com o objectivo de facilitar a escrita das classes 4Sensing. Esta geração de novo objetos a partir das instancias do modelo são feitas com o auxílio da classe *SenseFactory.java*. Esta compõe-se de métodos públicos capazes de "fabricar"elementos do tipo *Sense* a partir de objetos instancias do modelo. A inicialização da classe *Visitor.java* termina com estruturas de dados preenchidas e organizadas com a informação necessária à escrita das classes 4Sensing. Acedendo a essa informação a classe *Main.java* inicia o processo de criação e posicionamento de cada ficheiro no projeto do simulador 4Sensing. Estes ficheiros consistem em classes em linguagem *Groovy*, cada um deles representando uma tabela virtual. Terminando a execução da classe *Main.java* temos o simulador pronto a executar. De momento, a sequência de passos: validação do ficheiro *XMI*, processo de interpretação do *XMI* e criação das classes groovy, e finalmente execução do simulador; tem de ser feita manualmente, fazendo parte do trabalho futuro a automatização de todo este processo.

4.7 Contexto da Simulação

O simulador 4Sensing exige o desenvolvimento de um cenário de simulação. Os passos necessários na criação deste cenário são: definição do *tipo de nó móvel*, definição das *tabelas virtuais* e *tuplos*, e *parametrização* do simulador. Tanto o tipo de nó móvel como as tabelas virtuais são possíveis de especificar a partir do framework desenvolvido. A definição dos tuplos consiste em trabalho futuro como identificado no capítulo BLA. A parametrização do simulador consiste na especificação de propriedades que irão definir a execução do simulador, a forma como este irá estruturar e executar a rede de sensores. Esta parametrização deve ser disponibilizada pelo editor ao utilizador. A parametrização do simulador consiste em: *configuração* do simulador, definição da *pesquisa* a executar e forma de *ouput*. A definição da *pesquisa* está associada ao elemento *aplicação* do editor, sendo este a componente responsável pelas consultas no sistema. As restantes características associadas à parametrização do simulador são debatidas em 4.7.1 e 4.7.2.

4.7.1 Configuração

Existem vários parâmetros possíveis de configurar neste simulador, no entanto, serão identificados apenas o que se mostram relevantes do ponto de vista do utilizador que pretende desenvolver uma aplicação. Todos estes parâmetros possuem valores *default* utilizados quando não definido nenhum valor para os mesmos. A tabela 4.1 enumera e descreve todos eles. Os valores sem valor *default* devem-se a falta de informação relativamente à plataforma 4Sensing.

Parâmetro	Função	Default
NETWORK_STRATEGY	Tipo de estratégia de rede a utilizar	Centralized
DESCENTRALIZED_STRATEGY	Tipo de estratégia descentralizada a utilizar	NTREE
VT_WINDOW_SIZE	Tamanho da área geográfica abrangida pela consulta	-
RUN_TIME	Tempo de execução da simulação em segundos	0
IDLE_TIME	Intervalo de tempo entre o início da simulação e a execução da consulta	0
SIM_TIME_WARP	Determina velocidade de execução do simulador relativamente a tempo real	1E9
TOTAL_NODES	Número de nós na infraestrutura fixa 4Sensing	250

Tabela 4.1: Parâmetros a configurar

Estes parâmetros deverão ser todos atributos adicionados à classe "principal"do modelo *ecore*. Posteriormente, uma vez gerados, novamente, os ficheiros que compõem o editor (*.gmfgraph*, *.gmftool*, etc) , estes atributos estarão acessíveis ao utilizador, através das propriedades da tela. Na ligação ao simulador os valores destes atributos serão capturados e transmitidos ao simulador aquando a sua execução.

4.7.2 *Outputs*

Em qualquer simulação pretende-se gerar resultados avaliando o desempenho da aplicação e se estes equivalem ao resultado esperado. È por isso importante especificar uma forma de *output* para esses resultados através do editor. Esta funcionalidade pode ser alcançada através da adição de elementos ao modelo *ecore* que representem vários tipos de *output*, dando-lhes uma representação gráfica no editor. A barra de ferramentas deverá disponibilizar um grupo de subferramentas destinado apenas ao *output* de valores. Estes elementos devem ser adicionados e representados na tela através de *nós* simples ligados ao elemento *aplicação* através de transições.

As diferentes formas de *output* passam por: registo simples em *ficheiros*, representação em *tabelas* ou ambientes *gráficos*. As últimas duas formas de visualização implicam um pós-processamento de todos os resultados registados em ficheiros, deduzindo-se algo mais em concreto em relação aos mesmos. Como exemplo, seria interessante construir um gráfico, relativo à aplicação Cartel, que disponibilize uma estimativa das horas de maior e menor congestionamento ao longo do dia.

Neste momento o *output* devolvido pelo simulador é feito por completo na consola. Será necessário criar elementos no editor que expressem as várias possibilidades de *output* e, de seguida, tornar possível a sua interpretação por parte do código responsável

pela ligação ao simulador.

A nível de editor será necessário criar novos componentes no modelo *ecore* e as possíveis transições com os restantes. Editado o modelo terá de se passar pelo processo de edição e mapeamento nos ficheiros *GMF* de forma a associar os novos componentes do *ecore*, a novos elementos na barra de ferramentas e na tela do editor. Na ligação ao simulador estes novos elementos do editor terão de ser interpretados transmitindo certos parâmetros ao simulador para que este gere o *output* escolhido. Para já a plataforma 4Sensing simulador encontra-se preparado para apresentar resultados através de um ficheiro ou gráfico.

4.8 Limitações

Apesar de desenvolvida a framework gráfica pretendida, e esta se encontrar funcional, permitindo o desenvolvimento e simulação de aplicações, existe consciência que esta mostra ainda algumas limitações. O objetivo desta secção consiste em enumerar quais são essas limitações e, de que forma poderão ser ultrapassadas com trabalho futuro.

4.8.1 Operador *aggregate*

O operador *aggregate* é composto por um conjunto de operações executadas sucessivamente (secção 3.3.7) e/ou por código definido pelo utilizador. O editor desenvolvido oferece uma interface (ver figura 4.17) de forma a ser possível definir qual a sequência de operações. No caso de uma operação, esta tem os seus próprios argumentos que devem também eles ser especificados pelo utilizador. Por exemplo, caso escolhida a função *somatório*, deverá ser indicado qual a variável a ser somada e, qual a variável que irá conter o resultado das consecutivas somas. Este conjunto de variáveis possíveis de selecionar será dinâmico, dependendo do input e output deste componente. Por exemplo, no caso da tabela virtual *TrafficSpeed* analisada em 5.3.1, os argumentos utilizados foram *"speed"*, atributo correspondente ao tuplo *MappedSpeed* e, *"sumSpeed"*, correspondente a *AggregateSpeed*. Neste momento estas variáveis estão a ser inseridas diretamente no código. No futuro, devem ser especificadas pelo utilizador.

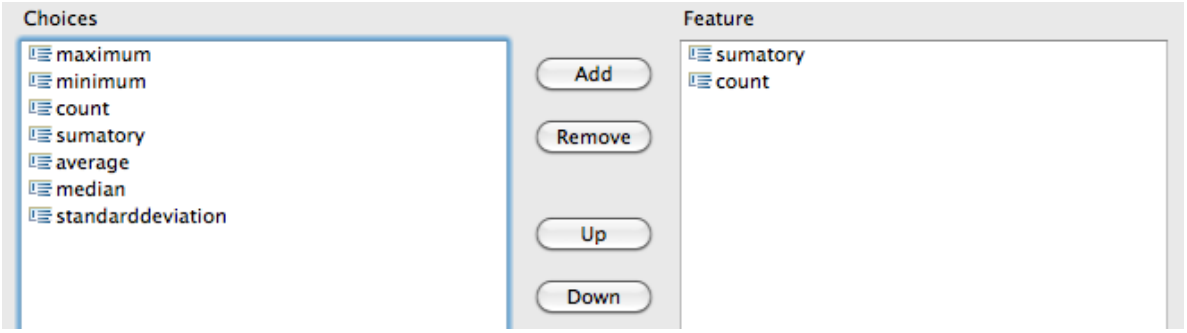


Figura 4.17: Visualização da interface de escolha de operações a executar pelo operador *aggregrate*.

4.8.2 Condições

Existem operadores, como *groupby*, compostos por condições, que consistem em atributos de um determinado tuplo. A listagem 5.1, referente à tabela virtual *TrafficCount* mostra a utilização do operador *groupby* tendo como argumento, o atributo *"segmentId"* do tuplo MappedSpeed. Deverá ser definido uma interface que possibilite ao utilizador a escolha de um ou mais atributos que farão de condição quando aplicado o componente. O conjunto de variáveis possíveis de escolher dependerá do tipo de tuplo associado ao *input* do componente.

4.8.3 Criação e seleção de tuplos

O editor deverá disponibilizar ao longo do desenvolvimento de uma aplicação a possibilidade de serem criados e removidos tipos de tuplo do sistema. Esta criação passa por definir os vários atributos pelos quais o tipo de tuplo será composto. Poderá consistir num tipo tuplo base, ou complexo. Sempre que necessário ao utilizador a escolha de um tuplo para *input* ou *output* de algum componente, deve ser apresentado a lista de todos os tipos de tuplo conhecidos até ao momento pelo sistema. Esta lista deve, portanto, ser dinâmica, consoante os tipos de tuplo presentes no sistema num dado momento. De momento a lista de tipos de tuplo disponibilizada pela framework (figura 4.12) é estática, tendo sido os tipos de tuplo adicionados manualmente ao modelo *ecore* (componente *TupleType*, figura 4.8).

4.9 Sumário

Neste capítulo é descrito todo o processo de desenvolvimento da DSL gráfica específica para uma framework no contexto da plataforma 4Sensing.

O processo de desenvolvimento de uma aplicação neste domínio implica 3 fases: edição gráfica, a compilação da estrutura da aplicação desenvolvida e a simulação da mesma por parte do simulador *4Sensing*. Ao longo deste capítulo foi descrito todo o trabalho feito de forma a possibilitar estas 3 fases.

A criação do editor, feita a partir da ferramenta *EMF* e *GMF*, foi iniciada com um ficheiro de formato *.ecore*, criado a partir de um diagrama composto pelos vários componentes necessários ao funcionamento da DSL gráfica. A partir do Ecore são gerados e, posteriormente editados, vários ficheiros, como, *.gmfgraph*, *.gmftool*, *.gmfmap*, etc. Estes especificam toda uma forma que o editor irá ter. Existem outros aspectos importantes como a necessidade de inserção de código, a coerência entre *inputs* e *outputs* de dados que devem ser tidos em conta nesta framework, entre outras.

Em relação à compilação, esta começa com a interpretação de um ficheiro *XMI*, criado por parte do editor, contendo toda uma implementação da aplicação desenvolvida. Esta implementação é refletida em classes, mais tarde integradas na plataforma 4Sensing para que possa ser executada a simulação.

5

Validação

Este capítulo pretende fazer uma demonstração e validação da *framework-specific DSL* desenvolvida, implementando exemplos de aplicações em *sensoriamento participado*. Ambos os exemplos foram adaptados ao cenário *SpeedSense* (discutido em 5.1), estudado no projeto 4Sensing e incorporado no simulador. A partir deste cenário criado pelo simulador foram pensados dois exemplos de aplicações, descritos em 5.2 e 5.3, expondo o desenvolvimento de cada uma - suas componentes e parametrizações - e respectiva simulação. Ambas as aplicações irão utilizar apenas um tipo de sensor (neste caso *GPS*) como forma de captura de dados para a aplicação e, uma tabela virtual para os manipular.

5.1 Cenário SpeedSense

O cenário *SpeedSense*, utilizado para demonstração da plataforma 4Sensing, consiste na simulação de uma área urbana. Neste caso a monitorização é feita através de sensores GPS de telemóveis transportados pelos condutores. Através desta captura de dados é possível inferir vários tipos de informação relativamente ao tráfego urbano. De momento o simulador 4Sensing encontra-se adaptado a este cenário da vida real. Este cenário inclui os seu próprios sensores (neste caso apenas GPS), tipos de tuplo, visualização gráfica, etc. Ambos os exemplos de aplicações que serão apresentados implicaram a utilização destes sensores e tipos de tuplo, divergindo apenas na forma como manipulam os dados, consoante a informação que desejam extrair. Os tipos de tuplo utilizados neste cenário são: *SGPSReading*, representa uma estruturação dos dados capturados e transmitidos pelo sensor GPS; *MappedSpeed*, mapeia a informação GPS num segmento geográfico do mapa do simulador; e *AggregateSpeed*, agrega diversos tipos de informação relativamente à velocidade num segmento geográfico.

5.2 Aplicação 1

Imaginemos que uma empresa pretende saber quais as estradas de uma cidade com maior trafego num dado momento, desencadeando assim, estratégias de publicitação de um novo produto. Este primeiro exemplo de aplicação, bastante simples, poderá cumprir com esses objectivos, ilustrando uma aplicação que calcula o número de veículos por segmento de estrada, num dado momento, observando-se assim quais os de maior tráfego. Pretende-se que esta aplicação se baseie numa arquitetura centralizada. A manipulação de dados desta aplicação será feita a partir da tabela virtual TrafficCount. O conteúdo dos .psfd e .psfi, relativamente a esta aplicação são, 5.1 e 5.2.

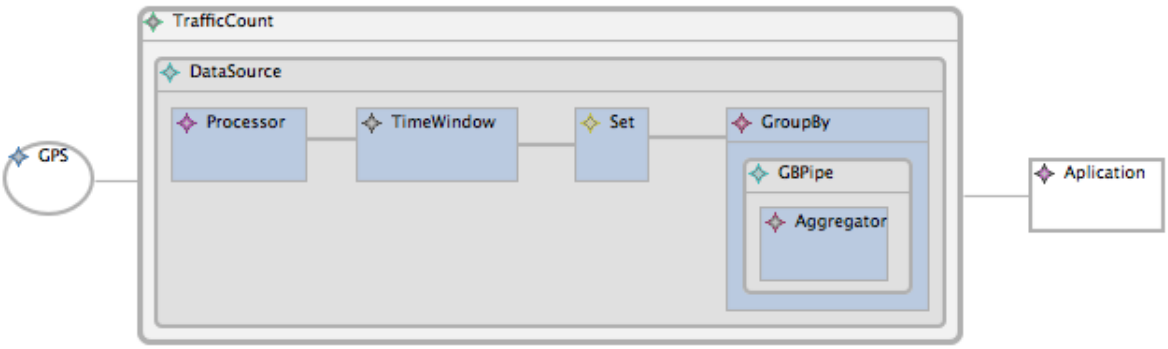


Figura 5.1: Ficheiro .psfd da Aplicação1.

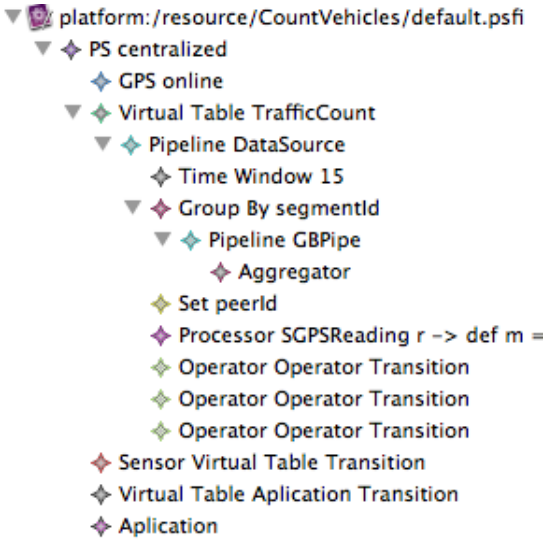


Figura 5.2: Ficheiro .psfi da Aplicação1.

5.2.1 TrafficCount

Como se trata de uma arquitetura centralizada, a sua tabela virtual, à qual atribuimos o nome *TrafficCount*, irá necessitar apenas de um *pipeline* do tipo *data sourcing* que fará

a aquisição e manipulação dos dados. 5.1 corresponde ao código gerado através da implementação gráfica desenvolvida. Esta tabela terá como input o sensor *GPS*. A sua *data sourcing* consiste nos seguintes operadores:

Listing 5.1: Código da tabela *TrafficCount*.

```
sensorInput(SGPSReading)
dataSource{
  process{SGPSReading r ->
    def m = new MappedSpeed(r);
    m.boundingBox = mapModel.getSegmentExtent(r.segmentId);
    return m;
  }
  timeWindow(mode: periodic, size:15, slide:10)
  set(['peerId'], mode: change,ttl: 10)
  groupBy(['segmentId']){
    aggregate(AggregateSpeed){MappedSpeed v ->
      count (v, 'count')
    }
  }
}
```

1) ***processor*** - mapeia coordenadas no espaço, contidas em cada tuplo *SGPSReading*, em segmentos de estrada no mapa, tuplos *MappedSpeed*. Este mapeamento é feito através de código implementado pelo utilizador e executado posteriormente por este operador;

2) ***timewindow*** - particiona um stream contínuo, constituído por tuplos do tipo *MappedSpeed*, em sequências com dimensão equivalente a *15 segundos* de transmissão. Reencaminha uma nova sequência para o próximo componente a cada *10 segundos*;

3) ***set*** - cria uma sequência sem repetições de veículos, com o tuplo mais recente correspondente a cada veículo participante;

4) ***groupby*** - gera *substreams* correspondentes a cada segmento de estrada. A cada *substream* é aplicado o seguinte operador;

5) ***aggregator*** - faz a contagem de tuplos presentes no substream, ou seja, o número de veículos por segmento de estrada. Esta informação relativamente a cada segmento de estrada é associada a um tuplo *AggregateSpeed*;

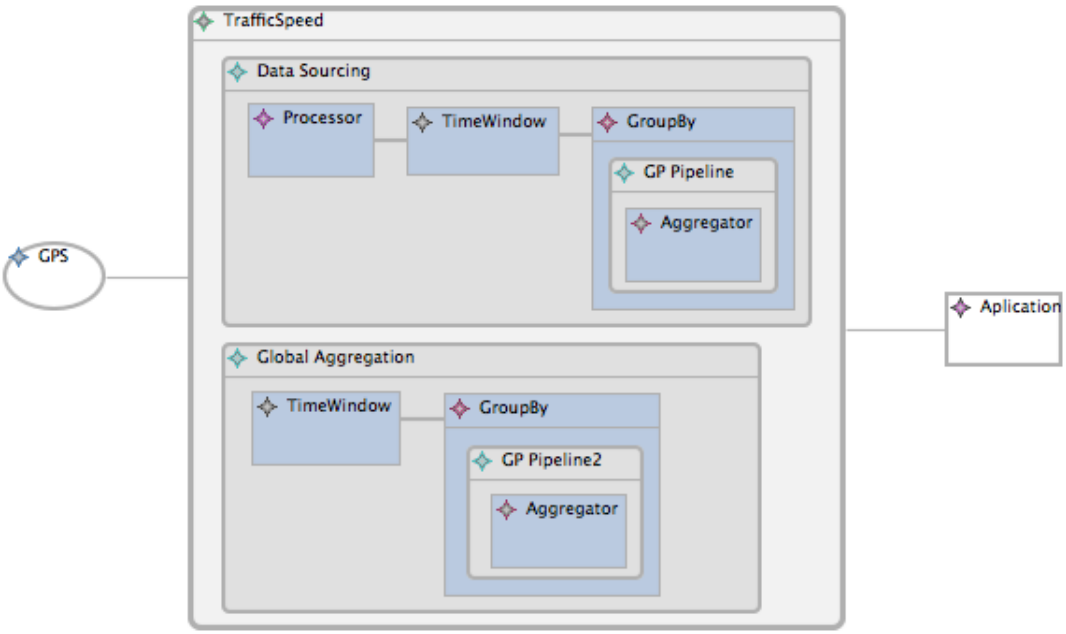


Figura 5.3: Ficheiro .psfd da Aplicação2.

5.3 Aplicação 2

O desenvolvimento desta aplicação, já abordado anteriormente para testes do simulador 4Sensing, eleva a complexidade da Aplicação1 [15]. O facto de se utilizar este exemplo de aplicação já estruturado anteriormente para testes, tem como objectivo demonstrar a possibilidade de desenvolvimento da mesma, mas agora através de uma implementação gráfica.

O desafio desta aplicação passa por descobrir a velocidade média do tráfego nos vários segmentos de estrada de uma cidade, num dado momento. Esta aplicação utilizará uma arquitetura *descentralizada* na distribuição dos dados pelos vários intervenientes do sistema. Para estratégia desta arquitetura descentralizada, optou-se, sem qualquer razão, por uma estratégia *QTREE* disponibilizada pela plataforma 4Sensing. O conteúdo dos ficheiros com a implementação da Aplicação2 podem ser vistos nas figuras 5.3 e 5.4, *.psfd* e *.psfi*, respetivamente. Para a manipulação de dados implementou-se a tabela virtual *TrafficSpeed* (5.3.1).

5.3.1 TrafficSpeed

A manipulação de dados desta aplicação é feita pela tabela virtual *TrafficSpeed*. Assim como na *TrafficCount* da Aplicação1 (secção 5.2), esta também terá como input o sensor *GPS* e assim tipos de tuplo *SGPSReading*. Ao contrário da anterior, esta aplicação utiliza uma arquitetura descentralizada, o que faz com que se definam dois pipelines para a sua execução. *Data sourcing*, para a recolha de dados a fazer por cada dispositivo abrangido

Figura 5.4: Ficheiro .psfi da Aplicação2.

pela consulta e, *global aggregation* para que o nó que efectuou a consulta, possa agregar todos os dados. 5.2 corresponde ao código gerado através da implementação gráfica feita. A sequência de operadores que irá compôr o *data sourcing* desta aplicação é:

1) ***processor*** - mapeia coordenadas no espaço, contidas em cada tuplo *SGPSReading*, em segmentos de estrada no mapa, tuplos *MappedSpeed*. Este mapeamento é feito através de código implementado pelo utilizador e executado posteriormente por este operador;

2) ***timewindow*** - particiona um stream contínuo, constituído por tuplos do tipo *MappedSpeed*, em sequências com dimensão equivalente a *15 segundos* de transmissão. Reencaminha uma nova sequência de *10* em *10 segundos* para o próximo componente;

3) ***groupby*** - gera *substreams* correspondentes a cada segmento de estrada. A cada *substream* é aplicado o seguinte operador;

4) ***aggregator*** - processa a soma das velocidades, assim como a contagem, dos tuplos presentes no substream. Ambos os valores, soma e contagem, são associados a um tuplo *AggregateSpeed*;

Listing 5.2: Código da tabela *TrafficSpeed*.

```
sensorInput(SGPSReading)
dataSource{
  process{SGPSReading r ->
    def m = new MappedSpeed(r);
    m.boundingBox = mapModel.getSegmentExtent(r.segmentId);
    return m;
  }
  timeWindow(mode: periodic, size:15, slide:10)
  groupBy(['segmentId']){
    aggregate(AggregateSpeed){MappedSpeed v ->
      sum (v, 'speed', 'sumSpeed')
      count (v, 'count')
    }
  }
}
globalAggregation{
  timeWindow(mode: periodic, size:10, slide:10)
  groupBy(['segmentId']){
    aggregate(AggregateSpeed){AggregateSpeed v ->
      avg (v, 'sumSpeed', 'count', 'avgSpeed')
    }
  }
}
```

Seguidamente, todos os tuplos *AggregateSpeed* gerados por cada dispositivo serão agregados através da seguinte sequência de operadores:

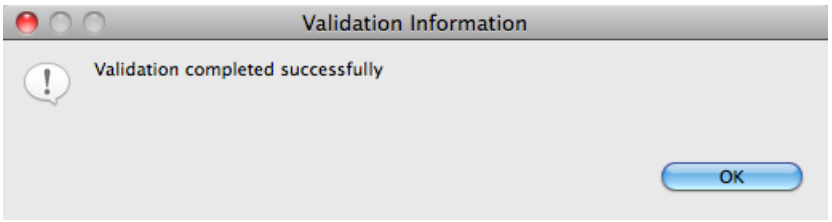
1) ***timewindow*** - particiona um stream contínuo, constituído por tuplos do tipo *AggregateSpeed*, em sequências com dimensão equivalente a *10 segundos* de transmissão. Reencaminha uma nova sequência de *10* em *10 segundos* para o próximo componente;

2) ***groupby*** - gera *substreams* correspondentes a cada segmento de estrada. A cada *substream* é aplicado o seguinte operador;

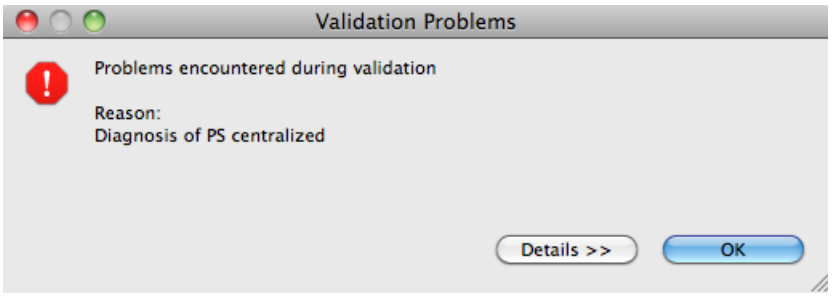
3) aggregator - calcula a média da velocidade de cada segmento de estrada a partir da soma e contagem dos vários dispositivos que responderam à consulta. Essa média de velocidade é também associada a um tuplo *AggregateSpeed*;

5.4 Sumário

Foi feito a demonstração e validação da *framework-specific DSL* através do desenvolvimento de exemplos de aplicações, ambos baseados num cenário *SpeedSense* executado a partir do simulador *4Sensing*. O primeiro exemplo de aplicação destina-se à contagem do número de veículos por segmento de estrada. O seu resultado permite-nos saber quais os segmentos com maior tráfego. A distribuição dos dados desta aplicação será feita de forma centralizada, ou seja, apenas com um pipeline, *data sourcing*, para a manipulação



(a) Janela de validação bem sucedida.



(b) Janela alertando problemas na validação.

Figura 5.5: Avisos de validação.

de dados (figuras 5.2 e 5.1). Ao contrário do primeiro exemplo, a segunda aplicação utilizada para validação, consiste numa arquitectura descentralizada, obrigando à utilização de 2 pipelines, *data sourcing* e *global aggregation* (figuras 5.4 e 5.3). Esta aplicação calcula a média da velocidade dos veículos por segmento de estrada.

Em ambas as aplicações, após a manipulação de dados, os resultados são enviados para o elemento *aplicação*, onde serão consumidos/visualizados. No final do desenvolvimento destas implementações deve ser feito a *validação* da aplicação, verificando que não existe nenhuma inconformidade. O utilizador receberá um aviso, como ilustrado na figura 5.5, em caso da validação ser bem sucedida, ou não. Após validada a implementação gráfica desenvolvida e, feito a integração do código no projeto 4Sensing, é executado o simulador. A figura 5.6 representa o decorrer de uma simulação ilustrando uma área urbana através de um mapa da cidade de Lisboa e respectivos veículos.

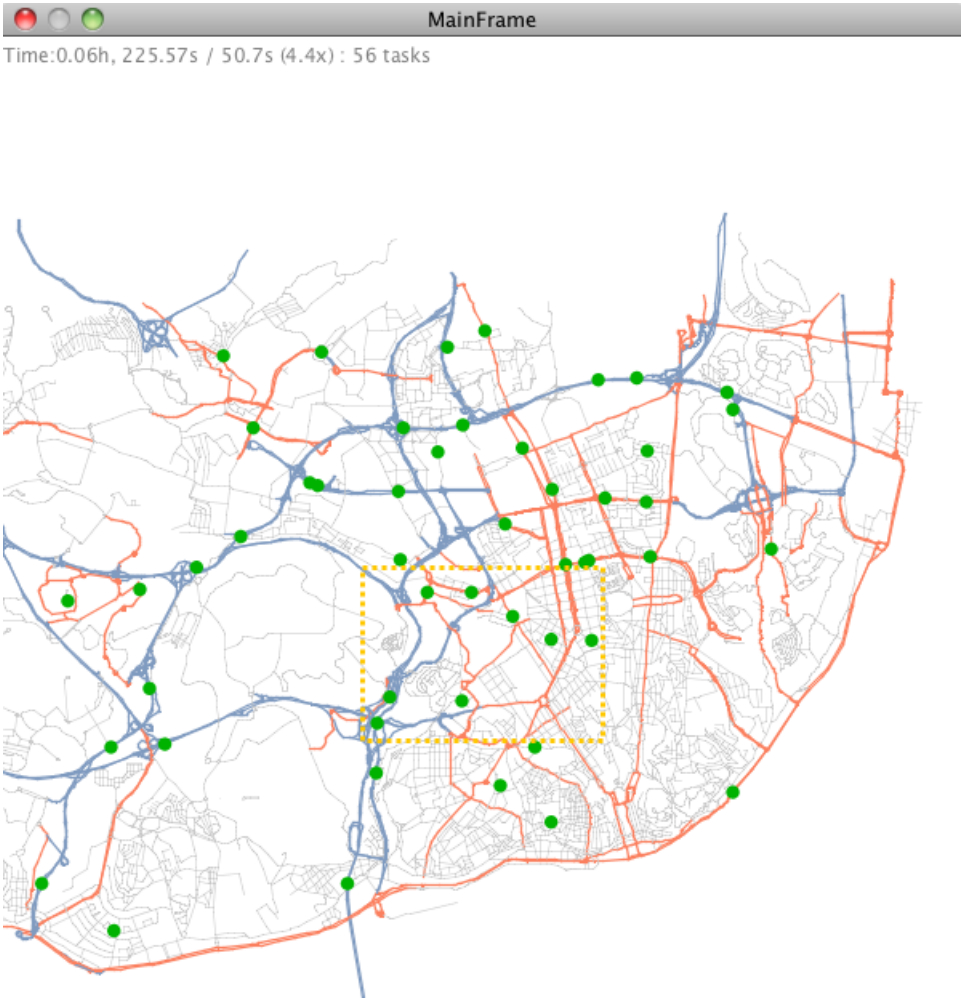


Figura 5.6: Visualização do simulador 4Sensing em execução.

6

Conclusões

O principal objectivo desta dissertação foi o desenvolvimento de uma *framework-specific DSL gráfica* no domínio de *sensoriamento participado*.

O trabalho da dissertação foi iniciado com a revisão de um anterior projeto, 4Sensing. Este descreve uma possível arquitetura de suporte neste domínio de aplicações, relativamente ao processamento de dados, tendo sido criados diversos componentes no desenvolvimento desta plataforma. A partir destes componentes e restantes características do domínio estruturou-se a arquitetura para uma *DSL*, assim como o modelo para uma *framework-specific DSL* criando-se um editor que permite o desenvolvimento gráfico de aplicações neste domínio de *sensoriamento participado*.

O decorrer deste projeto consistiu em 3 etapas: desenvolvimento de um editor gráfico baseado num meta-modelo, processo de compilação e processo de simulação da aplicação. Foi criado um editor gráfico que disponibiliza elementos necessários ao desenvolvimento duma aplicação. Estes são selecionados, parametrizados e relacionados entre si, ilustrando o funcionamento da aplicação. Terminado o desenvolvimento, a *framework-specific DSL gráfica* inicia o processo de interpretação da aplicação gerando código de integração ao simulador *4Sensing*. Integrado o código, o simulador fica responsável pela execução da aplicação criada, num ambiente simulado através de uma rede de sensores.

A *framework-specific DSL gráfica* criada permite o desenvolver de aplicações em *sensoriamento participado*, de forma intuitiva, interagindo-se com alguns elementos presentes no domínio (outros elementos requerem uma compreensão prévia do seu funcionamento e utilidade). A aplicação é executada através da simulação de uma rede de dispositivos móveis suportada pela plataforma *4Sensing*. Esta plataforma encontra-se ainda limitada no que diz respeito a cenários de aplicação sendo apenas possível a simulação de uma rede de dispositivos móveis associados aos vários veículos existentes numa cidade. De

momento, poderão ser desenvolvidos vários exemplos de aplicação que se adaptem a este cenário, como, por exemplo, os utilizados no capítulo 5.

Pode-se então concluir que o principal objetivo da dissertação foi cumprido. O sistema criado é capaz de desenvolver, mesmo que ainda limitado por certos aspectos, e simular o funcionamento de uma aplicação. Como referido, a *framework-specific DSL* ainda funciona de forma limitada, dado o espaço de tempo disponível para a elaboração desta dissertação. Por essa razão este mesmo capítulo, para além das contribuições, pretende também mostrar possíveis melhoramentos e desenvolvimentos no trabalho alcançado, informação útil para que seja possível prosseguir com o projeto.

6.1 Contribuições

- Identificação e modelação dos conceitos e abstrações relevantes no domínio de *sensoriamento participado*, focando na dimensão do processamento dos dados
- Extração de abstrações específicas da plataforma *4Sensing*
- *DSL gráfica* específica para a framework *4Sensing* e respectiva integração com a mesma
- Validação da *framework-specific DSL gráfica* através de exemplos de aplicações
- Identificação e, propostas de implementação, de trabalho futuro

6.2 Trabalho futuro

Futuros desenvolvimentos da framework passam por: aperfeiçoar certas características do editor, nomeadamente, definição de melhores interfaces e novas propriedades; a adição de mais elementos (ex. mais sensores); melhoramento da ligação entre editor e simulador possibilitando uma maior troca de dados entre os dois; evolução da DSL numa perspectiva mais geral do domínio de sensoriamento participado; etc. De seguida encontram-se algumas sugestões relativamente a estes desenvolvimentos. Todos eles irão implicar uma prévia aprendizagem nas ferramentas *EMF* e *GMF*, percebendo todo o processo de criação do editor.

6.2.1 Importar tabelas virtuais

Esta seria uma funcionalidade bastante interessante visto que uma tabela virtual pode ser vista como um *subdiagrama*, podendo ser definida num ficheiro à parte. Mais tarde, o utilizador poderia importar essa mesma tabela/ficheiro para a sua aplicação. Esta tabela poderá ser reutilizada várias vezes em mais do que uma aplicação, simplificando a visualização de todo o conjunto de componentes de uma aplicação e acelerando o seu processo de implementação.

6.2.2 Criação de sensores

Uma funcionalidade que deve ser disponibilizada pela framework é a criação de sensores, por parte de cada utilizador, adaptados às necessidades da sua aplicação.

6.2.3 Interface

A interface duma framework é uma componente extremamente importante para proporcionar a sua boa utilização. Características como a representação dos sensores através de imagens são exemplos de melhoramentos visuais que podem tornar a interação com a framework mais intuitiva.

Outros aspectos importantes no futuro, discutidos ao longo do capítulo 4, são:

- adição de *OCL* (secção 4.3.3) ao modelo, colmatando as restrições que não estão a ser validadas por este;
- uma interface que possibilite a inserção de código (secção 4.4) por parte do utilizador;
- automatizar o processo de ligação ao simulador e respetiva execução;
- completar o editor com todas as possíveis parametrizações do simulador *4Sensing* (secção 4.7.1);
- a presença de elementos no editor que expressem os resultados da simulaçao de uma aplicação (secção 4.7.2);
- possibilidade de especificar os argumentos das respetivas operações disponíveis pelo operador *aggregator* (secção 4.8.1);
- possibilidade de especificar quais os atributos de um tuplo que irão fazer de condição num determinado operador (secção 4.8.2);
- seleção e criação de tipos de tuplo através do editor gráfico (secção 4.8.3).

Bibliografia

[1] Google Android. http://www.android.com/, consultado em 2011.

[2] Google AndroidMarket. http://www.android.com/market, consultado em 2011.

[3] Apple AppStore. http://www.apple.com/iphone/apps-for-iphone/, consultado em 2011.

[4] Yves Bontemps, Patrick Heymans, Pierre yves Schobbens, and Jean christophe Trigaux. Semantics of foda feature diagrams. In *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation – Towards Tool Support*, pages 48–58, 2004.

[5] Andrew T. Campbell, Shane B. Eisenman, Nicholas D. Lane, Emiliano Miluzzo, Ronald A. Peterson, Hong Lu, Xiao Zheng, Mirco Musolesi, Kristóf Fodor, and Gahng-Seop Ahn. The rise of people-centric sensing. *IEEE Internet Computing*, 12:12–21, July 2008.

[6] Dana Cuff, Mark Hansen, and Jerry Kang. Urban sensing: out of the woods. *Commun. ACM*, 51:24–33, March 2008.

[7] Krzysztof Czarnecki. Overview of generative software development. In *In Proceedings of Unconventional Programming Paradigms (UPP) 2004, 15-17 September, Mont Saint-Michel, France, Revised Papers*, pages 313–328. Springer-Verlag, 2004.

[8] Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *In MoDELS*, pages 692–706, 2006.

[9] Krzysztof Czarnecki. Framework-specific modeling languages; examples and algorithms. Technical report, ECE, U. of Waterloo, Tech. Rep, 2007.

[10] Nuno Oliveira e Maria João Varanda Pereira e Pedro Rangel Henriques e Daniela da Cruz. Domain specific languages: A theoretical survey. 2009.

[11] Dimitrios S Kolovos e Richard F. Paige e Tim Kelly e Fiona A.C. Polack. Requirements for domain-specific languages. 2006.

[12] Eclipse. http://help.eclipse.org/, consultado em 2011.

[13] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G-S. Ahn, and A. T. Campbell. The bikenet mobile sensing system for cyclist experience mapping. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 87–101, New York, NY, USA, 2007. ACM.

[14] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The Pothole Patrol: Using a Mobile Sensor Network for Road Surface Monitoring. In *The Sixth Annual International conference on Mobile Systems, Applications and Services (MobiSys 2008)*, Breckenridge, U.S.A., June 2008.

[15] Heitor Ferreira, S. Duarte, and N. Preguiça. 4sensing - decentralized processing for participatory sensing data. In *ICPADS2010: Proceedings of the 16th International Conference on Parallel and Distributed Systems*, ICPADS 2010. IEEE Computer Society, 12 2010.

[16] National Instruments Forum. http://forums.ni.com, consultado em 2011.

[17] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, Upper Saddle River, NJ, 2009.

[18] Han Haywood. Multiple inheritance in java. 2003.

[19] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. Cartel: a distributed mobile sensor computing system. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 125–138, New York, NY, USA, 2006. ACM.

[20] National Instruments LabVIEW. http://www.ni.com/labview/, consultado em 2011.

[21] Nicholas D. Lane, Shane B. Eisenman, Mirco Musolesi, Emiliano Miluzzo, and Andrew T. Campbell. Urban sensing systems: opportunistic or participatory? In *Proceedings of the 9th workshop on Mobile computing systems and applications*, HotMobile '08, pages 11–16, New York, NY, USA, 2008. ACM.

[22] Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, and Andrew T. Choudhury, Tanzeem e Campbell. A survey of mobile phone sensing. *Comm. Mag.*, 48:140–150, September 2010.

[23] Benoît Langlois, Consuela elena Jitia, and Eric Jouenne. Dsl classification, 2008.

[24] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–344, December 2005.

[25] Emiliano Miluzzo, Nicholas D. Lane, Kristóf Fodor, Ronald Peterson, Hong Lu, Mirco Musolesi, Shane B. Eisenman, Xiao Zheng, and Andrew T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 337–350, New York, NY, USA, 2008. ACM.

[26] LabVIEW Mindstorms. http://www.ni.com/academic/mindstorms/, consultado em 2011.

[27] Jonathan Sprinkle, Marjan Mernik, Juha-Pekka Tolvanen, and Diomidis Spinellis. What kinds of nails need a domain-specific hammer? *IEEE Software*, 26(4):15–18, July/August 2009. Guest Editors' Introduction: Domain Specific Modelling.

[28] Mark Strembeck and Uwe Zdun. An approach for the systematic development of domain-specific languages. *Softw. Pract. Exper.*, 39:1253–1292, October 2009.

[29] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10:2002, 2001.

[30] Markus Voelter. Best practices for dsls and model-driven development. *Journal of Object Technology*, 2009.

[31] LabVIEW Wiki. http://labviewwiki.org, consultado em 2011.

7

Diagramas

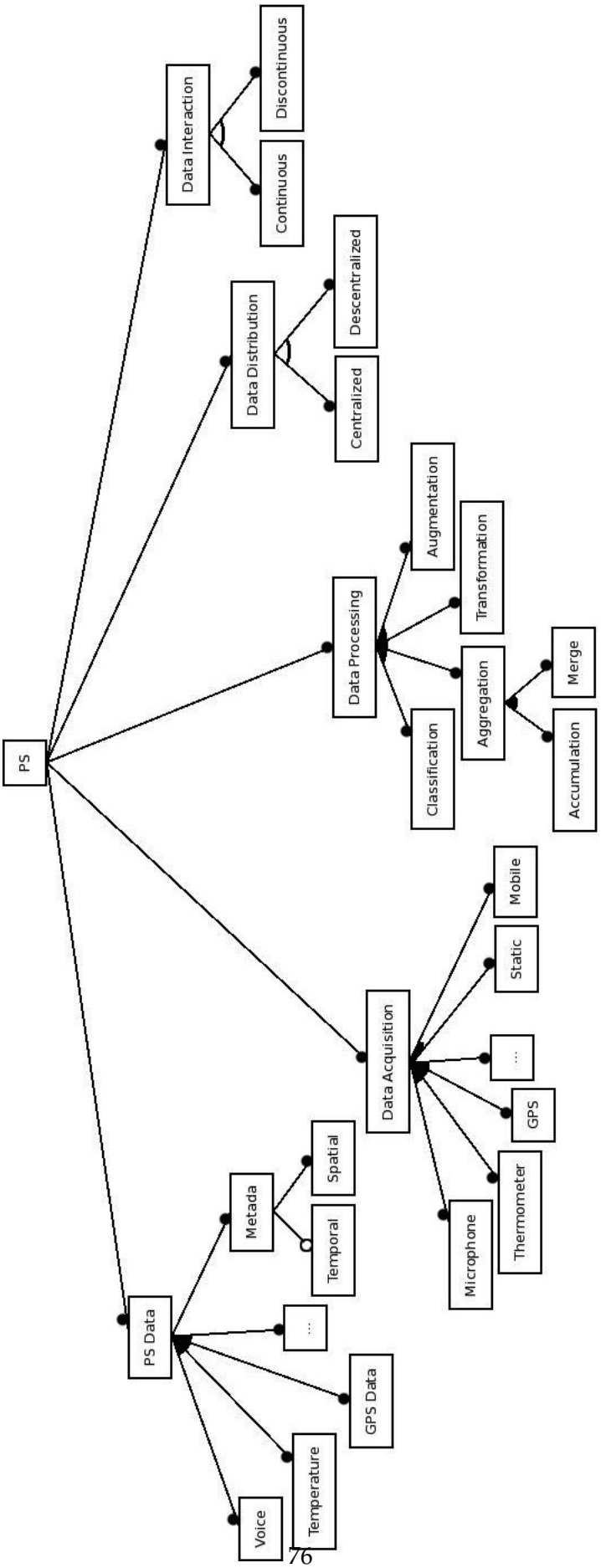


Figura 7.1: *Diagrama de características*, completo, no domínio de *sensoriamento participado*

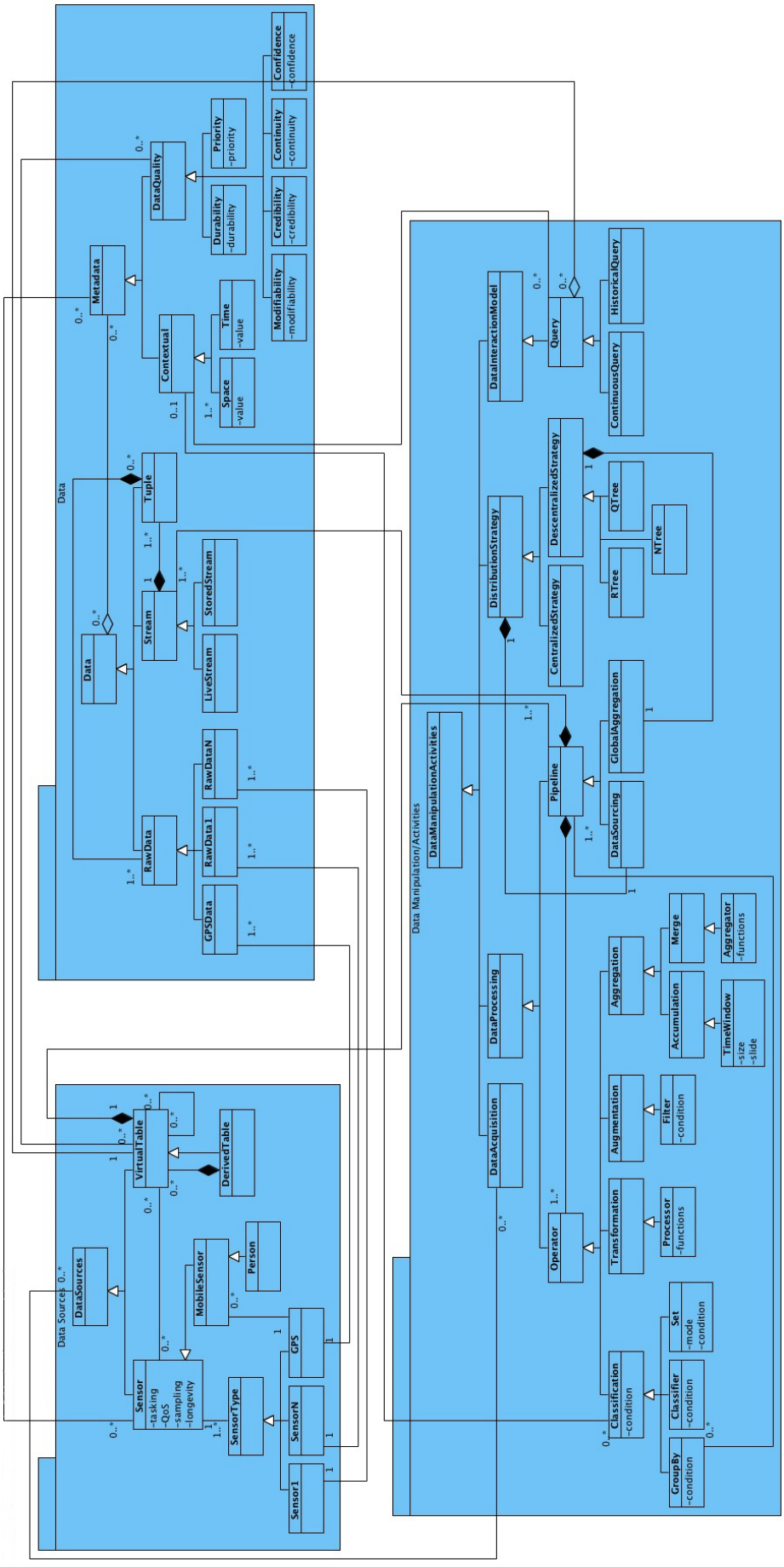


Figura 7.2: Diagrama completo da arquitetura desenvolvida para a DSL

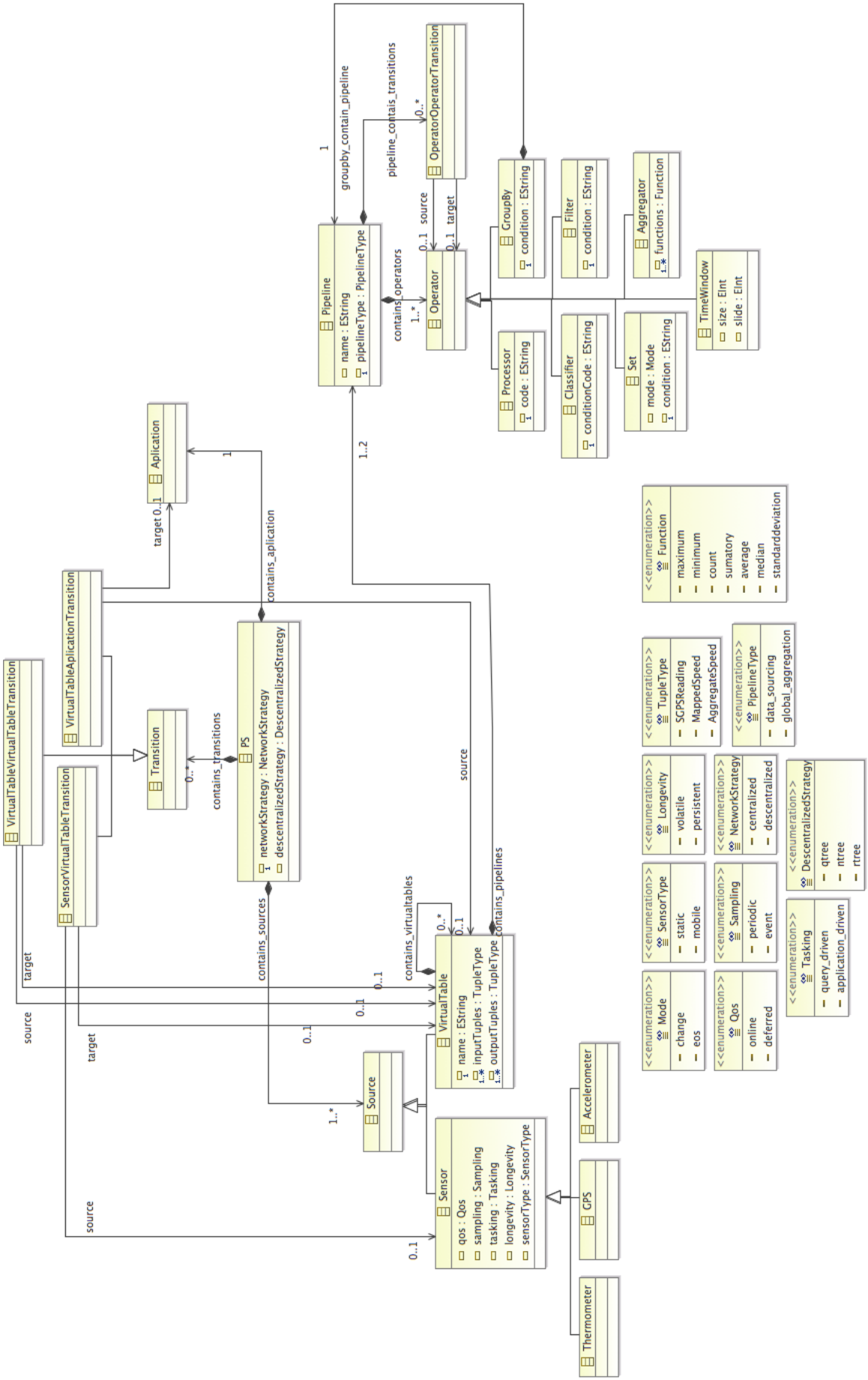


Figura 7.3: Diagrama completo do modelo *.ecore*