UNIVERSIDADE FEDERAL DO AMAZONAS
FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

GERAÇÃO AUTOMÁTICA DE CÓDIGO PARA REDES DE SENSORES SEM FIO BASEADO EM COMPONENTES DE SOFTWARE

ALYSON DE JESUS DOS SANTOS

MANAUS-AM
2011

UNIVERSIDADE FEDERAL DO AMAZONAS
FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ALYSON DE JESUS DOS SANTOS

GERAÇÃO AUTOMÁTICA DE CÓDIGO PARA REDES DE SENSORES SEM FIO BASEADO EM COMPONENTES DE SOFTWARE

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica, área de concentração Controle e Automação de Sistemas.

Orientador:

Prof. Dr. –Ing. Vicente Ferreira de Lucena Júnior

MANAUS
2011

ALYSON DE JESUS DOS SANTOS

GERAÇÃO AUTOMÁTICA DE CÓDIGO PARA REDES DE SENSORES SEM FIO BASEADO EM COMPONENTES DE SOFTWARE

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica, área de concentração Controle e Automação de Sistemas.

Aprovado em 09 de setembro de 2011.

BANCA EXAMINADORA

Prof. Dr. –Ing. Vicente Ferreira de Lucena Júnior
Universidade Federal do Amazonas – UFAM

Prof. Dr. José Pinheiro de Queiroz Neto
Universidade Federal do Amazonas – UFAM

Prof. Dr. José Francisco de Magalhães Netto
Universidade Federal do Amazonas – UFAM

Ficha Catalográfica

(Catalogação realizada pela Biblioteca Central da UFAM)

Santos, Alyson de Jesus dos

S237g Geração automática de código para redes de sensores sem fio baseado em componentes de software / Alyson de Jesus dos Santos - Manaus: UFAM, 2011.

103 f.; il

Dissertação (Mestrado em Engenharia Elétrica) – Universidade Federal do Amazonas, 2011.

Orientador: Prof. Dr. Vicente Ferreira de Lucena Júnior

1. Redes de sensores sem fio 2. Arquitetura Orientada a Modelos 3. Geração Automática de códigos 4. I. Lucena Júnior, Vicente Ferreira de (Orient.) II.Universidade Federal do Amazonas III. Título

CDU 004.4’415(043.3)

Ao meu pai, celestial que com grande misericórdia me alcançou com mão poderosa, e me redimiu com eterna redenção, tornando-me aceito diante Daquele que é tremendo e reina para sempre, amém.

Para Lucélia, José Rumão, Marizete, Lissandro, Lisângelo e Adison, Minha família o meu maior patrimônio celestial, dado por este Deus a quem tenho aprendido amar e servir.

Dedico.

Agradecimentos

Em primeiro lugar quero externar o meu sincero e profundo reconhecimento ao Senhor que me deu vida por meio de seu filho amado, Jesus.

Aos meus pais: José Rumão e Marizete, pessoas de grande valor para mim.

Aos meus sogros: Jenez e Luciene pelo apoio e incentivo.

A Fundação de Apoio Institucional Muraki, na pessoa do Dr. Paulo Alcântara e da Drª Marylane Gurgel pelo apoio financeiro, incentivo e compromisso.

Ao meu orientador professor Vicente Lucena, que não tenho nem palavras para expressar meu carinho, apoio, dedicação, incentivo e gratidão. Agradeço-o também pelas inúmeras orientações no desenvolvimento deste trabalho e no seu exemplo de pesquisador, que tanto nos motiva no desenvolvimento da ciência.

A Universidade Federal do Amazonas, pela oportunidade. Ao Centro Tecnológico de Eletrônica e Informática (CETELI) por suas valiosas contribuições.

A coordenação do Curso de Mestrado em Engenharia Elétrica, na pessoa estimada do Professor João Edgar Chaves Filho.

Aos Professores do Curso de Mestrado em Engenharia Elétrica.

A todos do grupo de pesquisa SEESA, alunos e professores, o meu muito obrigado por suas contribuições na minha formação.

Aos colegas de curso pela preciosa amizade e pelo companheirismo, em especial a Lincoln Ferreira Lima e Pedro Ivan das Graças Palheta, por seu apoio em todas as atividades do mestrado.

Ao amigo e irmão de todas as horas e momentos: Marcus de Lima Braga. Seja nos momentos de alegria, seja nos momentos de tristeza. Sempre compartilhou entusiasticamente de várias idéias e esteve presente diariamente na construção desse trabalho, tornando essa caminhada bem mais suave com sua amizade e companheirismo.

Aos amigos: Nelson Grana, Ricardo Grana, Marco Aurélio Jr, Mário Augusto Takumi Sato, Lívia Cunha, Rodrigo Bernardo, Anderson Luiz Rogério Evangelista e Francisco Mendes.

Aos amigos do grupo de oração pelas suas intercessões e fé.

Seja forte e corajoso, porque você conduzirá esse povo para herdar a terra que prometi sob juramento aos seus antepassados. Somente seja forte e muito corajoso! Tenha o cuidado de obedecer a toda a lei que o meu servo Moisés lhe ordenou; não se desvie dela, nem para a direita nem para a esquerda, para que você seja bem sucedido por onde quer que andar. Não deixe de falar as palavras deste Livro da Lei e de meditar nelas de dia e de noite, para que você cumpra fielmente tudo o que nele está escrito. Só então os seus caminhos prosperarão e você será bem sucedido. Não fui eu que lhe ordenei? Seja forte e corajoso! Não se apavore, nem se desanime, pois o Senhor, o seu Deus, estará com você por onde você andar.

Josué 1. 6-9.

Resumo

As Redes de Sensores Sem Fio (RSSFs) revolucionam a monitoração de ambientes, possibilitando a criação de aplicações e cenários nas mais diversas áreas do conhecimento. Como consequência, temos o crescimento muito rápido do desenvolvimento de novas aplicações e proporcionalmente aumento de complexidade dos sistemas propostos e do custo de desenvolvimento. A dificuldade no desenvolvimento de novas aplicações não se deve somente às características restritivas (memória, processamento, energia), mas também pelas instruções de programação de baixo nível destas redes. Este panorama é favorável a criação de novas metodologias e ferramentas que dêem suporte ao desenvolvimento de sistemas para tal plataforma. Este trabalho propõe o desenvolvimento de uma ferramenta intitulada Geração Automática de Código para Redes de Sensores Sem Fio (GAC-RSSFs), que tem como objetivo principal possibilitar a alta produtividade no desenvolvimento de aplicações para as RSSFs utilizando modelos especificados na Arquitetura Orientada a Modelos (*Model Driven Architecture - MDA*). GAC-RSSFs recebe como entrada um modelo baseado em componentes de uma aplicação criada pelo projetista e então realiza a geração automática de código em linguagem *nesC* (*network embedded systems C*). Três estudos de caso demonstram o potencial de GAC-RSSFs para ajudar os desenvolvedores de aplicações para essas redes.

Palavras-chave: RSSFs, MDA, nesC.

Abstract

The Wireless Sensor Networks (WSNs) revolutionize the monitoring of environment, enabling the creation of applications and scenarios in various areas of knowledge. As a consequence, we have a very fast growth of the development of new applications and rata increase of system complexity and cost of proposed development. The difficulty in developing new applications is not only due to restrictive features (memory, processor, power), but also by the instructions of low-level programming these networks. This situation is conductive to the creation of new methodologies and tools that support the development of system for that plataform. This work proposes the development of a tool called automatic code generation for wireless sensor network (WSN-GAC), which has as its main objective to enable any product high in developing applications for WSNs using models specified in Model driven architecture (MDA). WSN-GAC receives as input a model of a component-based application created by the designer and then performs automatic code generation of nesC (network embedded system C). Three case studied demonstrate the potential of WSN-GAC to help developers of applications for these networks.

Keywords: WSNs, MDA, nesC.

Índice de Figuras

Índice de Tabelas

Lista de Siglas

CIM	*Computation Independent Model*
CUDA	*Compute Unified Device Architecture*
DBC	*Desenvolvimento Baseado em Componentes*
EMF	*Eclipse Model Framework*
GAC-RSSFs	*Geração Automática de Código para Redes de Sensores Sem Fio*
GEF	*Graphical Editing Framework*
GMF	*Graphical Model Framework*
GPGPU	*General-Purpose Computation on GPU*
GPU	*Graphics Processing Unit*
IDE	*Integrated Development Environment*
JET	*Java Emitter Template*
M2M	*Model to Model*
M2T	*Model to Text*
MDA	*Model Driven Architecture*
MDD	*Model Driven Development*
MOF	*Meta-Object Facility*
MVC	*Model-View-Controller*
NESC	*Network Embedded Systems C*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
OTAN	*Organização do Tratado do Atlântico Norte*
PIM	*Plataform Independent Model*
PSM	*Plataform Specific Model*
RSSFs	*Redes de Sensores Sem Fio*
TinyOS	*Tiny Operating System*
UML	*Unified Modeling Language*
UML2	*Unified Modeling Language 2*
VTL	*Velocity Template Language*
XMDL	*X-Machine Description Language*
XMI	*XML Metadata Interchange*

XML	*eXtensible Markup Language*

Sumário

Capítulo 1- Introdução

Avanços recentes em miniaturização de hardware e comunicação sem fio criaram um novo paradigma em monitoração de ambientes (Akyildiz et al., 2002). Dispositivos pequenos e de baixo custo, dotados de unidades de processamento, comunicação e sensoriamento denominados nós sensores podem colaborar através do meio sem fio. Esses dispositivos são capazes de medir características dos ambientes como pressão, temperatura, luminosidade e velocidade (Estrin et al., 1999). Esses dados podem ser processados internamente na rede e depois enviados a outros nós sensores ou a um ponto de acesso denominado estação base (*Base Station*). Os nós sensores podem ser utilizados em várias aplicações, como detecção de incêndios, monitoração ambiental, monitoração e controle industrial, agricultura de precisão e rastreamento de alvos. Uma coleção de nós sensores trabalhando em uma aplicação compõem uma Redes de Sensores Sem Fio (RSSFs).

Programar os nós de uma RSSF é uma tarefa difícil e trabalhosa, e conta com duas categorias de desenvolvedores, que precisam criar as mais diversas aplicações, a custo de muito tempo e esforço de codificação do projeto. Temos na primeira categoria, os desenvolvedores especialistas, conhecedores em nível de detalhes do modelo de programação das RSSFs e da especificação da linguagem em particular. Na segunda categoria, os desenvolvedores inexperientes que precisam construir aplicações, porém tem pouco conhecimento tanto do modelo de programação como da especificação da linguagem. Todas as categorias apresentadas são importantes e devem ser consideradas, principalmente quando se trata de acelerar e facilitar o processo de desenvolvimento de software. Reduzir o tempo e o esforço dos desenvolvedores na codificação do projeto, sem diminuir a qualidade do produto final (software), vem sendo uma busca incessante dos engenheiros de software, através dos elementos fundamentais da engenharia de software que são: os métodos (como fazer), as ferramentas (semi-automação e automação dos métodos) e os procedimentos (união entre os métodos e as ferramentas). A partir destes princípios muitas técnicas, modelos, metodologias e ferramentas foram e estão sendo desenvolvidas para aumentar a produtividade dos desenvolvedores na construção de aplicações para sistemas embarcados.

Uma das iniciativas proposta é a utilização da *Model Driven Architecture* (MDA) criada pela *Object Management Group* (OMG) (OMG, 2009). O desenvolvimento orientado por modelos é foco de muitos estudos nos últimos anos. O interesse cresce à medida que se torna uma proposta viável no processo de desenvolvimento de software e na missão de prover soluções para o problema da produtividade do desenvolvedor (Mellor et al., 2003). O uso de representações gráficas precisas, porém abstratas de algoritmos, de tal forma que permita a construção de sistemas completos a partir de modelos que podem ser entendidos muito mais rapidamente e profundamente do que os códigos de linguagem de programação é a essência da MDA.

1.1 Motivação

A crescente complexidade de sistemas de software modernos cria a necessidade do surgimento de novas tecnologias e de novos paradigmas de desenvolvimento. Alguns dos requisitos presentes nesse tipo de sistema podem ser: a comunicação e a troca de dados de diferentes tipos, a capacidade de adaptar-se dinamicamente perante as mudanças de requisitos e a capacidade de adaptar-se ao ambiente de operação. Sendo assim, projetar, analisar e implementar sistemas modernos utilizando os paradigmas de desenvolvimento centrados em código requer um maior esforço, custo elevado, e um nível de dificuldade significativo. A busca por melhorias no processo de desenvolvimento, com o objetivo de reduzir os custos, aumentar a produtividade dos desenvolvedores e contornar os problemas do desenvolvimento centrado em código é algo que os engenheiros de software almejam conseguir.

Algumas das principais melhorias estão relacionadas ao aumento do nível de abstração necessário para projetar e implementar o *software*. Inicialmente, toda a codificação do *software* era realizada em linguagens muito próximas das linguagens de máquina, com pouca expressividade e de difícil manutenção. Com o aumento da complexidade dos sistemas desenvolvidos, essas linguagens foram se tornando insuficientes, resultando no surgimento de novas linguagens e paradigmas de programação com maior nível de abstração (Mellor et al., 2004).

Desde então, as linguagens de modelagem vem ganhando expressividade e tornando possível a construção de modelos de projeto do *software* em desenvolvimento. Esses modelos permitiram a elaboração de representações dos requisitos do *software*, que

podem ser visualizadas e manipuladas em tempo de análise e projeto do sistema, possibilitando uma melhor avaliação destes requisitos antes mesmo do código ser implementado.

Os modelos estavam voltados para artefatos de apoio ao processo de desenvolvimento, ajudando na comunicação dos desenvolvedores ou constituindo a documentação do *software*. No entanto, surge a necessidade de mudar este paradigma no sentido de utilizar os modelos como artefatos centrais no desenvolvimento (France e Rumpe, 2007), a partir do qual o código é gerado. Este novo paradigma, conhecido como MDA, permite que os desenvolvedores concentrem seus esforços na elaboração de modelos que representam os conceitos e funcionalidades desejáveis no *software* projetado. Estes modelos são transformados e geram artefatos de implementação automaticamente, de forma a refletir a solução expressa nos modelos (Schmidt, 2006).

Dessa forma, automatizar o desenvolvimento de software para as RSSFs pode ser facilitado com o uso da MDA, viabilizando o desenvolvimento de ferramentas que automatizem a geração de código nesC a partir de modelos conceituais UML.

1.2 Problema

As RSSFs têm atraído a atenção de muitos pesquisadores devido seu caráter pervasivo e flexível utilizado para o monitoramento e controle de fenômenos físicos. Em conseqüência disso, temos o crescimento muito rápido do desenvolvimento de novas aplicações e proporcionalmente aumento de complexidade dos sistemas propostos e do custo de seu desenvolvimento. A dificuldade no desenvolvimento de novas aplicações não se deve somente as suas características restritivas (memória, processamento, energia), mas também por sua programação de baixo nível e na necessidade de novas metodologias, ferramentas e processos aplicados as RSSFs. Desta forma, como modelar as aplicações para as RSSFs, utilizando novas metodologias, ferramentas e processos, de modo a gerar código executável nos nós sensores é o problema a ser abordado nesta dissertação.

1.3 Objetivo Geral

Pesquisar, projetar, desenvolver, implementar e testar uma ferramenta de modelagem de componentes de software e geração de código correspondente na linguagem nesC, de acordo com a metodologia MDA.

1.4 Objetivos Específicos

- Pesquisar as ferramentas de geração de código existentes na literatura, identificando princípios, regras, etapas, atividades, artefatos gerados, analisando os pontos que possam convergir e divergir;
- Elaborar um perfil UML (metamodelo) para as RSSFs que disponibilize classes básicas e estereótipos que representam o desenvolvimento baseado em componentes (componente e interface);
- Aplicar o método MDA na construção da ferramenta, de forma que implemente os modelos e as transformações de modelos descritos, respectivamente, nas seções 2.4.1 e 2.4.2;
- Construir a interface gráfica da ferramenta de software para o projetista representar os componentes e as interfaces das aplicações;
- Gerar código em linguagem nesC como saída, a partir de aplicações criadas pelo projetista baseado nos estudos de caso propostos.

1.5 Metodologia

A metodologia adotada para a realização do presente trabalho, conforme visto na Figura 1.1, está dividida em quatro grandes etapas: revisão bibliográfica, projeto conceitual e avaliação da proposta, implementação e análise dos resultados. A etapa de revisão bibliográfica produz o conhecimento necessário para a concepção da ferramenta. O projeto conceitual define os requisitos que serão implementados e a proposta de solução. A etapa de implementação gera os resultados (dentre eles, a ferramenta proposta por esta dissertação) e realiza os estudos de caso propostos para validar o código fonte em linguagem nesC, seguindo para a etapa de análise. Essa última etapa, por sua vez, produz as conclusões e as considerações finais do trabalho. Em cima dessas quatro etapas, foram

elaborados os capítulos da dissertação, com exceção do presente capítulo, que traz: a introdução da dissertação; a motivação e a justificativa para a realização do trabalho; objetivos gerais e específicos; estrutura do trabalho.

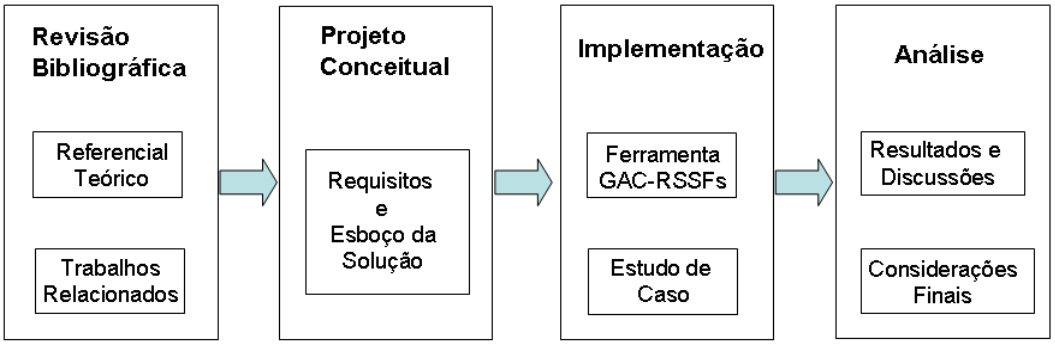

Figura 1.1. Metodologia do trabalho.
Fonte: Adaptado de Ferreira (2009)

1.6 Justificativa

A disponibilização de ferramentas que facilitem o desenvolvimento de aplicações para as RSSFs é um dos grandes desafios que se apresenta aos projetistas destas redes. Dentro das opções de ferramentas de conversão de modelos UML para código fonte em linguagem nesC, na realidade em que a dissertação foi elaborada, não há registro de *plugin* Eclipse que viabilize esse processo. No segmento comercial foi encontrada ferramenta proprietária LabView WSN NI, abordada na seção 3.4, que por meio de diagrama de blocos gera código em linguagem nesC, entretanto não utiliza os modelos descritos na MDA.

As aplicações voltadas para as RSSFs implementam rotinas de *software* diretamente sobre o processador, sem a preocupação com a independência da plataforma de *hardware* e capacidade de extensão. Isto dificulta o processo de desenvolvimento deste software, exigindo profundo conhecimento do modelo de programação. Com o intuito de elevar o nível de abstração de programação destas redes (Bakshi, 2008), um conjunto de bibliotecas de componentes fornece aos desenvolvedores modelos que representam os conceitos e funcionalidades desejáveis no *software* projetado.

Diante do exposto, a proposta da ferramenta GAC-RSSFs é oferecer a estrutura básica desse tipo de aplicação, através de um modelo de componentes que favorece o acréscimo, a substituição e a reutilização dos seus componentes, dando suporte para o

sistema operacional embarcado TinyOS, de forma a reduzir custos e tornar mais produtivo o desenvolvimento.

1.7 Organização do Trabalho

No primeiro capítulo deste trabalho descreve-se uma breve introdução sobre as RSSFs, bem como programar os nós de uma RSSFs, com as devidas considerações e embasamentos.

O capítulo 2 contém definições essenciais para compreender o universo das RSSFs e MDA, possibilitando adquirir o conhecimento necessário para representar os elementos de domínio das RSSFs sob a ótica de componentes de software e a construção dos modelos da MDA, bem como suas transformações e padrões usados.

O capítulo 3 apresenta os principais trabalhos relacionados, bem como uma análise detalhada das características de cada ferramenta de geração de código. Uma tabela comparativa dos recursos oferecidos pelas ferramentas foi gerado, definindo os critérios que são fundamentais para a construção de GAC-RSSFs.

O capítulo 4 apresenta a solução conceitual de GAC-RSSFs. O capítulo está dividido em três partes: i) a definição dos requisitos do *plugin*; ii) o esboço da solução conceitual; e iii) a descrição da arquitetura da plataforma Eclipse.

O capítulo 5 descreve o procedimento de implementação do *plugin* proposto, doravante denominado GAC-RSSFs. O capítulo está dividido em três partes: i) a definição das características do plugin; ii) a descrição da arquitetura da ferramenta; iii); e a implementação da ferramenta.

O capítulo 6 apresenta três estudos de caso que ilustram a construção de aplicações a partir da ferramenta GAC-RSSFs. Duas aplicações estão presentes na árvore de aplicações do TinyOS, são elas: Blink e Surge. A outra aplicação, denominada Posição, determina a posição do nó sensor, não consta na distribuição do TinyOS e foi desenvolvida pelo próprio autor do trabalho. Neste capítulo são descritos: i) os componentes e as interfaces das aplicações exemplo; ii) a representação das aplicações na interface gráfica da ferramenta; iii) o modelo armazenado em formato *eXtensible Markup Language* (XML) gerado a partir da interface gráfica; e iv) os arquivos gerados pela ferramenta GAC-RSSFs.

O Capítulo 7 é dedicado aos resultados da dissertação e às discussões. São analisados os resultados obtidos pelas aplicações dos estudos de caso proposto no capítulo anterior.

O Capítulo 8 apresenta as considerações finais e as conclusões da dissertação, além de apresentar propostas e recomendações para trabalhos futuros.

Capítulo 2- Fundamentos Teóricos

Esta dissertação envolveu duas abrangentes linhas de pesquisa: Redes de Sensores Sem Fio (RSSFs) e a MDA. Inicialmente são apresentados os elementos básicos de uma Rede de Sensores Sem Fio. Em seguida, são apresentados o Sistema Operacional TinyOS e a linguagem nesC, utilizada para implementação de aplicações. A definição de componentes e as características são estudadas, devido o Sistema Operacional TinyOS ser formado por uma arquitetura baseada em componentes. Por fim, é apresentada a Arquitetura Orientada a Modelos, conhecida como MDA, são discutidos os conceitos, as transformações sobre os modelos, os padrões e as linguagens oferecidas pela OMG que dão suporte a esta arquitetura.

2.1 Redes de Sensores Sem Fio

As RSSFs têm sido viabilizadas pela rápida convergência de três tecnologias: microprocessadores, comunicação sem fio e microsistemas eletromecânicos (Ilyas e Mahgoub, 2004; Loureiro et al., 2003). Uma RSSF pode ser utilizada para monitorar e, eventualmente, controlar um ambiente. Este tipo de rede é formado geralmente por centenas ou milhares de dispositivos autônomos que tendem a ser projetados com pequenas dimensões chamados nós sensores (Akyildiz et al., 2002).

Os nós individualmente possuem pouca capacidade computacional e severas restrições ao consumo de energia, mas um esforço colaborativo entre os nós permite a realização de uma grande tarefa (Ruiz et al., 2004). Os nós sensores podem ser lançados sobre áreas remotas (reservas ambientais, oceanos, vulcões, rios, florestas, etc.) e, sem intervenção de técnicos ou operadores, formar uma rede sem fio *ad hoc* que coleta dados sobre os fenômenos de interesse, realiza processamento local, e dissemina as informações para um ponto de acesso em um esquema de comunicação de múltiplos saltos (*multihop*). O ponto de acesso é o elemento através do qual a rede comunica-se com outras redes ou com um ou mais observadores (Ruiz, 2003). O ponto de acesso pode ser implementado em um nó sensor que será chamado de nó sorvedouro (*sink node)* ou em uma estação base.

A Figura 2.1 mostra como os sensores geralmente estão espalhados em uma área que está sendo monitorada. Cada sensor tem a capacidade de coletar informações e roteá-las ao nó *sink* que é o responsável por se comunicar com a aplicação para gerenciamento de sensores através da Internet.

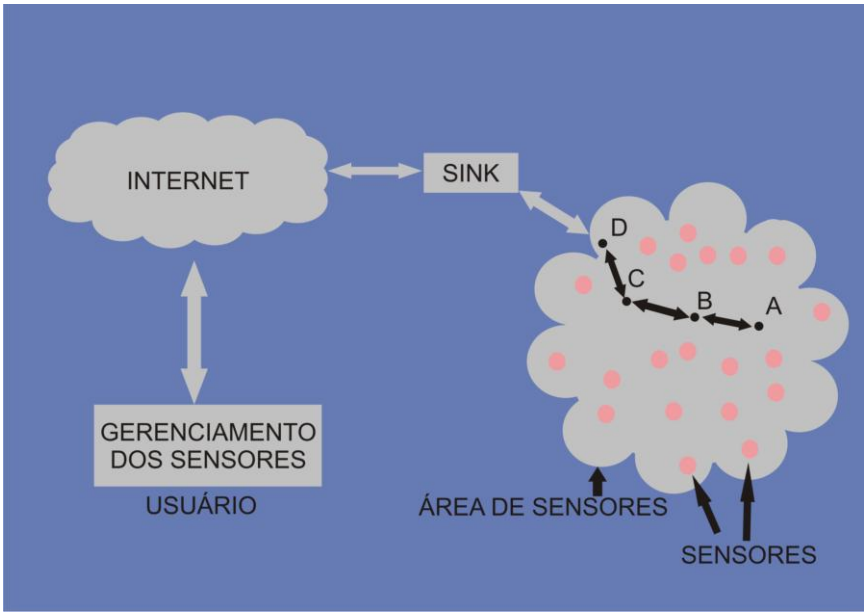

Figura 2.1. Elementos de uma RSSFs.
Fonte: Akyildiz et al. (2002).

Os elementos de hardware e software das RSSFs serão abordados nas próximas subseções.

2.1.1 Componentes de Hardware do Nó Sensor

Um nó sensor é um elemento computacional com unidade de energia, unidade de processamento, sensores e comunicação. Na Figura 2.2 são ilustrados os componentes típicos de hardware do nó sensor (Sohraby, 2007).

a) Unidade de energia - uma infra-estrutura de energia apropriada deve fornecer condição do sistema operar por algumas horas, meses ou anos, dependendo da aplicação.

b) Unidade de processamento - é usada para processar e manipular os dados, pelo armazenamento de longo e curto prazo, criptografia, correção de erros, modulação digital e transmissão digital.

c) Sensores - os principais fenômenos a serem observados com o auxilio de sensores são: aceleração, umidade, luz, fluxo magnético, temperatura, pressão e som.

d) Comunicação - os nós sensores devem ser capazes de se comunicar através de um sistema baseado em malhas com conectividade de rádio entre múltiplos nós sensores, utilizando roteamento dinâmico.

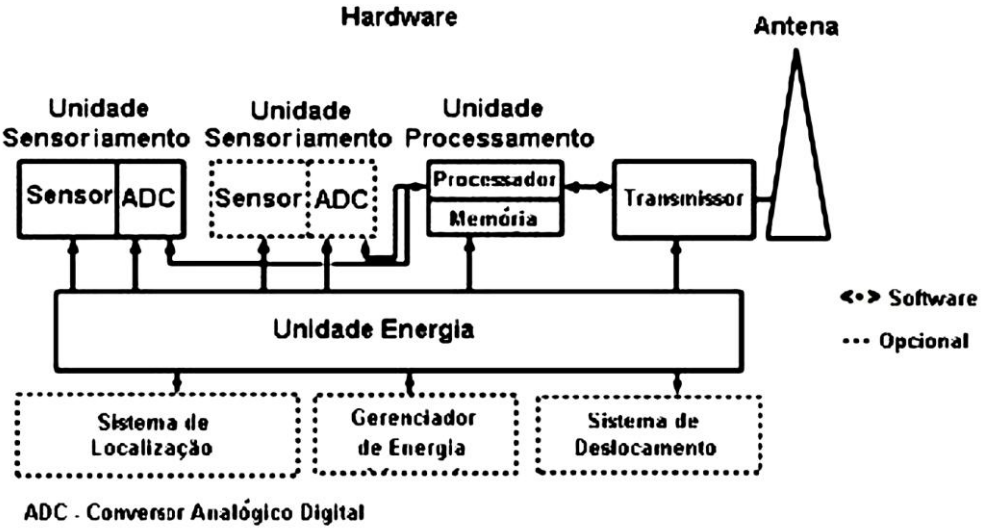

Figura 2.2. Componentes de Hardware de um Nó sensor.
Fonte: Sohraby (2007).

Os nós sensores devem usar de modo eficiente o processador e a memória, ao mesmo tempo em que devem garantir baixo consumo de energia. O ideal é que esses dispositivos sejam mantidos o máximo de tempo possível em modos de operação que minimizem o consumo de energia, e que sejam solicitados apenas quando for necessário tratar algum evento da rede.

2.1.2 Componentes de Software do Nó Sensor

O componente lógico de um nó sensor é o software que executa no processador (Loureiro et al., 2003). Os componentes típicos de software do nó sensor são mostrados na Figura 2.3 e descritos abaixo.

a) Sistema Operacional: o código deve ser pequeno e garantir funcionalidades básicas do sistema para que softwares direcionados a aplicações possam ser executados sobre a arquitetura do microcontrolador.

b) *Driver* de sensores: *software* que gerencia as funções básicas dos sensores.

c) Processo de comunicação: gerencia as funções de comunicação como rotas, *buffer* de pacotes, encaminhamento de pacotes, manutenção de topologia, controle de acesso ao meio e criptografia.

d) *Driver* de Comunicação: gerencia as funções básicas do canal de transmissão do rádio como sincronização, codificação de sinal, bit de recuperação, bit contador, nível do sinal e modulação.

e) Processamento de dados: são processamentos numéricos, de dados, valor e manipulação de sinal e outros processamentos básicos para aplicações.

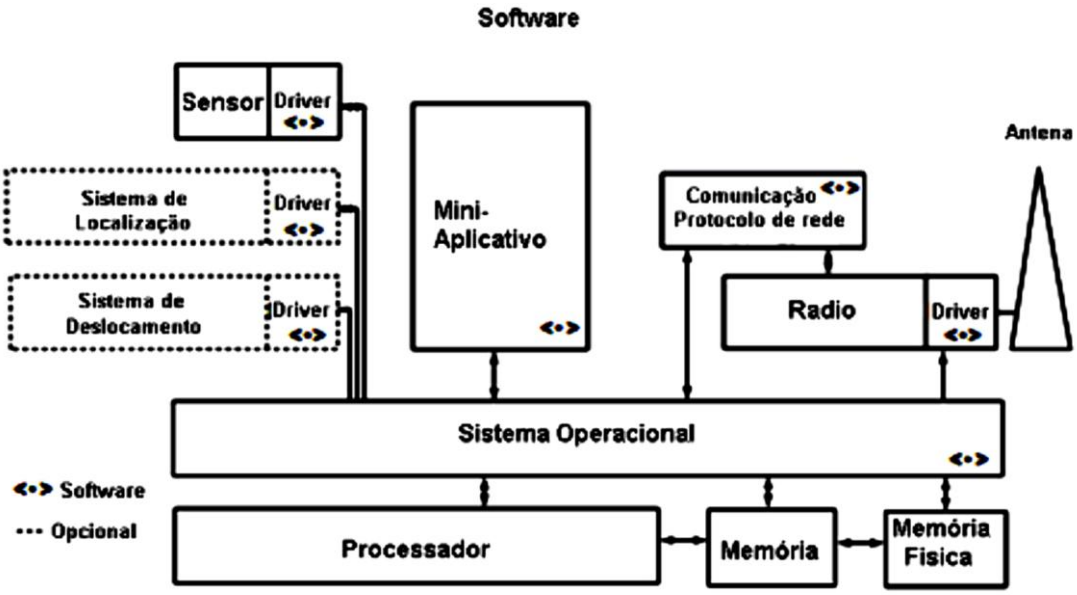

Figura 2.3. Componentes de Software de um Nó sensor.
Fonte: Sohraby (2007).

As RSSFs são totalmente dependentes das aplicações. Em qualquer projeto ou solução proposta para estas redes é necessário levar em consideração os requisitos da aplicação a ser desenvolvida, as características e restrições dos componentes dos nós sensores, assim como as características do ambiente onde tais redes serão aplicadas.

As aplicações desenvolvidas a partir do sistema operacional TinyOS são compostas por um conjunto de componentes agrupados. A seção 2.2 apresenta a definição de componentes, os conceitos relacionados, as características e os princípios.

2.2 Componentes

A idéia de componentes de software não é nova. Em 1968, durante uma conferência da OTAN sobre Engenharia de Software, McIlroy (McIlroy, 1968) argumentou que deveria haver empenho em produzir componentes reutilizáveis com o intuito de facilitar a tarefa dos desenvolvedores de software. O artigo apresentado na conferência, intitulado

"Componentes de Software Produzidos em Massa", por McIlroy (McIlroy, 1968), se tornou um dos mais revolucionários da área de reutilização. Nesse contexto, os componentes seriam rotinas que ficariam disponíveis para os programadores utilizarem nos seus softwares.

Um outro conceito de componente é apresentado por Szyperski (Szyperski, 1999): "Um componente é definido como uma unidade de software independente, que encapsula dentro de si seu projeto e implementação, e oferece interfaces bem definidas para o meio externo". A motivação para componentes não é unicamente relacionada a reutilização. As recentes pressões para a liberação de produtos no mercado (*time-to-market*), assim como a necessidade de lidar com modificações de maneira rápida e efetiva, têm contribuído para a relevância de componentes na produção de software.

2.2.1 Princípios de Componentes

Os Componentes (Gimenes e Huzita, 2005; Lucena Jr, 2002) têm pontos de interconexão chamados de interfaces que concentram um conjunto de serviços relacionados. As interfaces podem ser classificadas em dois tipos: interfaces fornecidas (*provides interfaces*) e interfaces requeridas (*uses interfaces*). A primeira define os serviços oferecidos pelo componente. A segunda, define os serviços que o componente necessita de outros componentes. Componentes se conectam por meio da interface requerida por um com a interface fornecida de outro.

As características de componentes sugerem que qualquer método para especificação e implementação de componentes deve incluir os seguintes requisitos:

a) Um componente deve fornecer uma especificação clara dos seus serviços. As interfaces fornecidas de um componente devem ser identificadas e definidas separadamente. Cada interface consiste em serviços especificados, mediante uma ou mais operações, sendo cada uma delas separadamente identificada e especificada de acordo com seus parâmetros de entrada e saída e respectivos tipos estabelecidos. Essas definições constituem a assinatura (*signature*) da interface;

b) As interfaces requeridas também devem ser definidas explicitamente. Essas interfaces definem os serviços necessários de outros componentes, para que um componente possa completar o seu próprio serviço;

c) A única forma de interação de um componente com outros é através de suas interfaces. Um componente deve garantir o encapsulamento de seus dados e processos;

d) Componentes que utilizam um outro componente devem fazê-lo com base apenas nas interfaces definidas e serviços especificados, não sendo feita suposição alguma sobre a sua implementação.

2.2.2 Desenvolvimento Baseado em Componentes

O Desenvolvimento Baseado em Componentes (DBC) surgiu como uma nova perspectiva para o desenvolvimento de software, cujo objetivo é a quebra dos blocos monolíticos em componentes interoperáveis, reduzindo tanto a complexidade no desenvolvimento quanto os custos, por meio da reutilização de componentes (Sametinger, 1997; D'Souza e Wills, 1999). O software passa a ser composto de partes relativamente independentes, que foram concebidas para serem substituíveis, reutilizáveis e interoperáveis.

O DBC pode considerar o desenvolvimento de componentes ou o desenvolvimento com componentes. A primeira perspectiva engloba as atividades envolvidas na concepção e implementação de um componente, devendo existir a preocupação em gerar a documentação necessária para que o componente possa ser posteriormente reutilizado. A segunda perspectiva considera a existência de componentes e relaciona as atividades necessárias para o desenvolvimento de software pela composição dos componentes.

Nesta dissertação, o desenvolvedor de aplicações utilizará somente as práticas relacionadas para o desenvolvimento com componentes, descrito na subseção 2.2.3.

2.2.3 Desenvolvimento com Componentes

Segundo Brown e Short (Brown e Short, 1997), as atividades essenciais para esse desenvolvimento são: seleção, qualificação, adaptação, composição e atualização.

1) Seleção: seleciona os componentes com o potencial para serem usados na construção da aplicação.

2) Qualificação: examina os componentes reutilizáveis para averiguar se, e em que proporção, se adequam aos requisitos do estilo arquitetural da aplicação.

3) Adaptação: representa o processo de alteração comportamental e/ou estrutural de um componente para se adequar aos requisitos de uma aplicação em particular.

4) Composição: é a atividade de integração de diferentes componentes para o desenvolvimento de uma aplicação.

5) Atualização: realiza-se a atualização dos componentes, onde versões antigas serão substituídas ou novos componentes, com comportamento e interface similares, serão incluídos.

2.3 O Sistema Operacional TinyOS

O *Tiny Operating System* (TinyOS) (Hill, 2000; TinyOS, 2008) foi desenvolvido inicialmente na Universidade da Califórnia, em Berkeley, com plataforma de software de código aberto e arquitetura de sistema baseada em componentes. Ele é utilizado por uma grande comunidade de usuários (Ruiz, 2004).

As aplicações são escritas em um dialeto da linguagem C, denominado nesC (Culler et al., 2003), que utiliza os conceitos estruturais do TinyOS no seu modelo de execução. Os conceitos básicos deste modelo são:

a) Separação entre construção e composição: aplicativos são formados por componentes, os quais são combinados para criar aplicativos mais complexos.

b) Especificação de funcionalidade através de interfaces: interfaces podem ser fornecidas ou usadas pelos componentes. As interfaces fornecidas representam a funcionalidade que o componente provê ao aplicativo; as interfaces usadas representam a funcionalidade necessária ao componente para executar seu trabalho.

c) As Interfaces são bidirecionais: interfaces especificam um conjunto de funções que serão implementadas pelo componente provedor da interface (comando) e outro conjunto que será implementado pelo componente usuário da interface (evento). O comando (*Commands*) é a requisição ao componente para execução de algum serviço; o evento (*Events)* é a sinalização do componente indicando o fim da execução de um serviço; as tarefas (*Tasks)* são atômicas entre si; elas executam até o seu término, mas podem ser interrompidas por eventos externos. Na Figura 2.4 é ilustrada a abstração dos comandos, eventos e tarefas.

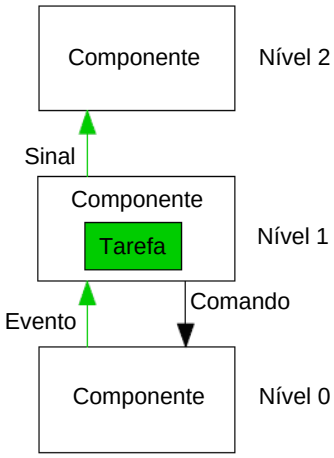

Figura 2.4. Fluxo de execução dos comandos, eventos e tarefas.
Fonte: Hill (2000).

Os comandos requisitam execuções de mais baixo nível (nível 1 para o nível 0) e não são bloqueáveis. Cada componente de baixo nível possui controladores que correspondem aos pedidos vindos da camada acima. Por outro lado, os eventos (nível 0 para o nível 1) são invocados de modo a lidarem com os eventos de hardware de forma direta ou indireta. Os componentes de baixo nível são responsáveis por lidar com as interrupções de hardware, podendo executar um pequeno processamento e gerar outros eventos. Assim, a execução de um componente baseia-se no envio para baixo nível dos comandos e a geração de eventos para as camadas acima. As tarefas podem chamar comandos de nível inferior (nível 1 para o nível 0), sinalizar eventos de um nível superior (nível 1 para o nível 2) e programar outras tarefas dentro do próprio componente.

Uma configuração completa do sistema consiste de um programa composto de uma aplicação e dos componentes do TinyOS (Ruiz et al., 2004). Uma aplicação é um grafo de componentes agrupados hierarquicamente. Na Figura 2.5 apresenta-se um exemplo de grafo de componentes de uma aplicação do TinyOS. A aplicação *SenseToRfm* coleta, periodicamente, sinais de luz e envia o valor em broadcast para outros sensores. Os nós do grafo são os componentes e as arestas são interfaces. As setas para cima indicam o fluxo de eventos e as setas para baixo indicam o fluxo de comandos.

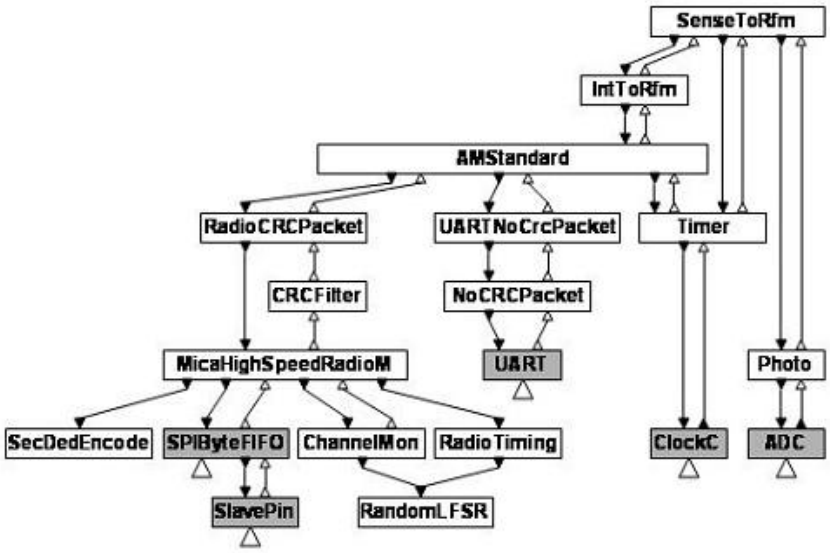

Figura 2.5. Grafo simplificado dos componentes da aplicação *SenseToRfm* do TinyOS.
Fonte: Hill (2000).

2.3.1 A Linguagem nesC

A linguagem nesC (Culler et. al, 2003) é bastante similar a C e sua maior diferença é o modelo de ligação entre os componentes. O uso de componentes diminui o tempo de desenvolvimento de aplicações e permite a sua reusabilidade.

Os programas escritos em nesC são organizados em três tipos de arquivos: interface, módulo e configuração.

Interface: são pontos de acesso aos componentes e devem obedecer a um padrão e conter apenas as assinaturas dos comandos e eventos. A Figura 2.6 mostra um exemplo de interface, neste caso a interface Clock.

```
interface Clock {
command result_t setRate(char interval,
                         char scale);
event result_t fire();
}
```

Figura 2.6. Interface Clock.

Módulo (*module*): contêm o código da aplicação implementando uma ou mais interfaces. A Figura 2.7 ilustra o exemplo do módulo da aplicação Blink.

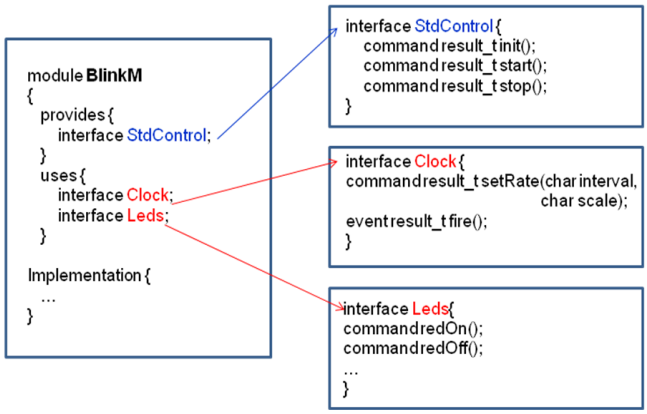

Figura 2.7. Módulo da aplicação Blink.

Configuração (*configuration*): são componentes cuja função é ligar os componentes uns aos outros de acordo com as interfaces fornecidas e usadas por esses componentes. Dois componentes podem apenas interagir entre si por meio de uma interface. Na Figura 2.8 é ilustrado a configuração da aplicação *Blink*, fornecida na instalação padrão do TinyOS.

```
configuration Blink{
}

implementation{
    components Main, BlinkM, ClockC, LedsC;

    Main.StdControl-> BlinkM.StdControl;
    BlinkM.Clock-> ClockC;
    BlinkM.Leds-> LedsC;
}
```

Figura 2.8. Configuração da aplicação Blink.

2.3.2 TOSSIM/TiniViz

O TOSSIM (*TinyOS Simulator)* é a ferramenta de simulação para as aplicações desenvolvidas sobre o TinyOS. Para simular uma aplicação TinyOS no TOSSIM, a aplicação deve ser compilada para plataforma PC, resultando na geração de um arquivo executável.

O simulador TOSSIM inclui uma biblioteca de componentes de hardware genéricos pré-definidos (processador, rádio, sensor, bateria) que são emulados durante a simulação. Desta forma, a simulação pode capturar o comportamento do sistema completo com alto grau de fidelidade (Levis e Lee, 2003).

O TinyViz é uma interface gráfica implementada em Java para o TOSSIM que oferece ao usuário mecanismos de interação, visualização e controle da simulação. Essa interação do usuário com a simulação se dá através de componentes gráficos (*plugins*), os quais podem ser da biblioteca original do TinyViz ou desenvolvidos especialmente para uma determinada aplicação. A interação entre o TOSSIM e o TinyViz é apresentada na Figura 2.9, sendo descrito no próximo parágrafo.

Os componentes que constituem o TinyViz estão dispostos no interior do retângulo mais externo. O módulo de comunicação (*Communication*) é responsável por receber eventos externos por meio da interface serial. Tais eventos podem ser provenientes do simulador TOSSIM ou de um dispositivo real conectado ao computador. Uma vez recebidos, os eventos são colocados no barramento de eventos (*Event Bus*), de onde podem ser processados por *plugins* ou visualizados diretamente na interface gráfica (GUI). De forma semelhante, ao utilizar os *plugins*, o usuário pode enviar mensagens à simulação, as quais são colocadas no *EventBus* para, posteriormente, serem transmitidas pelo módulo de comunicação (Ruiz et al., 2004).

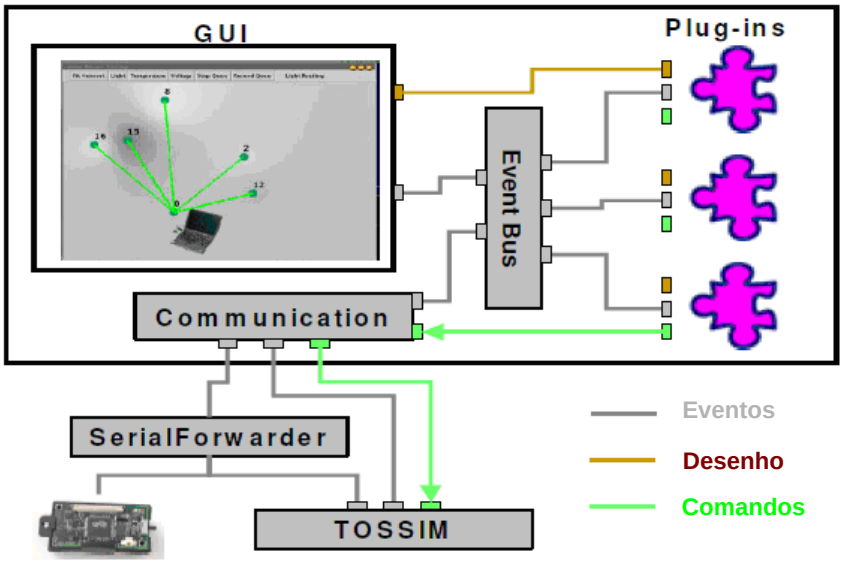

Figura 2.9. Esquema de interação entre TOSSIM E TinyViz.
Fonte: Ruiz (2004).

2.4 Desenvolvimento Orientado a Modelos

O desenvolvimento de software orientado a modelos (*Model Driven Development* - MDD) tem como idéia principal reconhecer a importância dos modelos no processo de

software, não apenas como um guia para tarefas de desenvolvimento, mas como parte integrante do software.

A proposta do MDD é fazer com que o engenheiro de software não precise interagir manualmente com todo o código fonte, podendo se concentrar em modelos de mais alto nível, ficando protegido das complexidades requeridas para implementação nas diferentes plataformas. Um mecanismo automático é responsável por gerar automaticamente o código a partir dos modelos. Neste cenário, os modelos não são apenas um guia, ou uma referência. Eles fazem parte do software, assim como o código fonte (Lucrédio, 2009).

2.4.1 Arquitetura Orientada a Modelos

A Arquitetura Orientada a Modelos, conhecida como *Model Driven Architecture* - MDA (Mukerji e Miller, 2003) surgiu ao longo do ano 2001, de uma iniciativa da *Object Management Group* (OMG), uma organização internacional que aprova padrões abertos para aplicações orientadas a objeto. A MDA tem o lema *"Design once, build it on any platform"* (Modele uma vez, codifique para qualquer plataforma), inspirado no lema da linguagem Java (JavaSoft, 2011) *"Write once, run anywhere"* (Codifique uma vez, rode em qualquer plataforma).

Um modelo é uma representação simplificada de algum conceito, com o objetivo de observação, manipulação e entendimento sobre tal conceito (Mellor *et al.*, 2004). No desenvolvimento de software, modelos são criados com o objetivo de diminuir a complexidade inerente ao desenvolvimento.

A MDA introduz os conceitos de *Computation Independent Model* (CIM) ou Modelo Independente de Computação, *Platform Independent Model* (PIM) ou Modelo Independente de Plataforma e *Platform Specific Model* (PSM) ou Modelo Específico de Plataforma.

Um CIM é uma visão do sistema de um ponto de vista que não depende de computação. Um CIM não mostra detalhes da estrutura dos sistemas. É também conhecido como modelo do domínio, e utiliza um vocabulário familiar aos profissionais e especialistas no domínio em questão (OMG, 2009).

Um PIM é uma visão do sistema de forma independente da plataforma de implementação. Essa independência de plataforma, no entanto, chega até um certo nível,

de forma que o mesmo modelo possa ser reaproveitado em diferentes plataformas do mesmo tipo (OMG, 2009).

Um PSM é uma visão do sistema do ponto de vista de uma plataforma específica. Um PSM combina as especificações de um PIM com detalhes sobre como o sistema utiliza aquele tipo particular de plataforma (OMG, 2009).

A MDA possibilita a transformação de modelos *Unified Model Language* (UML) em outros modelos e/ou artefatos (Mellor, 2004; Mukerji e Miller, 2003). A transformação entre CIM e PIM é menos passível de automação, pois envolve mais decisões e maiores possibilidades de interpretação dos requisitos. Já a transformação entre PSM e código fonte é mais passível de automação, já que o PSM está intimamente ligado com a plataforma de implementação. Na próxima seção apresentam-se as transformações de modelos da MDA.

2.4.2 Transformações de Modelos

Existem transformações que manipulam exclusivamente modelos, conhecidas como transformações *Model to Model* (M2M), ou modelo-modelo. Já outras transformações podem gerar o código fonte a partir de um modelo de entrada definida como transformações *Model to Text* (M2T) ou modelo-texto.

As transformações modelo-modelo realizam a transformação de um modelo de entrada, em um modelo de saída. Estes modelos podem ser instâncias do mesmo metamodelo ou de metamodelos diferentes. O tipo básico de transformação modelo-modelo é a transformação PIM-PSM, ilustrada na Figura 2.10.

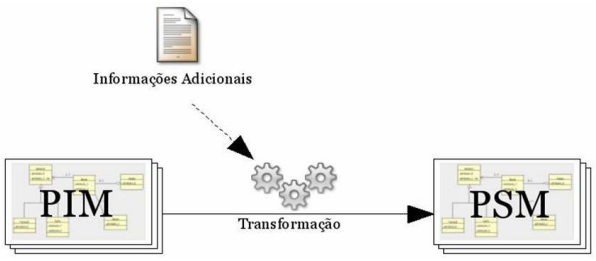

Figura 2.10. Transformação PIM-PSM na MDA.
Fonte: Mellor (2003).

As transformações modelo-texto realizam a transformação de um modelo de entrada (modelo de classes da UML) e realiza a geração dos arquivos de código fonte para uma linguagem (Java, C++, nesC – objeto do trabalho) como saída. O código fonte gerado

pela transformação é representado através de um modelo que posteriormente é serializado em forma de arquivo texto. Na Figura 2.11 é ilustrada a transformação PIM-PSM-Código.

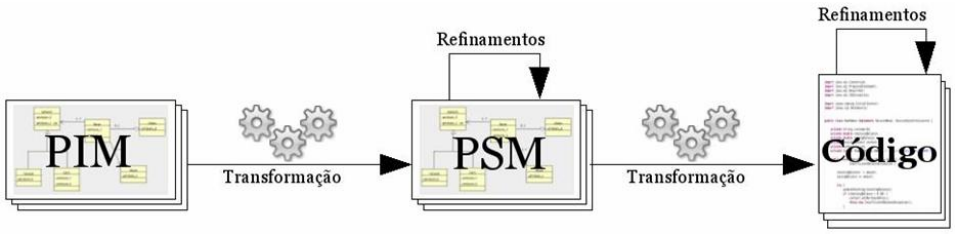

Figura 2.11. Transformação PIM-PSM-Código na MDA.
Fonte: Mellor (2003).

2.4.3 Padrões OMG usados na MDA

Para compreender o mecanismo adotado pela MDA, tais como mapeamento e transformações de modelos, há a necessidade de abordar os padrões e linguagens oferecidas pela OMG, que dão suporte para a MDA. Estes padrões e linguagens têm por objetivo o intercâmbio e padronização na definição de modelos e serão abordados a seguir. Existem outras padronizações da OMG que podem ser usadas na MDA, no entanto, este trabalho concentra-se apenas nas definições do MOF, OCL e XMI.

O *Meta Object Facility* (MOF) é uma especificação da OMG que define uma linguagem abstrata para a descrição de modelos (OMG, 2009). Contudo, vale ressaltar que o MOF não é uma gramática, mas uma linguagem usada para descrever uma estrutura de objetos, em outras palavras, através do MOF, é possível especificar formalmente uma linguagem de modelagem. O MOF é também um *framework* extensível, pois permite que novos padrões de metadados sejam adicionados (Matula, 2003).

O MOF desempenha papel fundamental na MDA, pois, ao permitir que os mapeamentos das transformações sejam definidos em termos de construções MOF, é possível definir transformações entre modelos de diferentes metamodelos. Por exemplo, a transformação de um modelo de classes, em um modelo relacional.

Outra possibilidade é a definição, através do MOF, de metamodelos para as linguagens de programação. Assim, as transformações podem ser definidas com o objetivo de manter o sincronismo entre modelos UML e os modelos específicos para a linguagem de programação utilizada.

A arquitetura do MOF conta com quatro camadas (Matula, 2003), ilustradas na Figura 2.12.

O primeiro nível (M0) corresponde aos dados propriamente ditos. O segundo nível (M1) corresponde aos metadados, ou modelo. São os dados que descrevem os dados. O terceiro nível (M2) o metamodelo para definição de modelos. A especificação UML é um exemplo de metamodelo. O padrão MOF encontra-se no quarto nível (M3). Nesse nível estão os modelos que definem metamodelos, ou seja, MOF é uma linguagem para definição de linguagens de modelagem, como a UML, por exemplo. Como o MOF é um metamodelo, ele próprio é instância de si mesmo, e por isso não existe um quinto nível. A notação de classe da UML é utilizada para representar os metamodelos MOF.

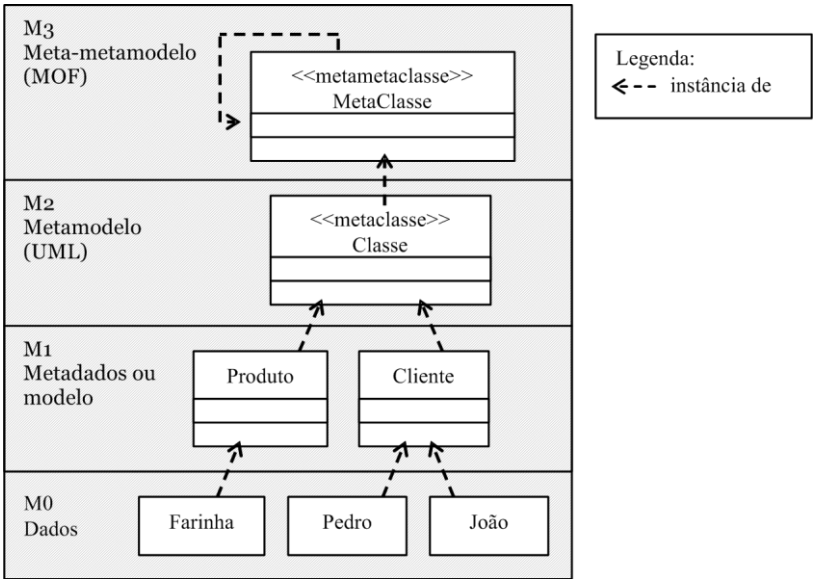

Figura 2.12. Arquitetura do MOF.
Fonte: Matula (2003).

A *Object Constraint Language* (OCL) é uma linguagem formal para descrever expressões em modelos UML (OCL, 2010). As expressões de OCL podem ser utilizadas para especificar operações/ações que, quando executadas, alteram o estado do sistema. Modeladores UML podem usar OCL para especificar restrições detalhadas de aplicações nos seus modelos e consultas (*queries*) no modelo UML. OCL não é uma linguagem de programação, mas uma linguagem de especificação formal e tem por objetivo descrever objetos e suas relações (OCL, 2010).

O documento OMG que descreve a OCL (OCL, 2010) expõe alguns possíveis usos desta linguagem de restrições de objetos: a) como uma linguagem de consulta; b) para especificar elementos estáticos em classes e tipos no modelo de classes; c) para especificar constantes para estereótipos; d) para descrever pré e pós-condições em operações e métodos; e) para especificar restrições em operações; e f) para especificar regras de

derivação para atributos em qualquer expressão sobre um modelo UML. Um documento especificado em XMI, que é explicado mais adiante, pode conter definições OCL.

A MDA também define o *XML Metadata Interchange* (XMI) (OMG, 2009), um formato para representar modelos em XML (*eXtensible Markup Language)*. Este formato define uma estrutura de documento que considera a relação entre os dados e seus correspondentes metadados. Assim, é possível para uma ferramenta, ao interpretar este formato, identificar quais os metadados que descrevem os dados sendo lidos. Diferentes metaníveis podem ser representados, desde o M0 até o M3 (Figura 2.12). O metamodelo UML também pode ser descrito em XMI, e neste caso teria uma referência para o metametamodelo MOF. Por ser baseado em XML, traz consigo várias vantagens, tais como a facilidade de ser interpretado e a possibilidade de se aplicar transformações.

2.5 Considerações Finais

Neste capítulo foram apresentados os conceitos relacionados às RSSFs e a MDA. A programação para RSSF é dirigida a eventos, sendo difícil e trabalhosa, pois exige que o desenvolvedor conheça em nível de detalhes o modelo de programação. Aumentar o nível de abstração destas aplicações, de forma que possibilite aos desenvolvedores um foco maior nas questões conceituais do software ao inves do código fonte, é uma das premissas da metodologia MDA. Aplicar os conceitos da MDA ao desenvolvimento de aplicações para as RSSFs, gerando código nesC é o propósito deste trabalho.

Os principais conceitos do DBC e da MDA foram explorados. As duas abordagens procuram mais qualidade e produtividade no desenvolvimento de software por meio da redução de esforço repetitivo e da adoção de soluções que agregam conhecimento prévio. Os componentes fazem o encapsulamento de artefatos de software diversos, informações e conceitos reutilizáveis de um domínio, incluindo algoritmos, estruturas de dados, funções e etc. A MDA também encapsula o conhecimento necessário para se produzir esses artefatos, mas em forma de transformações que mapeiam conceitos de mais alto nível até o código fonte.

Entendido os conceitos que envolvem as RSSFs e a MDA, é possível então seguir no conteúdo mais relacionado à proposta desta dissertação. No próximo capítulo, são apresentados os trabalhos relacionados que deram sustentabilidade ao desenvolvimento do trabalho proposto.

Capítulo 3- Trabalhos Relacionados

Neste capítulo serão abordadas as principais ferramentas de Geração Automática de Código conhecidas no ambiente acadêmico e no ambiente industrial. Para cada domínio, serão analisadas as características, a relevância do trabalho e a sua contribuição para o desenvolvimento do trabalho proposto. A análise é fundamentada em identificar e comparar as semelhanças dos recursos oferecidos, o foco dos trabalhos, à plataforma utilizada para o desenvolvimento, suporte a extensão, os modelos PIM e PSM, as transformações de modelos, o código fonte gerado, assim como as vantagens e desvantagens. As ferramentas analisadas são: Aplicando Model Driven Development à Plataforma GPGPU, X-Machine Toolkit within Eclipse Plataform, AndroMDA e LabVIEW Wireless Sensor Network Pionner. Nas próximas seções apresenta-se uma descrição de cada uma das ferramentas.

3.1 Aplicando Model Driven Development à Plataforma GPGPU

É uma ferramenta (Carvalho Jr., 2008) que aplica princípios da MDD ao desenvolvimento de aplicações para *Graphics Processing Units* (GPUs). As GPUs (Owens, 2005; Luebke, 2007) são processadores orientados a execução paralela de instruções, otimizados para processar operações sobre vetores, executando no modo SIMD (*Simple Instruction, Multiple Data*). Estes dispositivos são encontrados em video games, computadores pessoais e estações de trabalho, e pode estar situado na placa de vídeo ou integrado diretamente à placa-mãe.

A ferramenta construída aceita modelos como entrada do usuário e, como saída, gera automaticamente parte significativa do código da aplicação, expressa na linguagem *Array-OL* (Boulet, 2007), definida por CUDA (*Compute Unified Device Architecture*) (Nvidia, 2008), uma plataforma de programação para GPGPU (*General-Purpose computation on GPU*) (GPGPU, 2008; Owens, 2005).

No desenvolvimento desta ferramenta utilizou-se quatro componentes (Figura 3.1):

a) Perfil UML de CUDA: especifica os elementos básicos do modelo que são representados pelas tarefas e pelos dados. As tarefas consistem em atividades que possuem potencial para serem realizadas paralelamente. Os dados representam as informações que são manipuladas pelas tarefas. O modelo inclui os relacionamentos de entrada e saída entre tarefas.

b) Componente de Interface Gráfica: fornece uma interface gráfica do software para o usuário representar, alterar e visualizar os dados, as tarefas e as conexões de entrada e saída para as tarefas, por meio da tabela de propriedades que mostra as informações do elemento selecionado no momento.

c) Componente de Manipulação de Modelos: são os modelos utilizados pelo *plugin* GMF (*Graphical Modeling Framework*) para construir a ferramenta. Os modelos são: modelo de definição gráfica, o modelo de definição da ferramenta e modelo de definição de mapeamento, descritos na seção 5.4.2, itens 2, 3 e 4.

d) Componente de Geração de Código: recebe um modelo de entrada elaborado no componente de manipulação de modelos e transfere essas informações aos *templates*, gerando o código da aplicação CUDA como saída.

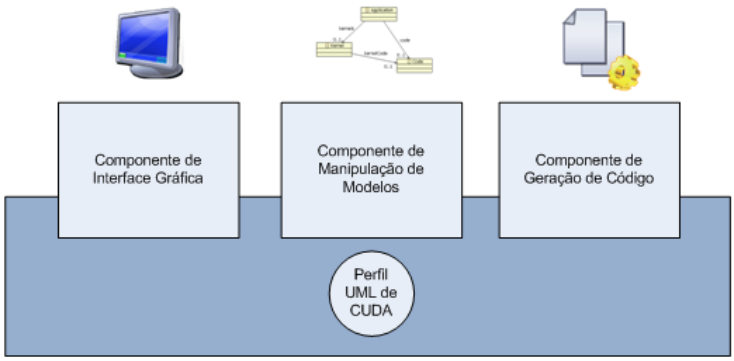

Figura 3.1. Arquitetura de CUDAMDA.
Fonte: Carvalho Jr. (2008).

Os *plugins* EMF (*Eclipse Modeling Framework*) (EMF, 2009) e o GMF (GMF, 2009) do ambiente de desenvolvimento integrado do Eclipse desempenharam um importante papel neste trabalho. No *plugin* EMF foi construído o metamodelo que representa o Perfil UML de CUDA. No *plugin* GMF produziu-se o editor gráfico do modelo e realizou-se a manipulação do modelo. No *plugin* JET (*Java Emitter Templates*) (JET, 2009) - recurso contido no EMF - gerou-se o código de CUDA.

A abordagem MDA adotada neste trabalho fornece uma visão de alto nível da aplicação, encapsulando o conhecimento necessário para se produzir artefatos de software

diversos, informações e conceitos reutilizáveis de um domínio, em forma de transformações que mapeiam conceitos de mais alto nível até o código.

A ferramenta está restrita a utilização de dados primitivos, não permitindo o uso de estruturas complexas dos parâmetros para as tarefas, além de conter alguns erros de interface.

3.2 X-Machine Toolkit within Eclipse Platform

É uma ferramenta (Abdunabi, 2007) *open source*, desenvolvida no ambiente de desenvolvimento integrado do Eclipse, que permite gerar código fonte para a plataforma Java a partir de um método formal intitulado *X-Machine*.

A *X-Machine* é uma máquina geral de computação proposta por Eilenberg e extendida por Holcombe (Holcombe e Ipate, 1998; Kefalas, 2000) que pode modelar estruturas de memória e transições. Uma X-Machine (Figura 3.2) é composta por *stream* de entrada σ e por *stream* de saída γ, onde cada *stream* se configura como fluxo de comunicação por onde passam dados. Cada transição na *X-Machine* faz com um elemento presente no *stream* de entrada σ seja adicionado ao *stream* de saída γ. A X-Machine pode ser definida formalmente como uma óctupla $M = (\Sigma, \Gamma, Q, M, \Phi, F, q0, m0)$ onde:

a) Σ, Γ são os alfabetos de entrada e saída.

b) Q é o conjunto finito de estados.

c) M é o conjunto (possivelmente) infinito chamado memória.

d) Φ é o tipo da máquina M, um conjunto finito de funções parciais φ que mapeiam uma entrada e um estado de memória numa saída e um novo estado de memória, $\boldsymbol{\varphi: \Sigma\ X\ M \rightarrow \Gamma\ X\ M}$.

e) F é a função parcial de próximo estado a qual, dado um estado e uma função do tipo Φ, denota o próximo estado. F é normalmente descrito como uma função de transição de estado, $\boldsymbol{F : Q\ X\ \Phi \rightarrow Q}$.

d) $q0, m0$ são o estado inicial e a memória inicial respectivamente.

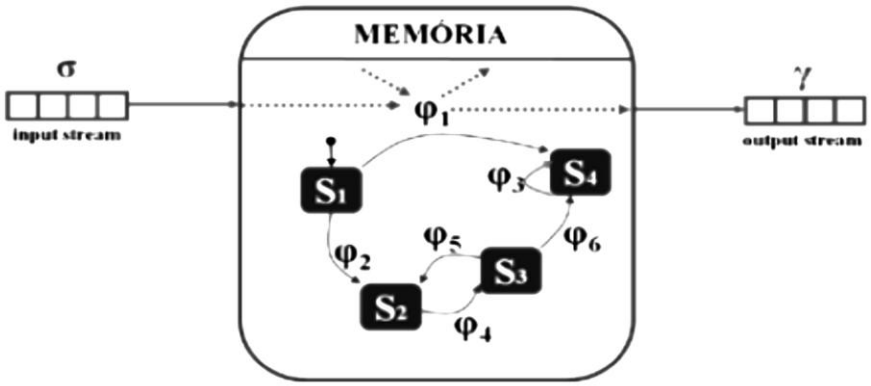

Figura 3.2 Exemplo abstrato de uma X-Machine.
Fonte: Holcombe e Ipate (1998).

A ferramenta possui os seguintes recursos (Figura 3.3):

a) Interface Gráfica para construção de aplicações *X-Machine* – permite ao usuário representar um sistema por meio dos estados (*state*) e das transições (transition).

b) Converte uma especificação XMDL (Walkinshaw, 2002; Kefalas, 2000) de um modelo *X-Machine* para uma representação XML. XMDL é uma linguagem de marcação que usa a estrutura matemática para especificar *Stream X-Machine.*

c) Importação e Exportação de um modelo *X-Machine* no formato XML para a ferramenta.

d) Geração de código na linguagem java a partir da especificação da aplicação *X-Machine* desenvolvida na interface gráfica. À medida que o diagrama é editado, atualiza-se automaticamente um arquivo XML que funciona como ponto de entrada para o *template* de geração de código.

e) A geração funcional de casos de teste permite ao usuário estratégias de teste do modelo, proporcionando uma melhor maturação do software.

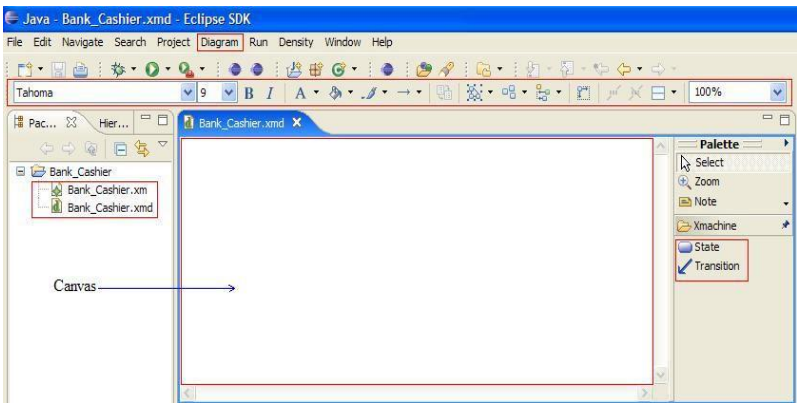

Figura 3.3 Editor da interface gráfica de *X-Machine*.
Fonte: Abdunabi (2007).

A ferramenta possui métodos bem definido que proporcionam os detalhes de como fazer para construir a ferramenta. As tarefas que incluem são: o processo de geração dos editores gráficos e a geração de código fonte. Sob aspecto algum são citados os modelos da MDA e muito menos as transformações de modelos suportados.

3.3 AndroMDA

AndroMDA(Kozikowski, 2005) é uma poderosa ferramenta *open source* que apresenta o método MDA como forma de desenvolvimento de aplicações web e geração de código de aplicações como saída, a partir de especificações UML.

As etapas do funcionamento da ferramenta estão ilustradas na Figura 3.4 e são:

a) Construir um modelo de sistema utilizando um perfil UML.

b) Exportar o modelo para um arquivo XMI.

c) A partir do arquivo XMI exportado, AndroMDA captura esse modelo através do *velocity template engine* e integra ao núcleo.

d) O arquivo é analisado e as características são interpretadas pelos cartuchos (*cartridge*). Os cartuchos são componentes que contém regras de mapeamentos para uma plataforma especifica.

e) Geração do código para plataformas e linguagens de programação, como por exemplo, *Spring*, *Struts*, *Enterprise Java Beans*, *Hibernate* e *Java Server Face.*

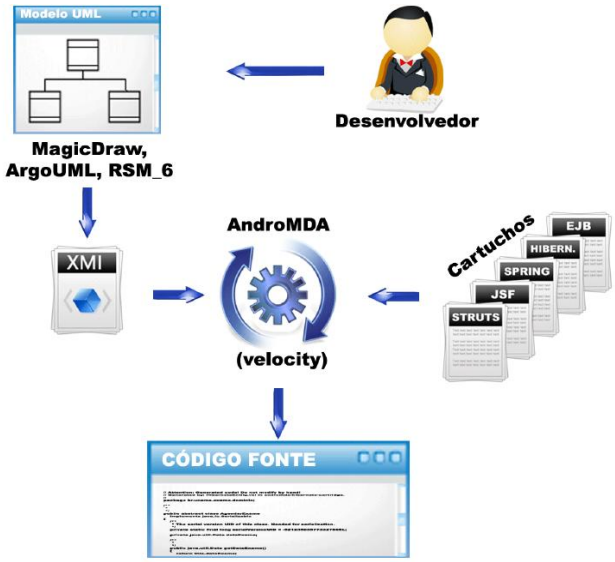

Figura 3.4 Funcionamento de AndroMDA.
Fonte: Kozikowski (2005).

É possível extender a ferramenta AndroMDA através da criação de novos cartuchos que agregam os novos *templates* desejados, implementados em *VTL* (*Velocity Template Language*) (Apache, 2005).

AndroMDA não permite a definição de transformações modelo-modelo (definido na subseção 2.4.2) e não provê recursos de interface para o usuário. A transformação modelo-modelo é realizada por outra ferramenta e a partir desta ferramenta é obtido o modelo PSM no formato esperado pela AndroMDA, que teria a atribuição de gerar código fonte. Com relação à interface gráfica do usuário, existem várias ferramentas no mercado tais como Rational Rose, Magic Draw e ArgoUML entre outras, que fornecem a interface para construção do modelo de domínio (PIM) e transformação do modelo PIM em PSM.

Ter foco no modelo e necessidades organizacionais (PIM) e a possibilidade de reutilizar o modelo PIM em outros projetos é uma vantagem da MDA, pois esses modelos são refletidos no PSM.

3.4 LabVIEW Wireless Sensor Network Pionner

LabVIEW (*Laboratory Virtual Instruments Engineering Workbench*) *Wireless Sensor Network Pioneer* (*LabVIEW WSN*, 2010) é uma ferramenta comercial desenvolvida pela *National Instruments* (*NI*) que trabalha com a abordagem gráfica de programação voltada para o universo das RSSFs. Na forma como está estruturada a ferramenta simplifica e acelera o desenvolvimento destas aplicações, disponibilizando um ambiente de programação do tipo “arraste e solte” para configurar sistemas sem fio, extrair medições, realizar análises, apresentar dados e possibilita ainda conectividade nativa à internet para interações remotas com estes sistemas.

Pode-se resumidamente citar as características do *LabVIEW Wireless Sensor Network Pioneer (LabVIEW WSN)* da seguinte forma:

a) Provê a separação real entre a interface de usuário e o código do programa. Os programas são construídos através do painel frontal e do diagrama de blocos. O painel frontal é utilizado para desenvolvimento da interface gráfica de operação do programa (pelo usuário). O diagrama de blocos é a interface onde é feito o desenvolvimento do algoritmo, programado em linguagem visual.

b) É portável para as plataformas Linux e Windows.

c) Os blocos do *LabVIEW WSN* são peças pré-compiladas, que são ligadas ao corpo do programa à medida que se vai editando. Não é uma linguagem interpretada.

d) O código desenvolvido na linguagem visual é compilado diretamente para linguagem de máquina, não é traduzido para nenhuma outra representação intermediária.

e) Os Blocos permitem que se digite o texto de um programa com as linguagens C e nesC.

f) Possui estruturas básicas de programação, como estrutura de repetição e estrutura condicional. E ainda, recursos para manipulação de funções matemáticas hiperbólica, exponencial e trigonométrica; funções para manipulação de strings; funções para personalizar mensagens de usuários que podem ser transmitidas para o computador host.

LabVIEW WSN não oferece recursos de depuração (*debug*), tais como acompanhar execução passo a passo, monitorar visualmente as variáveis, ou produzir visualizações de valores instantâneos em qualquer ponto do programa. A forma de rastrear o programa é por meio da troca de mensagens entre os nós sensores.

Na Figura 3.5 é mostrado o ambiente de desenvolvimento do *LabVIEW WSN* – A interface gráfica e código em linguagem nesC gerado para uma aplicação exemplo, neste caso o *Blink*, que faz piscar um led com uma taxa de 1 Hz.

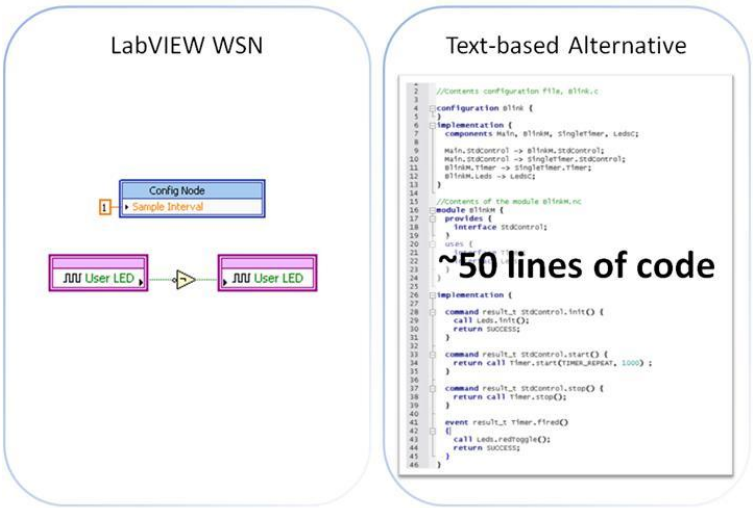

Figura 3.5 Ambiente de desenvolvimento do LabVIEW WSN.
Fonte: LabVIEW WSN (2010).

3.5 Tabela Comparativa dos recursos oferecidos pelas ferramentas

A Tabela 3.1 apresenta o comparativo entre as ferramentas de acordo com os aspectos funcionais explicitados a seguir. Os campos preenchidos com “sim” indicam que os critérios em questão são atendidos totalmente pela abordagem. Os campos preenchidos com “não” indicam que os critérios não são atendidos pela abordagem. Os principais critérios avaliados são: foco, sistema operacional, suporte à extensão, PIM, PSM, transformações modelo-modelo, transformações modelo-texto e código fonte gerado. A seguir será feita uma descrição dos critérios a serem avaliados.

a) Foco: grupos de definições das ferramentas quanto ao enfoque, ou seja, a finalidade da ferramenta, o contexto no qual ela será aplicada.

b) Sistema Operacional: especifica com quais sistemas operacionais a ferramenta funciona.

c) Gratuidade: identifica se a ferramenta é gratuita ou comercial.

d) Suporte à Extensão: identifica se é possível customizar a ferramenta.

e) PIM e PSM: identifca se as ferramentas implementam os modelos PIM e PSM da MDA.

g) Transformações Modelo-Modelo: este tipo de transformação é usado para transformar um tipo de modelo gráfico em outro tipo de modelo gráfico.

h) Transformações Modelo-Texto: este tipo de transformação é usado para transformar um modelo gráfico em um modelo texto.

i) Código Fonte Gerado: código expresso na linguagem de programação obtido pela ferramenta.

Ferramentas / Critérios	Aplicando MDD à Plataforma GPGPU	X-Machine Toolkit	AndroMDA	LabVIEW WSN	Desejado
Foco	Processadores gráficos	X-Machines	Aplicações Web	Redes de Sensores	Redes de Sensores
Sistema Operacional	Windows/ Linux	Windows/ Linux	Windows/ Linux	Windows/ Linux	Windows/ Linux
Gratuidade	Sim	Sim	Sim	Não	Sim
Suporte a extensão	Sim	Sim	Sim	Não	Sim
PIM	Sim	Sim	Sim	Não	Sim
PSM	Sim	Sim	Sim	Sim	Sim
Modelo-Modelo	Sim	Sim	Não	Não	Sim
Modelo-Texto	Sim	Sim	Sim	Sim	Sim
Código fonte gerado	Array-OL	Java	Spring, Struts, EJB, Hibernate e JSF	nesC	nesC

Tabela 3.1. Comparação entre os trabalhos relacionados.

Observa-se na Tabela 3.1 que a ferramenta "Aplicando Model-Driven Development à Plataforma GPGPU" tem foco voltado para o universo dos processadores, utiliza a metodologia MDD e realiza geração de código para a linguagem de baixo nível, denominada Array-OL, da plataforma de programação CUDA. A ferramenta "*X-Machine Toolkit*" emprega o método formal que segue a definição da máquina *X-Machine*, utilizado neste trabalho para gerar código de aplicações em linguagem java. A ferramenta modela o comportamento e funções do sistema e realiza a verificação formal; utiliza a metodologia MDA e realiza as transformações de modelos, com a possibilidade de extensão. A ferramenta "AndroMDA" está voltada para um contexto diferente das ferramentas citadas

anteriormente. Ela é empregada no desenvolvimento e geração de código de aplicações *web*, desenvolvidas em várias plataformas e linguagens de programação. AndroMDA permite suporte a extensão e utiliza o método MDA, mas não contempla a transformação modelo-modelo, empregada pelo método. A ferramenta "*Labview Wireless Sensor Network Pionner*" tem foco diferente das três (3) ferramentas citadas, pois é utilizada no contexto das RSSFs, é proprietária e não segue nenhuma metodologia que contemple a abordagem MDA. O modelo inicial dessa ferramenta, denomindado PSM na MDA, leva em conta as características da plataforma de desenvolvimento. Essa ferramenta gera código em linguagen nesC, apenas a configuração (subseção 2.3.1) da aplicação e não é extensível. Com o intuito de gerar código para as RSSFs, seja a configuração e o módulo das aplicações em lingugem nesC, usando a abordagem que tem como premissa o desenvolvimento baseado em modelos e possibilidade de suporte a extensão surge a ferramenta GAC-RSSFs (Desejado – Tabela 3.1) que visa preencher esta lacuna.

3.6 Considerações Finais

Neste capítulo foi feita uma análise das principais ferramentas de desenvolvimento de aplicações que geram código para aplicações com base nos critérios: (a) foco do trabalho; (b) plataforma do sistema operacional; (c) gratuidade; (d) suporte à extensão; (e) utiliza o modelo PIM; (f) utiliza o modelo PSM; (f) possibilidade de definição de transformação modelo-modelo*;* (g) possibilidade de definição de transformação modelo-texto*;* (h) código fonte gerado.

As abordagens utilizadas pelas ferramentas Aplicando MDD à Plataforma GPGPU e *X-Machine Toolkit* utilizam um formato padrão para a definição das transformações na prática bem definido, especificados pela OMG e implementados no *plugin* GMF do Eclipse. Essas ferramentas possuem uma interface gráfica própria, representando os elementos do domínio que estão sendo modelados.

AndroMDA utiliza um modelo PIM em formato XMI como entrada para as transformações e para a geração de código fonte de forma direta. A ferramenta é composta por *plugins* que podem ser substituídos conforme necessário, funcionando de forma integrada com ferramentas de geração de projeto e de geração de modelagem.

Labview Wireless Sensor Network Pionner não implementa o modelo PIM da MDA. O surgimento de novas tecnologias e plataformas pode aumentar a pressão existente

para a migração, de forma que o esforço despendido em tarefas específicas de uma plataforma pode não ser reaproveitado em outras plataformas. Desta forma, o modelo PIM está voltado para a representação do modelo conceitual, e uma vez completo, poderão ser realizadas transformações de modelos, gerando novamente assim o PSM.

GAC-RSSFs utilizará a metodologia baseada em modelos, empregada nas ferramentas "Aplicando MDD à Plataforma GPGPU" e "*X-Machine Toolkit*" com os padrões especificados pela OMG. Embora essas ferramentas tenham o foco diferente, elas são implementadas no *plugin* GMF do Eclipse, *plugin* também utilizado em GAC-RSSFs.

No capítulo seguinte será descrito o projeto conceitual da ferramenta GAC-RSSFs, proposta neste trabalho como uma solução que contempla os modelos e a transformações de modelos empregados da MDA, o código em linguagem nesC, utilizado pelo sistema operacional TinyOS e o ambiente de desenvolvimento integrado Eclipse com os *plugins* utilizados nesta ferramenta.

Capítulo 4- Projeto Conceitual da Ferramenta GAC-RSSFs

Nos capítulos anteriores foram apresentados os principais conceitos envolvidos nesta pesquisa e os trabalhos relacionados que fornecem o embasamento teórico para o desenvolvimento da ferramenta GAC-RSSF. Neste capítulo são apresentados os requisitos, o esboço da solução conceitual e os *plugins* da plataforma Eclipse utilizados na concepção de GAC-RSSF.

4.1 Requisitos

Um requisito refere-se à definição de uma característica que um sistema ou módulo de um sistema deve satisfazer para obter o resultado pretendido (Sommerville *et* al., 2007).

Os requisitos considerados no desenvolvimento de GAC-RSSFs se dividem em três áreas. Os requisitos dizem respeito: ao uso dos princípios da MDA, à interface gráfica do projetista de aplicações e à portabilidade.

Com relação ao uso dos princípios da MDA, considerou-se os seguintes requisitos:

a) Desenvolver metamodelo centrado no domínio das RSSFs: envolve a identificação das principais entidades que fazem parte das RSSFs, os relacionamentos entre estas entidades, e a linguagem de metamodelagem a ser utilizada. O metamodelo é desenvolvido em uma ferramenta que dê suporte à linguagem estabelecida, sem utilizar os conceitos específicos da plataforma.

b) Desenvolver transformações modelo-modelo: as transformações devem ser desenvolvidas com base nos metamodelos de origem e destino, padrões de transformação, definição da linguagem de transformações a ser utilizada, e implementação da transformação.

c) Desenvolver transformações modelo-texto: a definição da transformação do modelo visual para um modelo texto representado por um XMI da aplicação é insuficiente, sendo definido regras de transformação, que são compiladas e executadas por GAC-

RSSFs, e opcionalmente, completa-se o código gerado, caso necessário, para completar a transformação.

d) Gerar código a partir do modelo especificado na interface gráfica: a geração automática de código produz o código que corresponde ao modelo representado. O resultado desta atividade é uma aplicação nesC, composta pela configuração e pelo módulo. Envolve também a separação entre código gerado e código não-gerado das aplicações, e a complementação com código manual para satisfazer aos requisitos, caso a geração não produza todo o código necessário. Neste caso, a lógica da aplicação não foi modelada previamente.

e) Manter rastreabilidade entre modelos: significa manter ligações ou mapeamentos explícitos entre modelos, após o resultado de uma transformação.

Outro requisito a ser especificado é o desenvolvimento da interface gráfica do projetista de aplicações. Este requisito é essencial, uma vez que o usuário avalia o software através da facilidade com que consegue usá-lo. A interface gráfica deverá:

- Ser simples, funcional e padronizada, de modo que o usuário não necessite utilizar nenhum outro recurso para implementar em GAC-RSSFs.
- Permitir que o usuário possa acessar todas as funcionalidades de forma intuitiva, através de controles usualmente utilizados, como menus, botões e barra de ferramenta.
- Garantir que os recursos da ferramenta devem ser aproveitados, de forma que todos os elementos que contribuam para a construção das aplicações estejam integrados.

Por fim, um outro requisito a ser considerado nesta dissertação é a portabilidade (Pressman, 2005). Os serviços de portabilidade permitem que GAC-RSSFs e a sua estrutura de integração migrem através de diferentes plataformas de sistemas operacionais sem manutenção adaptativa significativa.

4.2 Esboço da Solução

A arquitetura do *plugin* GAC-RSSFs pode ser dividida em três camadas (Ferreira, 2009) distintas que encapsulam conceitos e tecnologias utilizadas na implementação da geração de código fonte – nesC (Figura 4.1). Tais camadas esboçam a solução empregada para atender aos requisitos descritos na seção anterior, e são:

1) Camada da plataforma Eclipse – é a camada mais externa, que engloba o conceito de *plugin* como mecanismo de extensão da ferramenta Eclipse. Ela é composta por ferramentas e tecnologias desta plataforma, produzindo o código fonte nesC.

2) Camada MDA – camada intermediária, que abrange o uso da abordagem MDA na transformação de modelos UML em código.

3) Camada GAC-RSSFs – camada interna que utiliza a UML2 com os perfis, os *plugins* do Eclipse GMF (EMF + GEF) e JET. Dentro dessa camada, o recurso de extensão UML usado é um perfil denominado perfil UML para as RSSFs (Subseção 5.4.1). O *plugin* GMF é utilizado para a construção de editores gráficos, funcionando de forma integrada com os *plugins* EMF e GEF. O EMF é utilizado para acessar e manipular o modelo UML da camada GAC-RSSFs, enquanto o GEF possibilita o desenvolvimento do editor gráfico de GAC-RSSFs. O código fonte é gerado por meio dos *templates* do *plugin* JET. O produto final da camada é o código fonte em linguagem nesC pronto para ser compilado e embarcado no nó sensor.

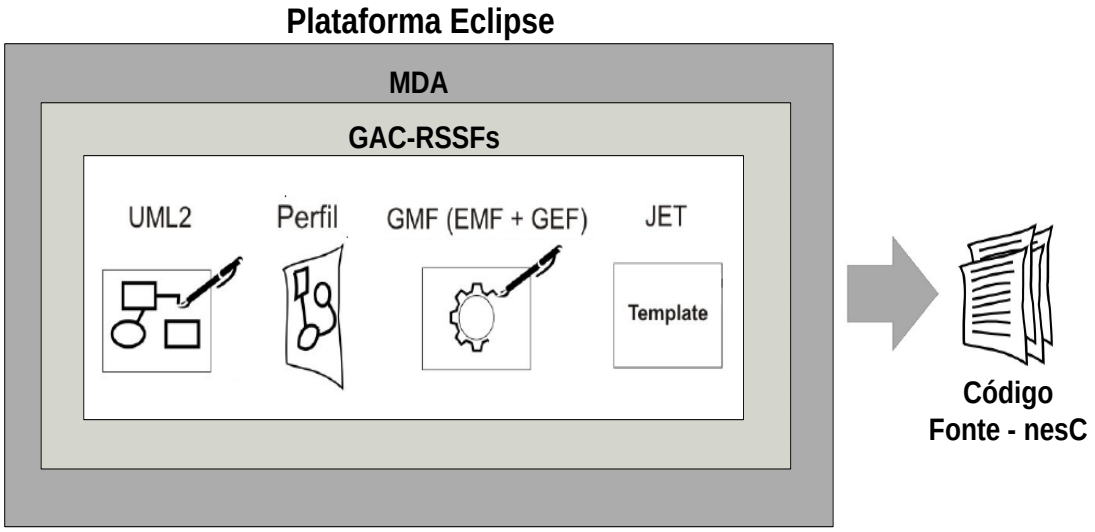

Figura 4.1 Esboço da arquitetura de GAC- RSSFs.
Fonte: adaptado de Ferreira (2009)

Na próxima subseção segue uma descrição da plataforma de desenvolvimento Eclipse com os seus *plugins*, utilizados na construção da ferramenta GAC-RSSFs.

4.2.1 Plataforma Eclipse

O Eclipse é um IDE (*Integrated Development Environment*) lançado em 2001, um projeto que foi doado pela IBM para a comunidade *open source*. Atualmente, o Eclipse tem como principal característica a utilização de uma arquitetura extensível baseada em componentes plugáveis, ou *plugins*, fornecendo um pequeno conjunto de serviços para controlar um grande conjunto de componentes trabalhando conjuntamente. Nesse contexto, todos os componentes do Eclipse, exceto o seu núcleo, foram desenvolvidos como *plugins*.

Com a infra-estrutura genérica, diferentes desenvolvedores podem realizar comunicações ou declarar dependências entre seus *plugins* sem algum problema de compatibilidade, suportando facilmente, por exemplo, a utilização de diversas linguagens de programação apesar do Eclipse ser um IDE baseado em Java.

Os *plugins* conhecidos como *Workbench* e o *Workspace* são indispensáveis na plataforma Eclipse. Eles provêm pontos de extensão para a maioria dos *plugins* nativos do Eclipse. O *Workbench* é o componente que possibilita a outros *plugins* estenderem a interface do Eclipse como menus, barras de ferramentas, requisitar tipos diferentes de eventos e criar novas janelas. O *Workspace* possibilita a interação do usuário com diferentes recursos como projetos e arquivos. A seguir serão analisados os *plugins* que foram utilizados no desenvolvimento de GAC-RSSFs, são eles: *Plugin Development Environment* (PDE), *Eclipse Model Framework* (EMF), *Graphical Edition Framework* (GEF), *Graphical Model Framework* (GMF) e *Java Emitters Template* (JET).

O PDE (PDE, 2009) é um *plugin* nativo do Eclipse que provê recursos e facilidades específicas para o desenvolvimento de *plugins* do Eclipse. O arquivo manifesto plugin.xml é o principal recurso utilizado para esse fim. Através desse arquivo, baseado em XML, o usuário pode especificar dependências entre *plugins*, definir pontos de extensão como itens de menu, especificar *classpath* do projeto e até gerar *builds* de um *plugin* em desenvolvimento de forma prática e rápida, tornando esse *plugin* a base para o desenvolvimento de componentes da plataforma Eclipse. Os *plugins* EMF, GEF, GMF e o JET, incluindo o *plugin* GAC-RSSFs, foram desenvolvidos a partir dessa ferramenta.

O EMF (EMF, 2009) é um *plugin* que visa gerar ferramentas e outras aplicações baseadas em modelos de classes simples, utilizando uma API reflexiva para manipular os metamodelos. Esses metamodelos podem ser descritos por meio de documentos XML, código Java ou um documento Ecore, que se trata de um documento do tipo XMI com uma interface gráfica. Seguindo o conceito de MDA, o EMF faz a sincronia entre o metamodelo

e o código fonte correspondente de GAC-RSSFs. O EMF foi utilizado em GAC-RSSFs para modelar os conceitos empregados na linguagem nesC.

O GEF (GEF, 2009) é um *plugin* que possibilita o desenvolvimento de editores gráficos no Eclipse a partir da modelagem de uma aplicação. O *plugin* utiliza a arquitetura *Model-View-Controller* (MVC), ilustrado na Figura 4.2. Neste padrão, a lógica da aplicação é separada entre o modelo, a visão e o controle. O modelo (*model*) é responsável unicamente pelos dados da aplicação, enquanto a visão (*view*) apresenta os dados da aplicação ao usuário. O controlador (*controller*) tem a responsabilidade de sincronizar estes dois contextos, fazendo com que as alterações realizadas na *view* sejam repercutidas no modelo e que as alterações no modelo façam com que a *view* seja atualizada, de forma a refletir as alterações no modelo. O GEF consiste em duas partes: o *Draw2d* e o *EditParts*. O *plugin Draw2d* provê um conjunto de ferramentas para construir e renderizar objetos gráficos, e seu núcleo se baseia em objetos utilizados para fazer o controle ou mapeamento entre *views* e *models*, observando mudanças que possam ocorrer em um *model* para posterior atualização de sua *view* correspondente. O *plugin* EditParts age como um controlador da arquitetura MVC do GEF.

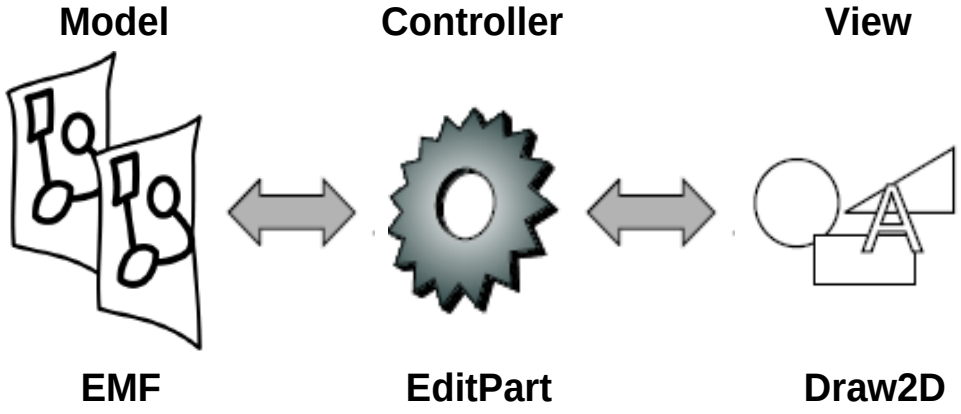

Figura 4.2 Arquitetura MVC do GEF.

O GMF (GMF, 2009) é um *plugin* do Eclipse com suporte para o desenvolvimento rápido de aplicações baseadas em editores gráficos de modelos. Basicamente, o GMF funciona integrando os *plugins* EMF e GEF de forma visual, trazendo praticidade e facilidade a implementação de ferramentas baseadas nos dois *plugins*. O *framework* pode ser dividido em dois componentes principais: o *tooling* e o *runtime*. O componente *tooling* consiste em editores para criação/edição de modelos que descrevem os aspectos

notacionais, semânticos e ferramentais de um editor gráfico, assim como geram código referente à implementação. O projeto gerado depende do componente *runtime* para execução e utilização de seu editor gráfico correspondente, provendo recursos referentes à persistência e sincronização entre modelo e notação visual, agindo de forma a interligar os elementos do EMF e do GEF em tempo de execução.

O JET (JET, 2009) é um *plugin* gerador de código, inspirado em JSP (*Java Server Pages*), projetado para trabalhar utilizando a plataforma Eclipse. Contudo, ao invés de produzir páginas HTML (*Hypertext Markup Language*) em resposta a uma requisição HTTP (*Hypertext Transfer Protocol*), o JET produz recursos do Eclipse (arquivos, pastas, projetos) dado um modelo abstrato como entrada. O componente pode ler diversos tipos de modelos abstratos como entrada, incluindo documentos XML simples ou documentos Ecore do EMF. Sua proposta é a utilização de artefatos que agem como modelo ou *template* para geração de código, possibilitando, durante a ação, a substituição de campos preestabelecidos no *template* pelo conteúdo dos atributos do modelo o qual está sendo transformado.

4.3 Considerações Finais

Ao longo deste capítulo apresentamos o projeto conceitual da ferramenta GAC-RSSFs, composto pelos requisitos, esboço da solução conceitual e os *plugins* da plataforma de desenvolvimento Eclipse. Nos requisitos foram definidas as características que a ferramenta deve atender, englobando aspectos identificados como essenciais. No esboço da solução conceitual de GAC-RSSFs é apresentada a modelagem da arquitetura da aplicação, representado pelo desenho da solução sob a forma de camadas. Cada camada tem atividades e interações bem definidas e específicas. A arquitetura da plataforma Eclipse considera o nível de integração entre as ferramentas e os dados que são produzidos por outras ferramentas, unindo conceitos e tecnologias MDA. Toda essa base sólida de conhecimento foi necessária para que fosse construída a ferramenta proposta por esta dissertação.

Capítulo 5- Desenvolvimento da Ferramenta GAC-RSSFs

O presente capítulo tem o foco no processo de desenvolvimento da ferramenta GAC-RSSFs na plataforma Eclipse. Na primeira parte do capítulo são especificadas as características do *plugin* GAC-RSSFs. Na segunda parte, descreve-se a arquitetura de GAC-RSSFs, para que o objetivo do trabalho fosse atingido: gerar código para as RSSFs a partir de um modelo UML. Na última parte do capítulo, apresenta-se a implementação do *plugin* GAC-RSSFs.

5.1 Características do Plugin GAC-RSSFs

O *plugin* GAC-RSSFs possui as seguintes características:

a) *Plugin* para plataforma Eclipse, desenvolvido na ferramenta *Plug-in Development Environment* (PDE) integrada no *Integrated Development Environment* (IDE) Eclipse versão 3.2.2, Java (1.5), EMF (2.2.2), GEF (3.2.2), GMF (1.0.3), EMF *validation framework* (1.1.0), *XML Schema Infoset Model XSD* (2.2.2) e JET (0.8.1), adicionando a dependência de *plugins*. Os sistemas operacionais utilizados foram: Windows *7* e Windows *XP* com *service pack* 3;

b) Capacidade de converter modelos conceituais baseados em componentes para código em linguagem nesC.

5.2 Arquitetura de GAC-RRSFs

Definido os requisitos do software no capítulo anterior, o próximo passo é a definição de uma arquitetura que atenda aos requisitos contidos nesta especificação. O objetivo desta seção é expor a arquitetura desenvolvida para o software.

Os *plugins* que compõem a ferramenta são:

1) GAC-RSSFs *model plugin*;

2) GAC-RSSFs *edit plugin*;

3) GAC-RSSFs *tree editor plugin*;

4) GAC-RSSFs *graphical editor plugin*;

5) GAC-RSSFs *nesC code generator plugin*.

A Figura 5.1 ilustra, no nível de componentes, a arquitetura desenvolvida para GAC-RSSFs. Nas próximas subseções apresenta-se uma descrição acerca de cada um dos *plugins* desenvolvidos em GAC-RSSFs.

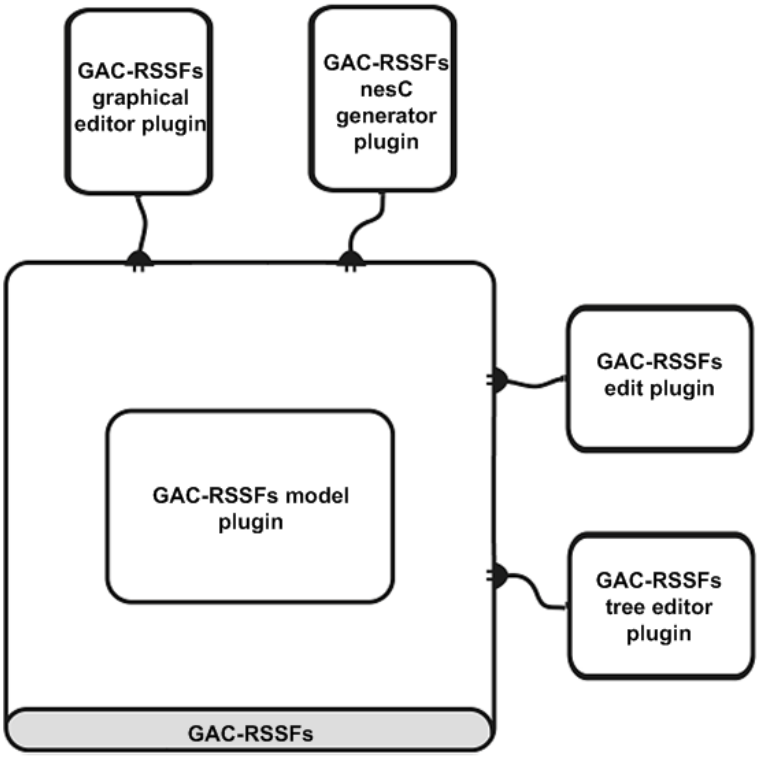

Figura 5.1. Arquitetura de GAC- RSSFs.

5.2.1 GAC-RSSFs Model Plugin

É um *plugin* que utiliza o componente *Model* da arquitetura MVC. Outros *plugins* podem ser construídos com base neste *plugin*. Para o *plugin* EMF gerar código são necessários dois modelos que são o metamodelo "Componentmodel.ecore" e o gerador de modelo "Componentmodel.genmodel". O metamodelo *ecore* foi criado na própria ferramenta EMF, enquanto o *genmodel* foi obtido por meio do processo de derivação do *ecore*. A geração de código será executada por outro *plugin* que suporta a lógica do negócio e o EMF (Gallardo et al., 2003) criará instância da especificação do modelo GAC-RSSFs no formato XML com a extensão "*.xm". O *framework EMF.model* provê a

implementação completa do modelo ecore, bem como a camada de persistência e as funcionalidades. O nome do *plugin* construído na ferramenta é "ufam.ppgee.ft.dcs.component".

5.2.2 GAC-RSSFs Edit Plugin

É um *plugin* que utiliza o componente *View* do padrão MVC. Ele fornece o código que será usado pelo editor para mostrar o nome dos objetos, ícones e a estrutura de árvore das aplicações construídas. O *framework EMF .Edit* (Gallardo et al., 2003) provê a camada de comunicação entre o *EMF.model* e o EMF.*editor*, isto é, entre o modelo e o editor. Tem como ponto de partida a derivação do modelo "Componentmodel.genmodel". O nome do editor de *plugin* construído na ferramenta é "ufam.ppgee.ft.dcs.component.edit".

5.2.3 GAC-RSSFs Tree Editor Plugin

É um *plugin* que utiliza os componentes *View* e C*ontroller* do padrão MVC. Ele provê entrada de dados do usuário com elementos não-gráficos e a visualização desta estrutura na forma de árvore, além das propriedades e os controles da ferramenta. O *framework EMF.Editor* (Gallardo et al., 2003) é obtido a partir do modelo "Componentmodel.genmodel". O nome do editor de *plugin* construído na ferramenta é "ufam.ppgee.ft.dcs.component.editor".

Na Figura 5.2, temos a representação da aplicação exemplo: Blink, no editor com estrutura de árvore. A representação deste exemplo no formato XML é apresentada no Anexo I.

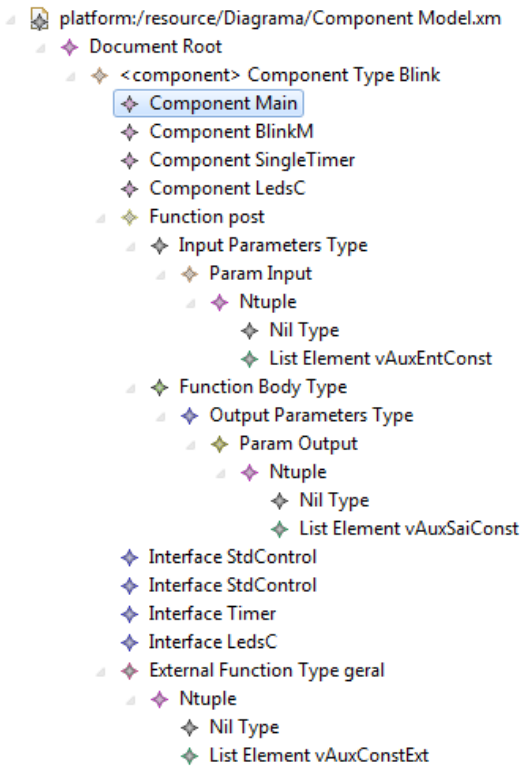

Figura 5.2. Representação da aplicação Blink no editor em estrutura de árvore na ferramenta GAC- RSSFs.

5.2.4 GAC-RSSFs Graphical Editor Plugin

É um *plugin* que utiliza os componentes *View* e *Controller* do padrão MVC. Ele provê a interface gráfica e permite que o projetista desenhe os componentes e as interfaces no diagrama. Como pré-requisito é necessário ter o modelo "Componentmodel.ecore" ao projeto e obter pelo processo de derivação do IDE os quatro modelos que são: modelo de definição gráfica "Componentmodel.gmfgraph" – gerador dos componentes gráficos; o modelo de definição da ferramenta "Componentmodel.gmftool" – gerador da paleta; o modelo de definição de mapeamento "Componentmodel.gmfmap" – mapeia qual classe se associa a um componente gráfico e qual componente gráfico se associa a um elemento da paleta; e o modelo de geração de código "Componentmodel.gmfgen" (Gallardo et al., 2003).

O modelo "Componentmodel.gmfgraph" é usado para definir as figuras, os nós (componentes) e as ligações (interfaces) que são mostrados na ferramenta GAC-RSSFs. Para os componentes, a imagem de um retângulo foi utilizada, enquanto que, para as interfaces, uma linha é utilizada. O modelo "Componentmodel.gmftool" é usado para especificar a paleta (*palette* – ilustrado na Figura 5.6 – letra C), criar os componentes e as interfaces na ferramenta, ações e os ícones dos elementos gráficos. O modelo

"Componentmodel.gmfmap" é usado para ligar os três modelos desenvolvidos: o modelo de domínio "Componentmodel.ecore", o modelo de definição gráfica "Componentmodel.gmfgraph" e o modelo de definição da ferramenta "Componentmodel.gmftool". O modelo "Componentmodel.gmfgen" é usado para gerar o código da interface gráfica da ferramenta GAC-RSSFs. O nome do editor de interface gráfica da ferramenta é "ufam.ppgee.ft.dcs.component.diagram" e a extensão do arquivo que armazena as informações presentes na interface gráfica do usuário é "*.xmd".

5.2.5 GAC-RSSFs nesC Code Generator Plugin

É um *plugin* que fornece a facilidade de gerar o código nesC a partir de uma instância criada no editor gráfico da ferramenta. JET2 (Gallardo et al., 2003) é o *plugin* utilizado para criar GAC-RSSFs nesC Code Generator *Plugin*, representado por: "ufam.ppgee.ft.dcs.component.codeGenerator" na ferramenta. Este *plugin* utiliza um arquivo de entrada no formato XML com extensão "*.xml". Em GAC-RSSFs Model *Plugin* é gerado um arquivo com a extensão "*.xm". A ligação destes *plugin* é realizada por meio da criação de dependência. GAC-RSSFs nesC Code Generator *Plugin* depende de GAC-RSSFs Model *Plugin.*

5.3 Implementação

A ferramenta GAC-RSSFs, conforme pode ser visto na Figura 5.3, é dividida em quatro componentes principais que são: o Perfil UML para as RSSFs, Processo de Geração de Editores, a Interface Gráfica do especialista de aplicações, e a Geração de Código Fonte em linguagem nesC. Nas próximas subseções são detalhados os componentes de GAC-RSSFs.

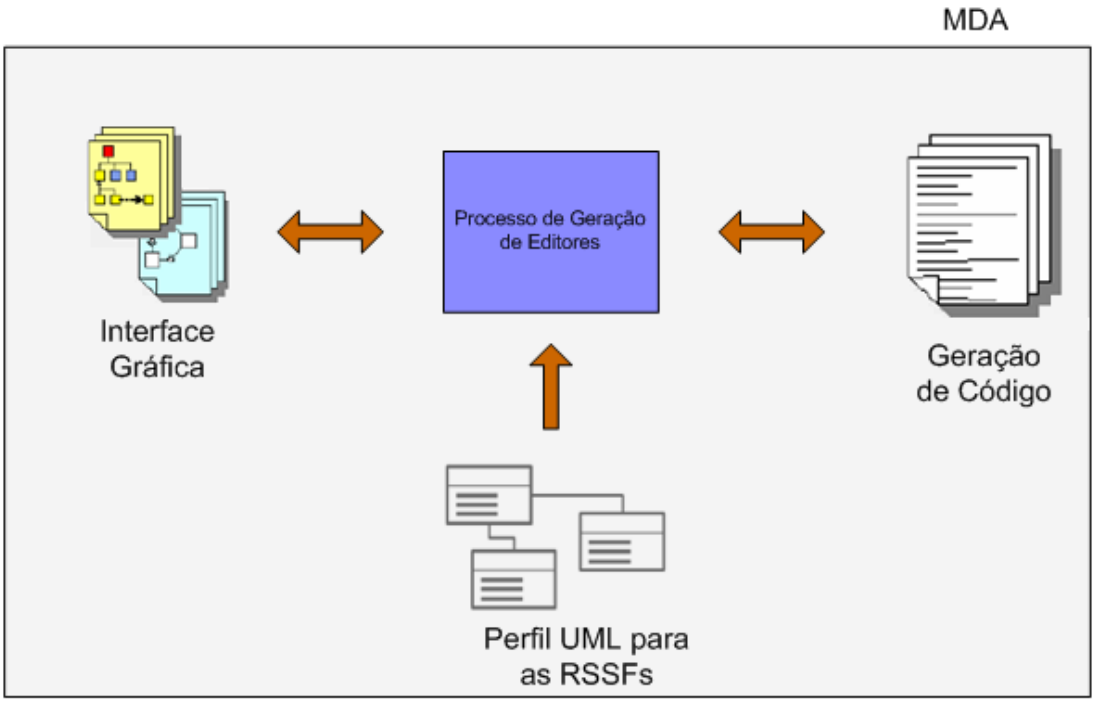

Figura 5.3. Componentes da implementação de GAC-RSSFs.

5.3.1 Perfil UML para as RSSFs

Trata-se da especificação dos elementos da linguagem nesC implementados no modelo *ecore* (Budinsky et al., 2003). Esse modelo consiste num metamodelo de uma linguagem de domínio específico.

Neste metamodelo tem-se como classe raiz, ou *Diagram* segundo (Budinsky et al., 2003), a classe *DocumentRoot*. Esta classe faz a relação de composição com as classes *ListElement*, *ExternalFunctionType*, *ComponentType*, *InputParametersType*, *FunctionBodyType* e *OutputParametersType*. Existem outras classes que fazem parte do modelo, que serão visualizadas na Figura 5.4.

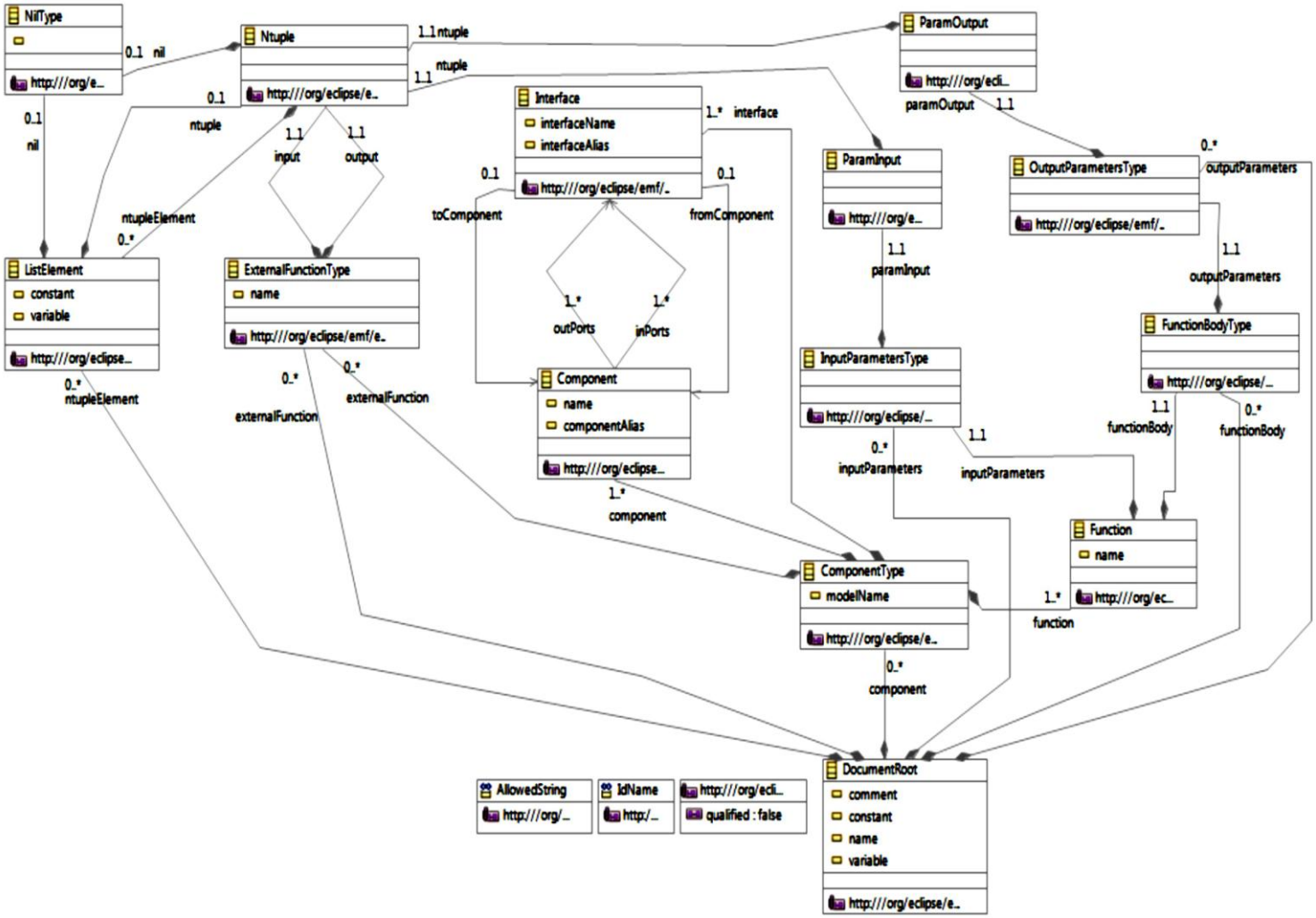

Figura 5.4. Metamodelo ecore de GAC-RSSFs.

Uma lista dos conceitos do metamodelo que representam as classes e o seu significado podem ser consultados na Tabela 5.1.

Elemento	Descrição
AllowedString	A definição do tipo básico de dado String.
Component	São os componentes da aplicação. Possui os atributos ComponentAlias e name.
ComponentType	O componente pai da aplicação, ou seja, o componente no nível mais alto da hierarquia. Possui o atributo ModelName.
DocumentRoot	O elemento raiz do modelo de GAC-RSSFs. Possui os atributos Name e Comment.
ExternalFunctionType	A função externa que será usada na codificação. Possui o atributo Name.
Function	A função interna que será usada na codificação. Possui o atributo Name.
FunctionBodyType	A classe abstrata utilizada para os parâmetros de saída, formado pelo elemento raiz do modelo e pela função.
IdName	A definição do tipo básico com valor Identificador.
InputParametersType	A classe abstrata utilizada para os parâmetros de entrada, formado pelo elemento raiz do modelo.
Interface	A interface de ligação entre os componentes. Possui os atributos InterfaceAlias e InterfaceName.
ListElement	Os parâmetros da função. Possui os atributos Constant e Variable.
NilType	O parâmetro nulo na função.
Ntuple	O tipo genérico de parâmetros usado na função, ou seja, descreve os tipos de parâmetros que podem ser de entrada ou saída.
OutputParametersType	A classe abstrata utilizada para os parâmetros de saída, formado pelo elemento raiz do modelo.
ParamInput	A classe abstrata utilizada para os parâmetros de entrada.
ParamOutput	A classe abstrata utilizada para os parâmetros de saída.

Tabela 5.1. Descrição da classes do metamodelo de GAC-RSSFs.

5.3.2 Processo de Geração de Editores

Para gerar o editor gráfico de GAC-RSSFs começamos por fazer o metamodelo da linguagem usando o modelo *ecore* descrito na seção 5.3.1. O *plugin GMF* apresenta os procedimentos definidos no *Dashboard* (Figura 5.5), que guia o utilizador de modo a efetuar o processo na ordem correta. O *workflow* com o processo de criação de metamodelos até a obtenção do código é formado pelo modelo de domínio, modelo gráfico, modelo de ferramenta, modelo de mapeamento e a geração do editor de diagrama, sendo descrito a seguir.

1) Criação do Modelo de Domínio (*Domain Model*): modelo conceitual que descreve as várias entidades envolvidas num sistema e as relações entre elas.

2) Criação do Modelo Gráfico (*Graphical Definition Model*): modelo que define os elementos gráficos que poderão ser mostrados pelo editor visual.

3) Criação do Modelo de Ferramentas (*Tooling Definition Model*): modelo que possui a definição de todos os elementos que vão estar disponíveis na paleta (Figura 5.6 - C) do editor visual.

4) Criação do Modelo de Mapeamento (*Mapping Model*): modelo que define o mapeamento entre os elementos de domínio e os elementos gráficos.

5) Geração do Editor do Diagrama (*Diagram Editor Generation Model*): editor responsável pela geração de código do diagrama, produzindo um projeto que agirá como um novo *plugin* para a plataforma Eclipse. Através desse editor, podem ser editadas preferências em relação à implementação do *plugin* a ser gerado.

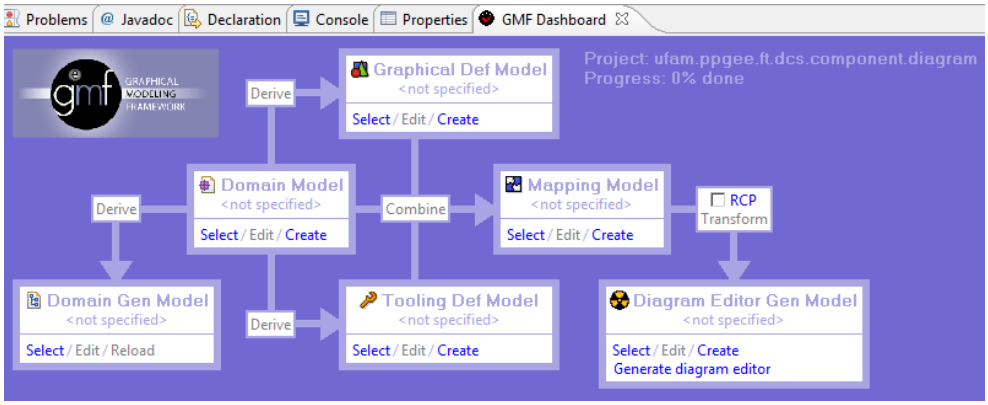

Figura 5.5. O *plugin* GMF Dashboard.

5.3.3 Interface Gráfica

A Figura 5.6 mostra uma visão geral da interface gráfica de GAC-RSSFs juntamente com alguns dos seus componentes destacados com letras. Cada componente desempenha uma funcionalidade específica dentro da ferramenta, sendo discutida brevemente a seguir:

Package Explorer (A) - para cada novo projeto de modelagem que é criado, surge a necessidade de criar e importar arquivos que são utilizados durante o processo de modelagem (bibliotecas, documentos de texto, imagens, e etc). O *package explorer* tem como funcionalidade central permitir a organização dos arquivos em uma estrutura de árvore a fim de que seja possível um melhor gerenciamento e manipulação dos arquivos.

Modeling View (B) - os modelos criados precisam necessariamente ser visualizados com o objetivo de atender dois requisitos básicos, que são a compreensibilidade e a comunicação. Diante dessa necessidade, a *modeling view* permite aos desenvolvedores visualizarem e editarem os modelos de forma interativa.

Palette (C) - os construtores que resultam do diagrama de classe proposto por GAC-RSSFs encontram-se na paleta. Observando a Figura 5.6 é possível identificar os elementos *Component* e *Interface*. O elemento *Component* é utilizado para criar os componentes das aplicações. O segundo elemento, chamado de *Interface*, liga os componentes e especifica a interface. Como exemplo, vamos utilizar a aplicação *Blink* citada nas seções 2.3.1 e 3.4 para ilustrar os elementos da paleta. Os componentes (*Component*) que a aplicação possui são: *Main*, *BlinkM*, *LedsC* e *SingleTimer*. As interfaces são: *StdControl*, *Timer e Leds*.

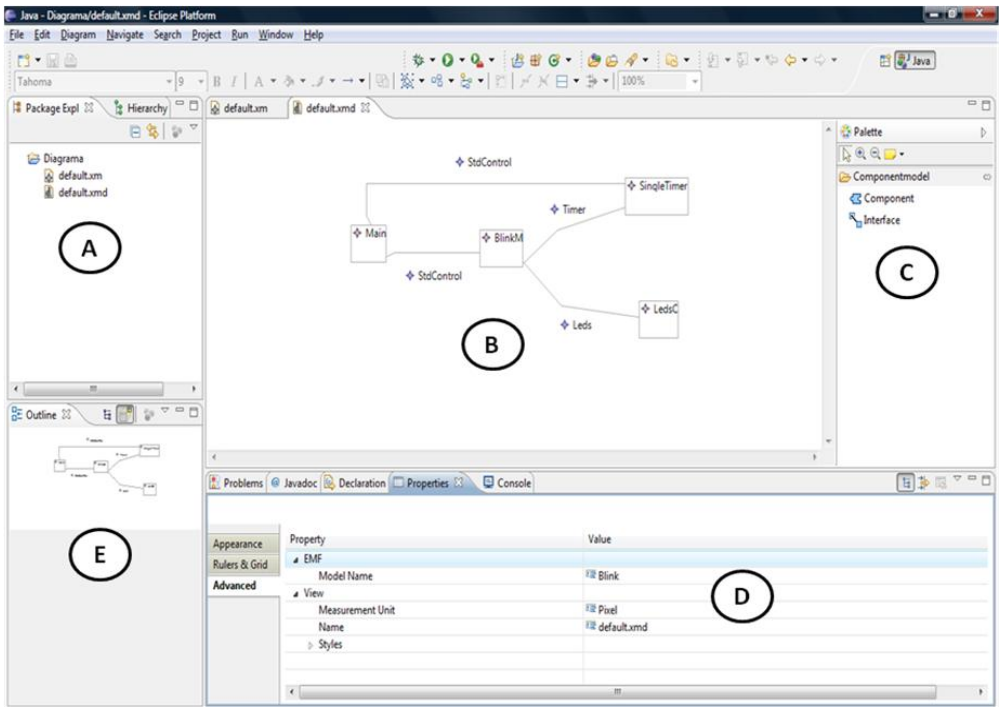

Figura 5.6. Visão geral da interface gráfica de GAC-RSSFs.

Properties View (D) - as preocupações sempre constantes durante a elaboração do metamodelo de GAC-RSSFs consistem na identificação e representação das características dos conceitos presentes no paradigma baseado em componentes. Observando a Figura 5.6, à propriedade *Model Name* é atribuído o valor *Blink*.

Outline View (E) - uma vez que um modelo tenha sido criado, uma visão geral da distribuição dos elementos presentes no mesmo poderá ser visualizada. Esta funcionalidade auxilia a navegação em diagramas muito grandes.

5.3.4 Geração de Código

O *plugin GAC-RSSFs nesC Code Generator* fornece a facilidade de gerar o código nesC a partir de uma dependência criada em GAC-RSSFs Graphical Editor *Plugin*. Este *plugin* é representado na ferramenta por "ufam.ppgee.ft.dcs.component.codeGenerator" e contém os *templates* conforme apresentado na Figura 5.7.

a) O arquivo de controle "main.jet" que controla o processo de geração de código. Este arquivo faz a chamada aos arquivos configuracao.jet, modulo.jet, descricaomodelo.jet, makefile.jet e dump,jet;

b) O arquivo de configuração "configuracao.jet" obtém o conjunto de componentes que são referenciados na aplicação desenvolvida, as interfaces usadas pelo componente e as interfaces providas pelos componentes.

c) O arquivo do módulo "modulo.jet" extrai as interfaces fornecidas e utilizadas pelo componente e a lógica da aplicação.

d) O arquivo "descricaomodelo.jet" informa a quantidade e quais componentes presentes na aplicação.

e) O arquivo "makefile.jet" obtém informações de configuração de nível mais alto da aplicação, permitindo escolher a plataforma alvo e invocar o compilador de aplicações nesC.

f) O arquivo "dump.jet" é utilizado para controle da geração de código para fins de teste.

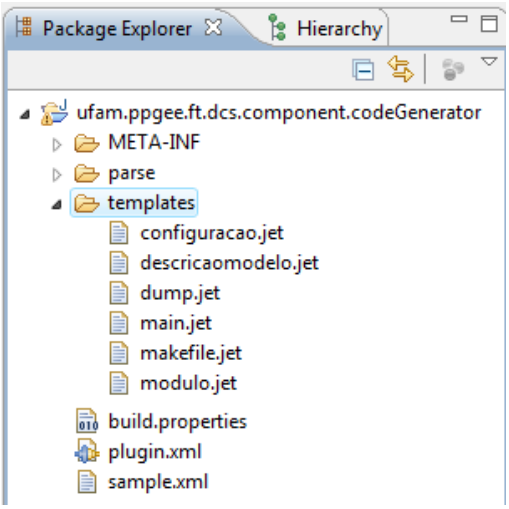

Figura 5.7. *Templates* de geração de código.

5.4 Considerações Finais

Neste capítulo foi detalhado o processo de desenvolvimento da ferramenta GAC-RSSFs. Temos como ponto de partida definição das características dos *plugins* utilizados, seguido da arquitetura da ferramenta, e finalmente, a implementação. Com relação às características dos *plugins* especificou-se a versão compatível, baseada no ambiente do Eclipse na versão 3.2 e a funcionalidade da ferramenta. Com relação à arquitetura da ferramenta é descrito o funcionamento interno dos *plugins* que foram desenvolvidos para GAC-RSSFs. Tais *plugins* funcionam de forma integrada, existindo uma relação de dependência entre eles, com uma seqüência de execução bem definida no *plugin* GMF. Com relação à implementação construiu-se o metamodelo que representa o modelo de domínio das RSSFs, a interface gráfica do especialista da aplicação e os templates de geração de código, além de descrever o processo de criação de sub-editores no desenvolvimento da ferramenta.

No próximo capítulo serão realizados os estudos de caso para a avaliação da ferramenta construída.

Capítulo 6- Construção de Aplicações para as RSSFs a partir da ferramenta GAC-RSSFs

Este capítulo apresenta três estudos de caso com a utilização da ferramenta GAC-RSSFs. Os dois primeiros estudos de caso são aplicações que estão disponibilizadas na distribuição do TinyOS. O primeiro trata de uma aplicação simples, que liga/desliga o led de um nó sensor. O segundo apresenta uma aplicação de referência, que captura periodicamente os valores lidos monitorados pelos sensores, executa processamento local e envia os dados para uma estação base. O terceiro estudo de caso é uma aplicação que não consta na distribuição do TinyOS, baseada na aplicação *Blink*, que obtém a posição do nó sensor.

6.1 Estudo de Caso 1: Blink

Trata-se de uma aplicação que liga/desliga o led vermelho do nó sensor a freqüência de 1 Hz, com disparo de um timer a cada 100 milissegundos (100ms).

6.1.1 Descrição dos componentes e interfaces utilizados na aplicação Blink

Na Tabela 6.1 descreve-se as interfaces com a listagem dos comandos e eventos utilizados na aplicação *Blink*.

Interface	Descrição	Comandos/Eventos
Leds	Provê os leds existentes na plataforma.	Comandos • init (void) • redOn (void) • redOff (void) • redToggle(void)
StdControl	Interface de controle padrão usada para inicializar e finalizar as operações do componente de uma aplicação.	Comandos • init() • start() Evento • stop()
Timer	Provê um contador genérico, que é usado para gerar eventos em intervalos regulares.	Comandos • start (char type, uint32_t interval) • stop (void) Evento • fired(void)

Tabela 6.1. Descrição das interfaces.

Na Tabela 6.2 apresenta-se os componentes com as interfaces providas e usadas pela aplicação *Blink*.

Componente	Interface provida	Interface Usada
LedsC	Leds	-
Main	StdControl	-
SingleTimer	Timer e StdControl	-
Blink	StdControl	Timer e Leds

Tabela 6.2. Descrição dos componentes e interfaces providas e usadas pela aplicação *Blink*.

6.1.2 Desenvolvendo a aplicação Blink na ferramenta GAC-RSSFs

Esta subseção mostra a representação gráfica dos componentes e interfaces (Figura 6.1) utilizados na aplicação *Blink* com a utilização de GAC-RSSFs.

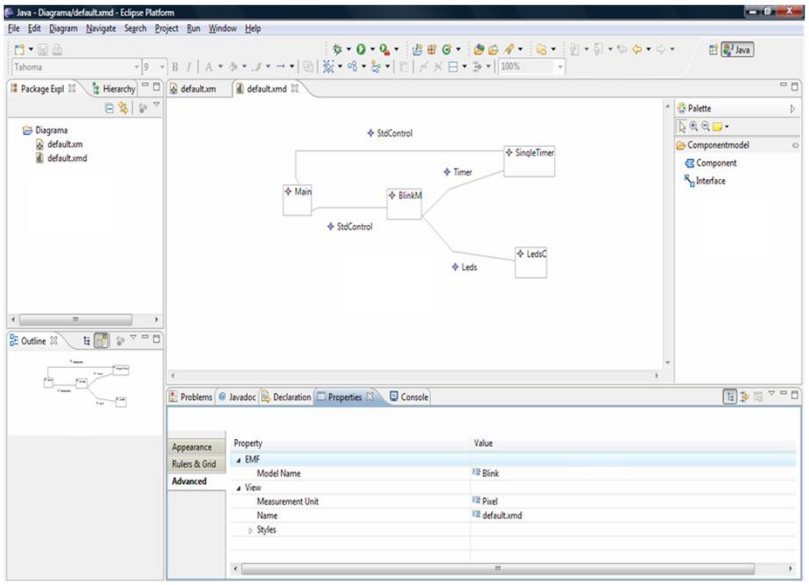

Figura 6.1. Sintaxe gráfica dos componentes e interfaces da aplicação *Blink*.

6.1.3 Modelo armazenado em formato XML gerado a partir da interface gráfica da aplicação Blink

As especificações do modelo de GAC-RSSFs criado na interface gráfica é armazenado no formato de arquivo XML com a extensão "*. xm". A representação XML da aplicação Blink gerada pela a interface gráfica é:

```
<?xml version="1.0" encoding="UTF-8"?>
<component model-name="Blink">
  <component name="Main"/>
  <component name="BlinkM"/>
  <component name="SingleTimer"/>
  <component name="LedsC"/>
  <interface fromComponent="Main" interfaceName="StdControl" toComponent="BlinkM"/>
  <interface fromComponent="BlinkM" interfaceName="Timer" toComponent="SingleTimer"/>
  <interface fromComponent="BlinkM" interfaceName="Leds" toComponent="LedsC"/>
  <interface fromComponent="Main" interfaceName="StdControl" toComponent="SingleTimer"/>
</component>
```

6.1.4 Arquivos gerados pela ferramenta GAC-RSSFs

Na Tabela 6.3 temos os arquivos que compõe a aplicação *Blink* com suas respectivas descrições.

Elemento	Descrição
Blink.nc	É definido o conjunto de componentes que serão conectados dentro da configuração. Neste caso, temos os componentes: *Main*, *Blink, Leds e SingleTimer*
BlinkM.nc	Arquivo que provê o código da aplicação, ou seja, a lógica da aplicação. Declara as interfaces providas e usadas pelos componentes.
DescricaoModelo.txt	Especifica a quantidade de componentes que fazem parte da aplicação Blink.
Makefile	Necessário para a compilação da aplicação.
README	Faz uma descrição da aplicação e as ferramentas utilizadas para visualização dos resultados da comunicação.
SingleTimer.nc	Fornece a abstração para acesso ao temporizador do processador.

Tabela 6.3. Arquivos gerados pela aplicação *Blink*.

O código gerado pela ferramenta GAC-RSSFS da aplicação *Blink*, seja a configuração (*Blink*) e o módulo (*BlinkM*) encontra-se no Anexo II.

6.2 Estudo de Caso 2: Surge

Trata-se de uma aplicação que solicita periodicamente a leitura dos sensores e usa a rede sem fio para transmitir os valores medidos para uma estação base através da árvore de roteamento. O modelo *multihop* é utilizado na comunicação dos nós das RSSFs. Dois ou mais nós da rede que necessitam trocar mensagens podem estar distantes, o que dificulta a comunicação direta. Neste caso, um ou mais nós intermediários recebem e enviam as mensagens até que esta chegue ao nó destinatário.

6.2.1 Descrição dos componentes e interfaces utilizados na aplicação Surge

Na Tabela 6.4 descreve-se as interfaces com a listagem dos comandos e eventos utilizados na aplicação Surge.

Interface	Descrição	Comandos/Eventos
ADC	Conversor analógico digital.	Comando • getData (void)
Leds	Provê os leds existentes na plataforma.	Comandos • init (void) • redOn (void) • redOff (void)
Receive	Trata a recepção de mensagens enviadas por outros nós da rede. A interface é usada para receber dados. Nesta implementação o único destino dos pacotes é a Estação Base.	Evento • TOS_MsgPtr receive (TOS_MsgPtr m)
RouteControl	Controla e monitora o operações de módulos que tratam o protocolo de roteamento.	Comandos • getParent (void)
Send	Trata o envio de dados do tipo TOSMsg	Comandos • send (TOS_MsgPtr msg, uint16_t length) • getBuffer (TOS_MsgPtr msg, uint16_t *length) Evento • sendDone (TOS_MsgPtr msg, result_t success)
StdControl	Interface de controle padrão usada para inicializar e finalizar as operações do componente de uma aplicação.	Comandos • init() • start() Evento • stop ()
Timer	Provê un contador genérico, que é usado para gerar eventos em intervalos regulares.	Comandos • start (char type, uint32_t interval) • stop (void) Evento • fired (void)

Tabela 6.4. Descrição das interfaces da aplicação Surge.

Na Tabela 6.5 apresenta-se os componentes com as interfaces providas e usadas pela aplicação *Surge*.

Componente	**Interface provida**	**Interface Usada**
BCast	StdControl e ReceiveMsg	ReceiveMsg
GenericComm Promiscuous	StdControl, CommControl, SendVarLenPacket, SendMsg e ReceiveMsg	-
LedsC e NoLeds	Leds	-
Main	StdControl	-
MultiHopRouter	StdControl, Receive, Intercept,Send e RouteControl	ReceiveMsg
Photo	ADC e StdControl	-
QueuedSend	StdControl, SendMsg e QueueControl	-
RandomLFSR	Random	-
Sounder	StdControl	-
Surge	StdControl	ADC, Leds, Timer, Send, Receive, RouteControl, StdControl
TimerC	Timer e StdControl	-

Tabela 6.5. Descrição dos componentes e interfaces providas e usadas pela aplicação *Surge*.

6.2.2 Desenvolvendo a aplicação Surge na ferramenta GAC-RSSFs

Esta subseção mostra a representação gráfica dos componentes e interfaces (Figura 6.2) utilizados na aplicação *Surge* com a utilização de GAC-RSSFs.

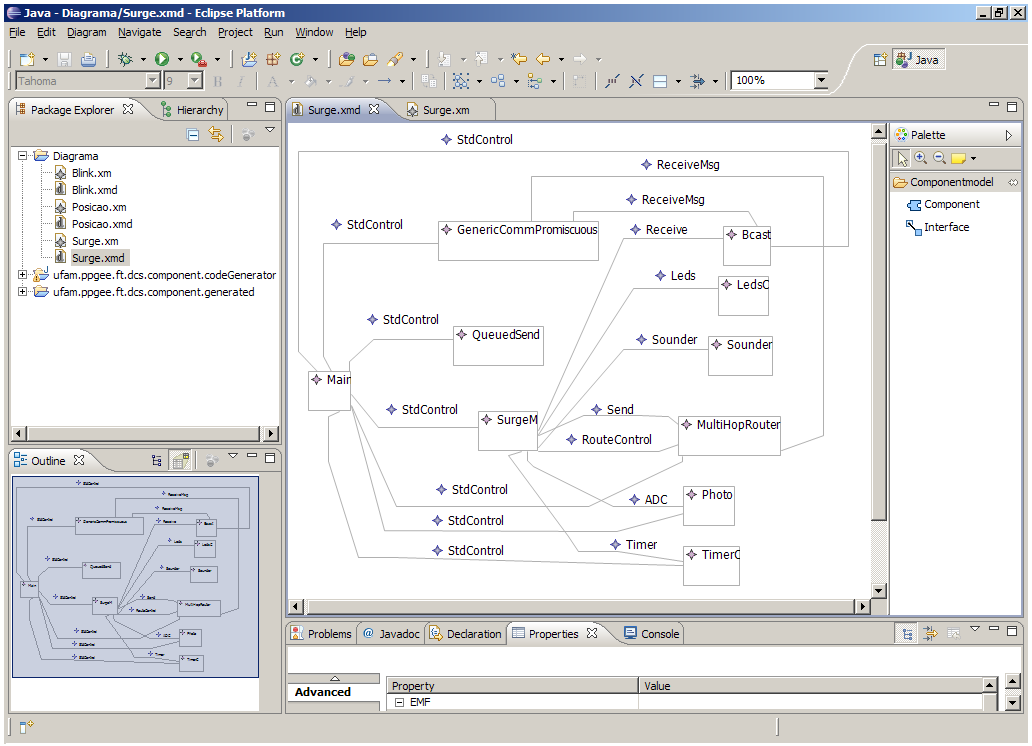

Figura 6.2. Sintaxe gráfica dos componentes e interfaces da aplicação *Surge*.

6.2.3 *Modelo armazenado em formato XML gerado a partir da interface gráfica da aplicação Surge*

A representação XML da aplicação Surge gerada pela a interface gráfica é:

```
<?xml version="1.0" encoding="UTF-8"?>
<component model-name="Surge">
  <component name="SurgeM"/>
  <component name="Main"/>
  <component name="TimerC"/>
  <component name="LedsC"/>
  <component name="QueuedSend"/>
  <component name="Bcast"/>
  <component name="Sounder"/>
  <component name="MultiHopRouter" componentAlias="MultihopM"/>
  <component name="Photo"/>
  <component name="GenericCommPromiscuous" componentAlias="Comm"/>
  <interface fromComponent="Main" interfaceName="StdControl" toComponent="QueuedSend"/>
  <interface fromComponent="Main" interfaceName="StdControl" toComponent="SurgeM"/>
  <interface fromComponent="Main" interfaceName="StdControl" toComponent="MultiHopRouter"/>
  <interface fromComponent="Main" interfaceName="StdControl" toComponent="Photo"/>
  <interface fromComponent="Main" interfaceName="StdControl" toComponent="TimerC"/>
  <interface fromComponent="Main" interfaceName="StdControl" toComponent="GenericCommPromiscuous"/>
  <interface fromComponent="SurgeM" interfaceName="Receive" toComponent="Bcast"/>
```

```
        <interface fromComponent="SurgeM" interfaceName="Leds" toComponent="LedsC"/>
        <interface fromComponent="SurgeM" interfaceName="Sounder" toComponent="Sounder"/>
        <interface fromComponent="SurgeM" interfaceName="Send" toComponent="MultiHopRouter"/>
        <interface fromComponent="SurgeM" interfaceName="RouteControl" toComponent="MultiHopRouter"/>
        <interface fromComponent="SurgeM" interfaceName="ADC" toComponent="Photo"/>
        <interface fromComponent="SurgeM" interfaceName="Timer" toComponent="TimerC"/>
        <interface fromComponent="Bcast" interfaceName="ReceiveMsg" toComponent="GenericCommPromiscuous"/>
        <interface fromComponent="MultiHopRouter" interfaceName="ReceiveMsg"
toComponent="GenericCommPromiscuous"/>
        <interface fromComponent="Main" interfaceName="StdControl" toComponent="Bcast"/>
      </component>
```

6.2.4 Arquivos gerados pela ferramenta GAC-RSSFs

Na Tabela 6.6 temos os arquivos que compõe a aplicação *Surge* com suas respectivas descrições.

Elemento	Descrição
DescricaoModelo.txt	Especifica a quantidade de componentes que fazem parte da aplicação Surge.
Makefile	Necessário para a compilação da aplicação.
README	Faz uma descrição da aplicação e as ferramentas utilizadas para visualização dos resultados da comunicação.
Surge.h	Arquivo de cabeçalho que define as variáveis a ser ocupadas, enumerações e o tipo de dado SurgeMsg.
Surge.nc	Especifica o conjunto de componentes que serão conectados dentro da configuração. Neste caso, temos os componentes: GenericCommPromiscuous, LedsC, NoLeds, Main, MultiHopRouter, Photo, QueuedSend, RandomLFSR, Sounder e TimerC.
SurgeCmd.h	Define a estrutura do tipo registro (struct) SurgeCmdMsg e a enumeração (enum) AM_SURGECMDMSG.
SurgeM.nc	Arquivo que provê o código da aplicação, ou seja a lógica da aplicação. Declara as interfaces providas e usadas pelos componentes.

Tabela 6.6. Arquivos gerados pela aplicação Surge.

O código gerado pela ferramenta GAC-RSSFS da aplicação *Surge*, seja a configuração (*Surge*) e o módulo (*SurgeM*) encontram-se no Anexo III.

6.3 Estudo de Caso 3: Posição do nó sensor

Trata-se de uma aplicação que obtém a posição do nó sensor com disparo de um timer a cada 100 milissegundos (100ms). A aplicação não está disponibilizada na distribuição do TinyOS.

6.3.1 Descrição dos componentes e interfaces utilizados na aplicação Posição do Nó sensor

Na Tabela 6.7 descreve-se as interfaces com a listagem dos comandos e eventos utilizados na aplicação Posição do nó sensor.

Interface	Descrição	Comandos/Eventos
ADC	Conversor analógico digital.	Comando • getData (void) Evento • dataReady (uint16_t data)
Location	Fornece a localização fisica do nó sensor em três dimensões (3D).	Comando • getLocation(void) Evento • locationDone (location_3d_t *loc)
StdControl	Interface de controle padrão usada para inicializar e finalizar as operações do componente de uma aplicação.	Comandos • init() • start() Evento • stop()
Timer	Provê un contador genérico, que é usado para gerar eventos em intervalos regulares.	Comandos • start (char type, uint32_t interval) • stop (void) Evento • fired (void)

Tabela 6.7. Descrição das interfaces da aplicação Posição do nó sensor.

Na Tabela 6.8 apresenta-se os componentes com as interfaces providas e usadas pela aplicação Posição do nó sensor.

Componente	Interface provida	Interface Usada
Main	StdControl	-
Posicao	StdControl	Timer e ADC
Photo	ADC e StdControl	-
SingleTimer	Timer e StdControl	-

Tabela 6.8. Descrição dos componentes e interfaces providas e usadas pela aplicação Posição do nó sensor.

6.3.2 *Desenvolvendo a aplicação Posição do nó sensor na ferramenta GAC-RSSFs*

Esta subseção mostra a representação gráfica dos componentes e interfaces (Figura 6.3) utilizados na aplicação Posição do nó sensor com a utilização de GAC-RSSFs.

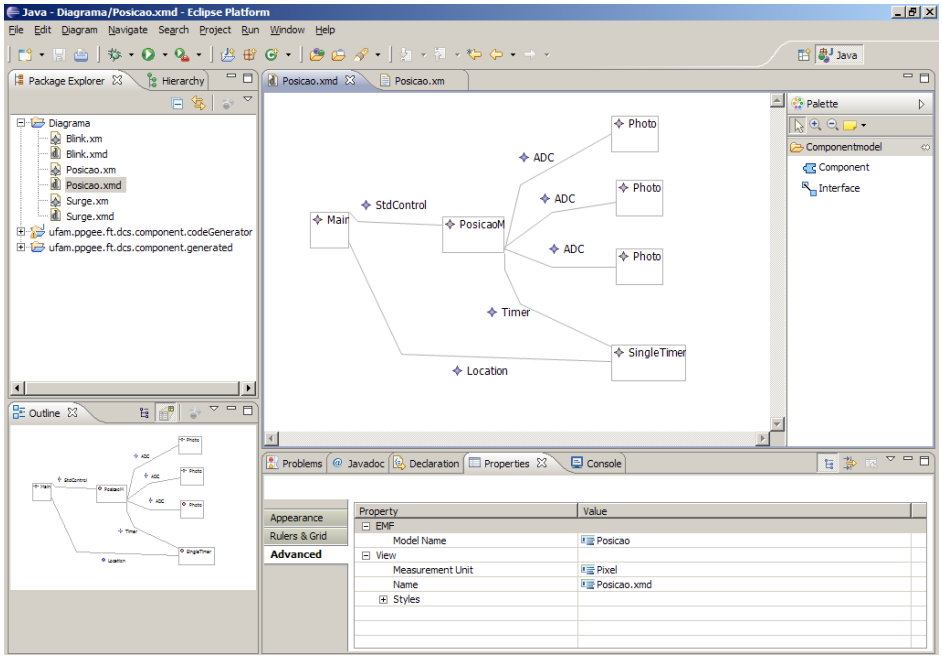

Figura 6.3. Sintaxe gráfica dos componentes e interfaces da aplicação Posição do nó sensor.

6.3.3 *Modelo armazenado em formato XML gerado a partir da interface gráfica da aplicação Posição do nó sensor*

A representação XML da aplicação Posição do nó sensor gerada pela a interface gráfica é:

```
<?xml version="1.0" encoding="UTF-8"?>
<component model-name="Posicao">
  <component name="Main"/>
  <component name="PosicaoM"/>
  <component name="Photo" componentAlias="Photo1"/>
  <component name="Photo" componentAlias="Photo2"/>
  <component name="Photo" componentAlias="Photo3"/>
  <component name="SingleTimer"/>
  <interface fromComponent="Main" interfaceName="StdControl" toComponent="PosicaoM"/>
  <interface fromComponent="Main" interfaceName="Location" toComponent="SingleTimer"/>
  <interface fromComponent="PosicaoM" interfaceName="Timer" toComponent="SingleTimer"/>
  <interface fromComponent="PosicaoM" interfaceName="ADC" toComponent="Photo" interfaceAlias="ADCZ"/>
```

```
  <interface fromComponent="PosicaoM" interfaceName="ADC" toComponent="Photo" interfaceAlias="ADCY"/>
  <interface fromComponent="PosicaoM" interfaceName="ADC" toComponent="Photo" interfaceAlias="ADCX"/>
</component>
```

6.3.4 Arquivos gerados pela ferramenta GAC-RSSFs

Na Tabela 6.9 temos os arquivos que compõe a aplicação Posição do nó sensor com suas respectivas descrições.

Elemento	Descrição
DescricaoModelo.txt	Arquivo que especifica a quantidade de componentes que fazem parte da aplicação Posicao.
Makefile	Arquivo necessário para a compilação da aplicação.
Posicao.nc	Arquivo que especifica o conjunto de componentes que serão conectados dentro da configuração. Neste caso, temos os componentes: *Main, PosicaoM, Photo e SingleTimer*
PosicaoM.nc	Arquivo que provê o código da aplicação, ou seja a lógica da aplicação. Declara as interfaces providas e usadas pelos componentes.
README	Arquivo que faz uma descrição da aplicação e as ferramentas utilizadas para visualização dos resultados da comunicação.

Tabela 6.9. Arquivos gerados pela aplicação Posição do nó sensor.

O código gerado pela ferramenta GAC-RSSFS da aplicação *Posicao*, seja a configuração (*Posicao*) e o módulo completo (*PosicaoM*) encontram-se no Anexo IV.

6.4 Considerações Finais

Apresentamos neste capítulo implementações de estudos de caso que foram realizados para análise da ferramenta proposta. O ambiente visual permitiu a melhor compreensão de como a aplicação é construída e como os componentes interagem entre si por meio das interfaces. À medida que os componentes e interfaces vão sendo criados na interface gráfica o arquivo XML guarda tais elementos, para posterior geração de código nesC. Os exemplos *Blink* e *Surge* foram testados nos nós sensores e simulados nas ferramentas TOSSIM e TinyViz no ambiente do TinyOS, validando o código gerado pela ferramenta.

Na aplicação *Posicao* sua lógica não foi modelada previamente no sistema operacional TinyOS, e muito menos na ferramenta LabVIEW Wireless Sensor Network Pionner. Consequentemente, a ferramenta LabVIEW Wireless Sensor Network Pionner não é capaz de gerar o módulo da aplicação. Na ferramenta GAC-RSSFs é gerado parte do módulo da aplicação. Os elementos obtidos na geração do módulo são as interfaces que são

usadas e providas, os métodos da interface *StdControl* e os métodos da interface *Timer*. No Anexo V é apresentado o módulo da aplicação *Posição* gerado a partir de GAC-RSSFs.

Capítulo 7- Resultados e Discussões

O processo de implementação de GAC-RSSFs mostrou que o diagrama de classe da UML deixa de exercer somente a função de documentar o processo de desenvolvimento e passa também a exercer o papel fundamental de metamodelo utilizado como ponto de partida do processo de geração da ferramenta GAC-RSSFs. No metamodelo são realizadas transformações modelo-modelo e modelo-código fonte, através das tecnologias e conceitos MDA.

Na ferramenta GAC-RSSFs foi produzido um editor gráfico na forma de um *plugin* para o Eclipse, a partir da criação do metamodelo proposto. Para cada instância de aplicação gera-se um arquivo XML, que servirá de entrada para a ferramenta de template JET, e posterior geração de código *nesC*. Desta forma, GAC-RSSFs também pode ser considerada uma ferramenta MDA.

Na ferramenta são representados os componentes e as interfaces no editor gráfico ou no editor com estrutura de árvore.

No editor gráfico temos o menu *Diagram* (A), *toolbar* (B), a *Palette* (C) e as *Canvas* (D) como mostra a Figura 7.1.

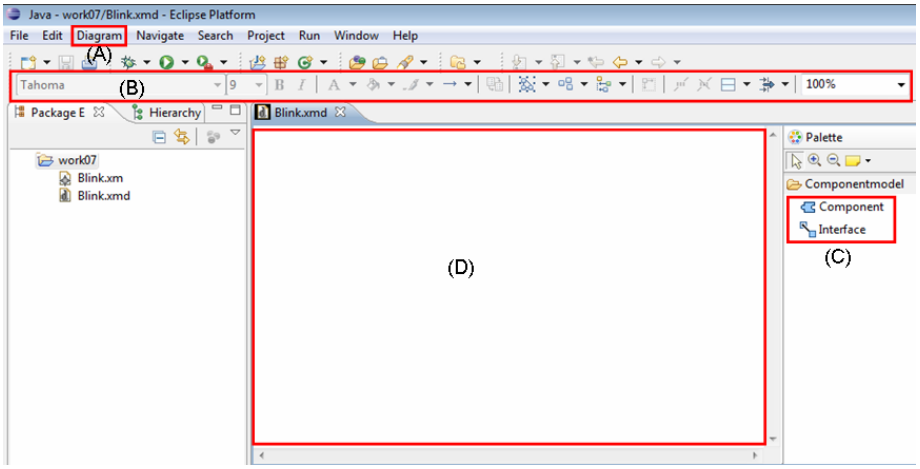

Figura 7.1. Editor gráfico de GAC-RSSFs.

No editor com estrutura de árvore são representados os elementos como itens de uma árvore na hierarquia que foram modelados os elementos do modelo "Componentmodel.ecore", de forma a refletir os relacionamentos no esquema XML. A representação do editor com estrutura de árvore é mostrada nas Figuras: 7.2, 7.3 e 7.4.

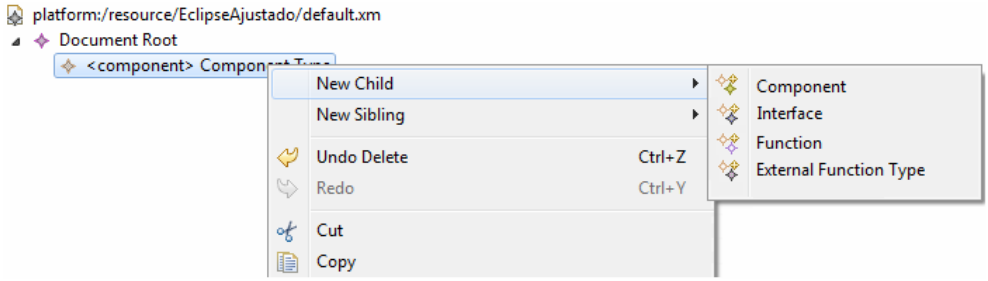

Figura 7.2. Menu usado na construção dos elementos de *Component Type*.

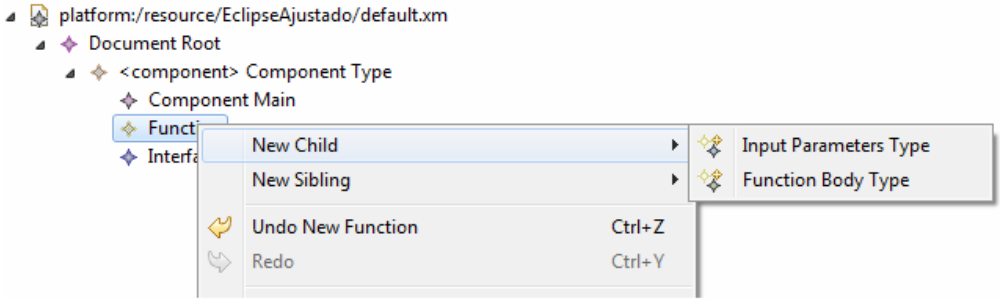

Figura 7.3. Menu usado na construção dos elementos da *Function*.

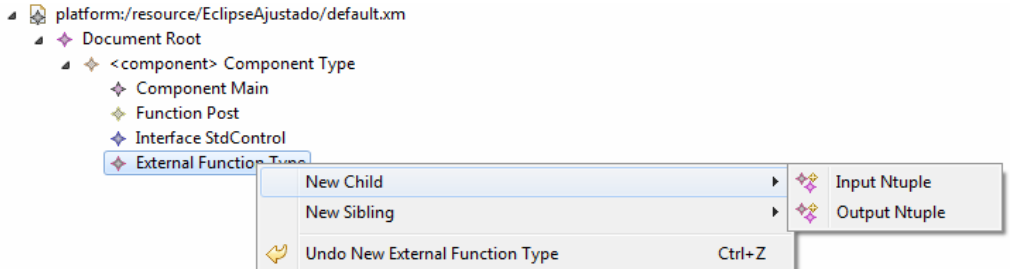

Figura 7.4. Menu usado na construção dos elementos de *External Function Type*.

A ferramenta GAC-RSSFs atende o requisito de usabilidade, pois o esforço para aprender, operar, preparar a entrada (aplicação baseada em componente) e interpretar a saída (código fonte em linguagem nesC) são extremamente simples em GAC-RSSFs, exercendo um impacto positivo sobre a capacidade de produzir aplicações para as RSSFs, de forma a manter a consistência da aplicação e a documentação da ferramenta.

A representação da parametrização das interfaces é algo que não é contemplado na interface gráfica, somente nos *templates* de configuração e módulo.

Duas aplicações distintas foram construídas para avaliar o tamanho do código gerado com o auxílio da ferramenta GAC-RSSFs e sem o auxílio da ferramenta. Com o auxilio da ferramenta o desenvolvedor elaborou o modelo da aplicação e gerou o código fonte. Sem o auxílio da ferramenta o desenvolvedor utilizou o editor texto para produzir todo o código fonte. O objetivo desta avaliação foi averiguar se a ferramenta seria ou não útil no processo de desenvolvimento. A Tabela 7.1 contém os resultados do tamanho do código das aplicações *Blink* e *Surge*.

Aplicações	**Configuração/ Módulo**	**Sem a Ferramenta**	**GAC-RSSFs**
Blink	Blink	312 bytes	339 bytes
	BlinkM	519 bytes	595 bytes
Surge	Surge	965 bytes	990 bytes
	SurgeM	4.561 bytes	4.747 bytes

Tabela 7.1. Comparativo do tamanho das aplicações sem a ferramenta e com a ferramenta GAC-RSSFs.

Nota-se que em todos os casos o tamanho do código fonte gerado das aplicações pela ferramenta GAC-RSSFs é maior do que o código fonte desenvolvido nos editores de texto (bloco de notas, notepad, gedit, kwrite), tanto o módulo como a configuração das aplicações escritas em nesC. Os fatores que agravaram o aumento do tamanho de código fonte são: a quantidade de caracteres, os espaços em branco e os espaços entre linhas. Em sistemas com restrições severas de memória, como é o caso das RSSFs, o tamanho do código é um fator crítico no funcionamento das aplicações no nó sensor.

Outro aspecto é que a ferramenta foi desenvolvida com foco na corretude do código gerado. Aspectos de performance, como otimização do código gerado, são bastante importantes no desenvolvimento de aplicações das RSSFs, mas não foram considerados.

Capítulo 8- Considerações Finais

A ferramenta apresentada neste trabalho possibilita automatizar o processo de construção de aplicações para as RSSFs, tendo como foco a geração de código em nesC a partir de modelos baseados em componentes representados através de uma interface gráfica pelo projetista de aplicações. A interface gráfica facilitará a montagem de aplicações, fazendo uso do componente e da interface disponibilizados na paleta de componentes (ilustrado na Figura 5.6 – letra C).

A abordagem orientada a modelos adotada neste trabalho proporciona o aumento do nível de abstração da aplicação, de modo que o esforço ficará centrado nos modelos e não mais nos códigos fonte, possibilitando aos desenvolvedores um melhor entendimento do negócio do sistema e diminuição dos erros na codificação, algo comum quando se está lidando com programação de baixo nível, que normalmente trabalha com códigos não triviais.

Algumas limitações puderam ser detectadas durante esse trabalho. Essas limitações são apresentadas a seguir.

a) O foco maior da ferramenta está na representação da estrutura estática dos componentes e geração do código fonte. Inferir e gerar parte do comportamento dinâmico de certas operações nas RSSFs não foi contemplado neste trabalho.

b) A transformação do modelo em geração de texto implementada não suporta a geração não-destrutiva de código fonte. Alterações realizadas sobre o código fonte gerado não são preservadas após uma nova geração de código.

8.1 Trabalhos Futuros

Como trabalhos futuros, sugere-se:

a) Criar diagramas que representem o comportamento dinâmico dos componentes, sejam os comandos, os eventos e as tarefas. A representação da máquina de estado do componente proporciona geração de uma maior porcentagem de código fonte da lógica da aplicação.

b) Criar arquivos de *template* que possibilitem gerar código em outras linguagens de programação, tais como Java e C++.

8.2 Dificuldades Encontradas

No início dos trabalhos, a maior dificuldade foi entender o funcionamento das RSSFs, ou seja, a concepção em si das RSSFs, conhecer a plataforma do sistema operacional TinyOS e as aplicações baseadas em componentes escritas na linguagem nesC.

A segunda e maior dificuldade foi modelar os componentes, ou seja, criar o metamodelo que contempla de forma genérica as aplicações do TinyOS. Foram realizadas várias modelagens nas ferramentas *Rational Rose* e no *plugin* EMF do Eclipse. A primeira ferramenta utilizada de modelagem foi o *Rational Rose*, na qual o modelo de classe era criado, e posteriormente, exportado para o *plugin* EMF. Na exportação, algumas representações na ferramenta *Rational Rose* não eram suportadas no *plugin* EMF, gerando erros diversos. Logo, a ferramenta *Rational Rose* foi descartada, e passou-se a utilizar o próprio *plugin* do Eclipse EMF. A maior parte do tempo gasto neste trabalho foi no desenvolvimento do modelo que representa o paradigma baseado em componente.

A terceira dificuldade está relacionada ao uso da tecnologia Eclipse no desenvolvimento. Os problemas são: há conflito de versões entre os *plugins* utilizados, seja o EMF, o GEF, o GMF e o JET, além das dependendências entre *plugins*. Na versão do Eclipse 3.2.2, usada neste trabalho, há falta de documentação sobre os bugs comuns ao utilizar o *framework*. Novas tecnologias estão em desenvolvimento, aprender e conhecer efetivamente estes *plugins* a fundo, de forma a produzir *plugins* de alta qualidade leva-se bastante tempo.

A quarta dificuldade encontrada está na customização do código referente ao diagrama gerado pelo GMF. Muitas classes são geradas automaticamente através da construção de um projeto GMF, contudo, há pouca documentação que facilite a identificação das funcionalidades, sendo que, uma das grandes fontes de informação sobre a utilização e customização desse *plugin* só pode ser obtida através de lista de discussão.

A quinta dificuldade está no desenvolvimento do *plugin* na plataforma do sistema operacional Windows Vista. O primeiro ponto é que o sistema operacional TinyOS apresenta problemas de incompatibilidade com o Windows Vista, logo não obteve-se êxito na instalação. Os mesmos procedimentos utilizados na instalação do Windows Vista foram

realizados no Windows XP, no Windows 7 e no Linux (distribuição Mandriva) com êxito. Da mesma forma a última parte do processo de geração automática do *plugin* gráfico *GMFGen* ficava comprometida, não obtendo-se êxito no Windows Vista. Nos sistemas operacionais Windows XP e no Windows 7 obteve-se êxito na geração do *plugin GMFGen*.

Referências Bibliográficas

ABDUNABI, T. X-Machine Toolkit Within Eclipse Plataform. 2007. 91p. Dissertação (Mestrado em Ciência da Computação). Universidade de Sheffield.

AKYILDIZ, I. F.; SU, W.; SANKARASUBRAMANIAM, Y. E.; CYIRCY, E. (2002). Wireless sensor networks: A survey. Computer Networks, 38(4): 393-422.

APACHE (2005). Velocity Java-based Template Engine. Disponível em: http://jakarta.apache.org/velocity. Acesso em 30 ago 2011.

BAKSHI, A. B.; PRASANNA, V. K. Architecture-Independent Programming for Wireless Sensor Networks. 1ed. [S.l.]: John Wiley and Sons Ltd, 2008.

BOULET, P. Array-OL revisited, multidimensional intensive signal processing specification. França: Research Report, 2002. 27p. RR-6113, vol. 2.

BROWN, A. W.; SHORT, K. (1997). On components and Objetcs: The Foundation of Component-Based Development. International symposium on assessment of software tools and technologies (SAST 97), Pittsburgh, PA. IEEE Press. p 112-121.

BUDINSKY, F., PATERNOSTRO, M., MERKS, E. EMF: Eclipse Modeling Framework. 2ª ed. [S.l.]: Addison-Wesley Professional, 2008.

CARVALHO JR, A. J. Aplicando Model-Driven Development à Plataforma GPGPU. 2008. 51p. Monografia (Graduação em Ciência da Computação). Universidade Federal de Pernambuco.

CULLER, D.; GAY, D.; LEVIS, R. B.; WELSH, M.; BREWER, E. (2003). The nesC language: A holistic approach to networked embedded systems. In Conference on Programming Language Design and Implementation of ACM SIGPLAN.

D'SOUZA, D.; WILLS, A. (1999). Objects, Components and Frameworks with UML: The Catalysis Approach. [S.l.]: Addison-Wesley.

EMF. Eclipse Modeling Framework Project. Disponível em: <http://www.eclipse.org/modeling/emf/>. Acesso em 30 ago 2011.

ESTRIN, D.; GOVINDAN, R.; HEIDEMANN, J. S.; KUMAR, S. (1999). Scalable Coordination in Sensor Networks. In Proc. of the 5th Annual International Conference on Mobile Computing and Networks (MobiCOM 99), pp. 263-270, Seattle, Washington, USA.

FERREIRA, M. A. R. Desenvolvimento de um plug-in Eclipse para modelagem de aplicações geoespaciais. 2009. 126p. Dissertação (Mestrado em Ciência e Tecnologia Ambiental). Universidade do Vale do Itajaí.

FRANCE, R.; RUMPE, B. Model-driven development of complex software: A research roadmap. In: 29th International Conference on Software Engineering 2007 - Future of Software Engineering. Minneapolis. USA: IEEE Computer Society, 2007. p. 37–54.

GALLARDO, D; BURNETTE, E.; MCGOVERN, R. (2003). Eclipse in Action: A Guide for Web Developers, Manning. Addison Wesley, 1ª Edição. Boston.

GEF. Graphical Editing Framework. Disponível em: <http://www.eclipse.org/gef/>. Acesso em 30 ago 2011.

GPGPU. General-Purpose GPU. Disponível em: <http://www.gpgpu.org>. Acesso em 30 ago 2011.

GIMENES, I. M. S.; HUZITA, E. H. M (2005). Desenvolvimento Baseado em Componentes: Conceitos e Técnicas, Ciência Moderna, 1ª Edição. Rio de Janeiro.

GMF. Graphical Modeling Framework. Disponível em: <http://www.Eclipse.org/modeling/gmf/>. Acesso em 30 ago 2011.

HILL, J. L. System Architecture for Wireless Sensors Networks. Berkeley, 2003. 196p. Tese (Doutorado em Ciência da Computação). Universidade da Califórnia.

HOLCOMBE, M.; IPATE, F. (1998). Correct Systems: Building a Business Process Solution, Springer Verlag, London.

ILYAS, M.; MAHGOUB, I. (2004). Handbook of sensor networks: compact wireless and wired sensing systems, chapter 20. CRC Press LLC.

JavaSoft. Java Software web. Disponível em: <http://www.interhack.net/people/cmcurtin/rants/write-once-run-anywhere/write-once-run-anywhere.pdf. Acesso em 30 ago 2011.

JET. Java Emitter Templates. Disponível em: <http://www.eclipse.org/modeling/m2t/?project=jet#jet >. Acesso em 30 ago 2011.

KEFALAS, P. XMDL User Manual 1.6. Thessalouiki: Londres: Technical Report, 2000. 25p. CS 07/00, vol. 1.

KOZIKOWSKI, J. (2005). A Bird's Eye view of AndroMDA. Disponível em: <http://www.andromda.org/docs/contrib/birds-eye-view.html>. Acesso em 30 ago 2011.

LabVIEW WSN. LabView Wireless Sensor Network. Disponível em: <www.ni.com/wsn>. Acesso em 30 ago 2011.

LEVIS, P.; LEE, N. (2003). Tossim: A simulator for tinyos networks. Disponível em: <http://today.cs.berkeley.edu/tos/tinyos-1.x/doc/nido.pdf >. Acesso em 30 ago 2011.

LOUREIRO, A. A. F.; NOGUEIRA, J. M. S.; RUIZ, L. B.; MINI, R. A. F.; NAKAMURA, E. F.; FIGUEIREDO, C. M. S. (2003). Redes de sensores sem fio. Simpósio Brasileiro de Redes de Computadores – SBRC, PP. 179-226. Belo Horizonte, MG, Brasil.

LUCENA JR., V. F. Flexible Web-based Management of Components for Industrial Automation. 2002. 171p. Tese (Doutorado em Engenharia Elétrica). Faculdade de Eletrônica e Informática, Universidade de Stuttgart.

LUCRÉDIO, D. Uma Abordagem Orientada a Modelos para Reutilização de Software. 2009. 287p. Tese (Doutorado em Ciência da Computação e Matemática Computacional). Faculdade de São Carlos, Universidade de São Paulo.

LUEBKE, D., HUMPHREYS, G. (2007). How GPUs Work. IEEE Computer, vol.40, no.2, pp.96-100.

MATULA, M. (2003). Netbeans Metadata Repository. Disponível em: <http://mdr.netbeans.org >. Acesso em 30 ago 2011.

MCILROY, M. D. (1968). Mass produced software components. In: NATO Software Engineering Conference. [S.l.: s.n.]. p. 138–155.

MELLOR, S. J.; CLARK, A. N.; FUTAGAMI, T. (2003). Model-driven development. IEEE Software, v. 20, n. 5, p. 14–18.

MELLOR, S. J., SCOTT, K., UHL, A., et al. (2004). MDA Distilled Principles of Model-Driven Architecture, Addison-Wesley.

MUKERJI J.; MILLER, J. (2003). MDA Guide version 1.0.1. disponível em <http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf>. Acessado em 02 fev 2009.

NVIDIA. Compute Unified Device Architecture: Programming Guide. Disponível em: <http://www.nvidia.com/object/cuda_home_new.html>. Acessado em: 06 mar 2008.

OCL. Object Management Group: UML 2.0 OCL Specification. Disponível em: <http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf>. Acessado em: 06 ago 2010.

OMG. MDA Guide Version 1.0.1. Disponível em: <http://www.omg.org>. Acesso em 30 ago 2011.

OWENS, J. D.; LUEBKE, D.; GOVINDARAJU, N.; HARRIS, M.; KRUGER, J.; LEFOHN, A. E.; PURCELL, T. (2005). A survey of general-purpose computation on graphic hardware. In Proceedings of Eurographics 2005, State of the Art Reports. 21-51.

PDE. Plugin Development Enviroment. Disponível em: <http://www.eclipse.org/pde/>. Acesso em 30 ago 2011.

PRESSMAN, R. (2005). Software Engineering: A Practitioner's Approach, 6th Edition, McGraw Hill.

RUIZ, L. B. MANÁ: Uma Arquitetura para o Gerenciamento de Redes de Sensores Sem Fio. 2003. 214p. Tese (Doutorado em Ciência da Computação). Universidade Federal de Minas Gerais.

RUIZ, L. B.; NOGUEIRA, J. M. S.; Loureiro, A. A. (2004). Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems, volume 1, chapter III: Sensor Network Management. CRCPress.

SAMETINGER (1997), J. Software Engineering with Reusable Components. Springer Verlag. 1ª Edição. Áustria.

SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. IEEE Computer, v. 39, n. 2, p. 25–31, 2006.

SOMMERVILLE, I. (2007). Software Engineering, Addison Wesley, 8ª Edição. United States of America.

SOHRABY, K.; MINOLI, D. Wireless Sensor Networks Technology, Protocols, And Applications. Wiley, 2007.

SZYPERSKI (1999), C. Component Software: Beyond Object-Oriented Programming. [S.l.]: Addison Wesley.

TinyOS. Sistema Operacional TinyOS. Disponível em: <http://www.tinyos.net/>. Acesso em 30 ago 2011.

WALKINSHAW, N. Visual X-Machine Description Language (VXDML). 2002. 87p. Dissertação (Mestrado em Ciência da Computação). Universidade de Sheffield.

Apêndice A- Publicações

SANTOS, A., BRAGA, M. e LUCENA JR, V. Geração Automática de Código para Redes de Sensores Sem Fio Baseado em Componentes de Software. V Congresso Norte-Nordeste de Pesquisa e Inovação – Connepi 2010. ISBN 978-85-64320-00-0. Sergipe.

BRAGA, M., SANTOS, A. e LUCENA JR, V. Modelagem e Geração de Código para Redes de Sensores Sem Fio Usando X-Machine. In Proceedings of the 9th International Information and Telecommunication Technologies Symposium – I2TS 2010. ISBN 978-85-64030-00-8, Dec. 2010, Vol. 1. Rio de Janeiro.

BRAGA, M., SANTOS, A. e LUCENA JR, V. Usando Communication X-Machine na Construção de Aplicações para Redes de Sensores Sem Fio. IV Simpósio de Modelagem Computacional do Sul – 3MCSul 2010. 402p. ISSN 2179-0671. Rio Grande do Sul.

Anexo I- O arquivo XML extraído de GAC-RSSFs Tree Editor Plugin da aplicação Blink

```
<?xml version="1.0" encoding="UTF-8"?>
<component model-name="Blink">
 <component name="Main"/>
 <component name="BlinkM"/>
 <component name="SingleTimer"/>
 <component name="LedsC"/>
 <function name="post">
  <input-parameters>
   <param-input>
    <ntuple>
     <nil/>
     <ntuple-element>
      <constant>vAuxEntConst</constant>
      <variable>vAuxEnt</variable>
     </ntuple-element>
    </ntuple>
   </param-input>
  </input-parameters>
  <function-body>
   <output-parameters>
    <param-output>
     <ntuple>
      <nil/>
      <ntuple-element>
       <constant>vAuxSaiConst</constant>
       <variable>vAuxSai</variable>
      </ntuple-element>
     </ntuple>
    </param-output>
   </output-parameters>
  </function-body>
 </function>
 <interface fromComponent="Main" interfaceName="StdControl" toComponent="SingleTimer"/>
 <interface fromComponent="Main" interfaceName="StdControl" toComponent="BlinkM"/>
 <interface fromComponent="BlinkM" interfaceName="Timer" toComponent="SingleTimer"/>
 <interface fromComponent="BlinkM" interfaceName="LedsC" toComponent="LedsC"/>
 <external-function name="geral">
  <input>
   <nil/>
   <ntuple-element>
    <constant>vAuxConstExt</constant>
    <variable>vAuxExt</variable>
   </ntuple-element>
  </input>
 </external-function>
</component>
```

Anexo II- Blink

Blink

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC.Leds;
}
```

BlinkM

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
implementation {

  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 1000);
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }
  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

Anexo III- Surge

Surge

```
includes Surge;
includes SurgeCmd;
includes MultiHop;


configuration Surge {
}
implementation {
  components Main, SurgeM, TimerC, LedsC, NoLeds, Photo,
RandomLFSR,
    GenericCommPromiscuous as Comm, Bcast, MultiHopRouter as
multihopM, QueuedSend, Sounder;

  Main.StdControl -> SurgeM.StdControl;
  Main.StdControl -> Photo.StdControl;
  Main.StdControl -> Bcast.StdControl;
  Main.StdControl -> multihopM.StdControl;
  Main.StdControl -> QueuedSend.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> Comm.StdControl;
  //  multihopM.CommControl -> Comm;

  SurgeM.ADC   -> Photo.ADC;
  SurgeM.Timer -> TimerC.Timer[unique("Timer")];
  SurgeM.Leds  -> LedsC.Leds; // NoLeds;
  SurgeM.Sounder -> Sounder.Sounder;

  SurgeM.Bcast -> Bcast.Receive[AM_SURGECMDMSG];
  Bcast.ReceiveMsg[AM_SURGECMDMSG] ->
Comm.ReceiveMsg[AM_SURGECMDMSG];

  SurgeM.RouteControl -> multihopM;
  SurgeM.Send -> multihopM.Send[AM_SURGEMSG];
  multihopM.ReceiveMsg[AM_SURGEMSG] ->
Comm.ReceiveMsg[AM_SURGEMSG];
  //multihopM.ReceiveMsg[AM_MULTIHOPMSG] ->
Comm.ReceiveMsg[AM_MULTIHOPMSG];
}
```

SurgeM

```
module SurgeM {
  provides {
    interface StdControl;
  }
  uses {
    interface ADC;
    interface Timer;
    interface Leds;
    interface StdControl as Sounder;
    interface Send;
    interface Receive as Bcast;
    interface RouteControl;
  }
}

implementation {

  enum {
    TIMER_GETADC_COUNT = 1,            // Timer ticks for ADC
    TIMER_CHIRP_COUNT = 10,            // Timer on/off chirp count
  };

  bool sleeping;           // application command state
  bool focused;
  bool rebroadcast_adc_packet;

  TOS_Msg gMsgBuffer;
  norace uint16_t gSensorData;        // protected by gfSendBusy
flag
  bool gfSendBusy;


  int timer_rate;
  int timer_ticks;
  static void initialize() {
    timer_rate = INITIAL_TIMER_RATE;
    atomic gfSendBusy = FALSE;
    sleeping = FALSE;
    rebroadcast_adc_packet = FALSE;
    focused = FALSE;
  }

  task void SendData() {
    SurgeMsg *pReading;
    uint16_t Len;
    dbg(DBG_USR1, "SurgeM: Sending sensor reading\n");

    if (pReading = (SurgeMsg *)call
Send.getBuffer(&gMsgBuffer,&Len)) {
      pReading->type = SURGE_TYPE_SENSORREADING;
      pReading->parentaddr = call RouteControl.getParent();
      pReading->reading = gSensorData;
```

```
      if ((call Send.send(&gMsgBuffer,sizeof(SurgeMsg))) !=
SUCCESS)
     atomic gfSendBusy = FALSE;
    }

  }

  command result_t StdControl.init() {
    initialize();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, timer_rate);
    return SUCCESS;
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  event result_t Timer.fired() {
    dbg(DBG_USR1, "SurgeM: Timer fired\n");
    timer_ticks++;
    if (timer_ticks % TIMER_GETADC_COUNT == 0) {
      call ADC.getData();
    }
    // If we're the focused node, chirp
    if (focused && timer_ticks % TIMER_CHIRP_COUNT == 0) {
      call Sounder.start();
    }
    // If we're the focused node, chirp
    if (focused && timer_ticks % TIMER_CHIRP_COUNT == 1) {
      call Sounder.stop();
    }
    return SUCCESS;
  }

  async event result_t ADC.dataReady(uint16_t data) {
    //SurgeMsg *pReading;
    //uint16_t Len;
    dbg(DBG_USR1, "SurgeM: Got ADC reading: 0x%x\n", data);
    atomic {
      if (!gfSendBusy) {
     gfSendBusy = TRUE;
     gSensorData = data;
     post SendData();
      }
    }
    return SUCCESS;
  }

  event result_t Send.sendDone(TOS_MsgPtr pMsg, result_t success)
{
    dbg(DBG_USR2, "SurgeM: output complete 0x%x\n", success);
    //call Leds.greenToggle();
```

```
    atomic gfSendBusy = FALSE;
    return SUCCESS;
  }


  event TOS_MsgPtr Bcast.receive(TOS_MsgPtr pMsg, void* payload,
uint16_t payloadLen) {
    SurgeCmdMsg *pCmdMsg = (SurgeCmdMsg *)payload;

    dbg(DBG_USR2, "SurgeM: Bcast  type 0x%02x\n", pCmdMsg->type);

    if (pCmdMsg->type == SURGE_TYPE_SETRATE) {       // Set timer
rate
      timer_rate = pCmdMsg->args.newrate;
      dbg(DBG_USR2, "SurgeM: set rate %d\n", timer_rate);
      call Timer.stop();
      call Timer.start(TIMER_REPEAT, timer_rate);

    } else if (pCmdMsg->type == SURGE_TYPE_SLEEP) {
      // Go to sleep - ignore everything until a SURGE_TYPE_WAKEUP
      dbg(DBG_USR2, "SurgeM: sleep\n");
      sleeping = TRUE;
      call Timer.stop();
      call Leds.greenOff();
      call Leds.yellowOff();

    } else if (pCmdMsg->type == SURGE_TYPE_WAKEUP) {
      dbg(DBG_USR2, "SurgeM: wakeup\n");

      // Wake up from sleep state
      if (sleeping) {
     initialize();
        call Timer.start(TIMER_REPEAT, timer_rate);
     sleeping = FALSE;
      }

    } else if (pCmdMsg->type == SURGE_TYPE_FOCUS) {
      dbg(DBG_USR2, "SurgeM: focus %d\n", pCmdMsg-
>args.focusaddr);
      // Cause just one node to chirp and increase its sample
rate;
      // all other nodes stop sending samples (for demo)
      if (pCmdMsg->args.focusaddr == TOS_LOCAL_ADDRESS) {
     // OK, we're focusing on me
     focused = TRUE;
     call Sounder.init();
     call Timer.stop();
     call Timer.start(TIMER_REPEAT, FOCUS_TIMER_RATE);
      } else {
     // Focusing on someone else
     call Timer.stop();
     call Timer.start(TIMER_REPEAT, FOCUS_NOTME_TIMER_RATE);
      }

    } else if (pCmdMsg->type == SURGE_TYPE_UNFOCUS) {
      // Return to normal after focus command
```

```
      dbg(DBG_USR2, "SurgeM: unfocus\n");
      focused = FALSE;
      call Sounder.stop();
      call Timer.stop();
      call Timer.start(TIMER_REPEAT, timer_rate);
    }
    return pMsg;
  }

}
```

Anexo IV- Posição

Posicao

```
configuration Posicao{
}

implementation {

  components Main, PosicaoM, SingleTimer, Photo as DEMO_PosicaoM1,
Photo as DEMO_PosicaoM2, Photo as DEMO_PosicaoM3;

  Main.Location -> SingleTimer.Location;
  Main.StdControl -> PosicaoM.StdControl;
  PosicaoM.ADC -> DEMO_PosicaoM1.ADC;
  PosicaoM.ADC -> DEMO_PosicaoM2.ADC;
  PosicaoM.ADC -> DEMO_PosicaoM3.ADC;
  PosicaoM.Timer -> SingleTimer.Timer;
}
```

PosicaoM

```
module PosicaoM {
  provides {
     interface StdControl;
     interface Location;
  }
  uses {
     interface Timer;
     interface ADC as ADCX;
     interface ADC as ADCY;
     interface ADC as ADCXZ;
  }
}

implementation {

  location_3d_t cur_loc;

  command result_t StdControl.init() {
     atomic{
      cur_loc.x=0;
      cur_loc.y=0;
      cur_loc.z=0;
      }
     return SUCCESS;
  }
```

```
  command result_t StdControl.start() {
     return call Timer.start(TIMER_REPEAT, 1000);
  }

  command result_t StdControl.stop() {
     return call Timer.stop();
  }

  command result_t Location.getLocation() {
     return call ADCX.getData();
  }

 event result_t Timer.fired(){
     call Location.getLocation();
     return SUCCESS;
  }

  async event result_t ADCX.dataReady(uint16_t val){
     atomic{
       cur_loc.x = val;
       if (!call ADCY.getData()){
         signal Location.locationDone(NULL);
       }
       dbg(DBG_USR1, "Posicao x: %x\n", cur_loc.x);
     }
  }

  async event result_t ADCY.dataReady(uint16_t val){
     atomic{
       cur_loc.y = val;
       if (!call ADCZ.getData()){
         signal Location.locationDone(NULL);
       }
       dbg(DBG_USR1, "Posicao y: %x\n", cur_loc.y);
     }
  }

  async event result_t ADCZ.dataReady(uint16_t val){
     atomic{
       cur_loc.z = val;
       signal Location.locationDone(&cur_loc);
       dbg(DBG_USR1, "Posicao z: %x\n", cur_loc.z);
     }
           return SUCCESS;
  }
}
```

Anexo V- PosiçãoM

PosicaoM

```
module PosicaoM {
  provides {
     interface StdControl;
     interface Location;
  }
  uses {
     interface Timer;
     interface ADC as ADCX;
     interface ADC as ADCY;
     interface ADC as ADCXZ;
  }
}

implementation {

  // Declarar variáveis

  static void inicializacao(){
     // Inicializar variáveis
  }

  command result_t StdControl.init() {
     inicializacao();
     return SUCCESS;
  }

  command result_t StdControl.start() {
     return call Timer.start(TIMER_REPEAT, 1000);
  }

  command result_t StdControl.stop() {
     return call Timer.stop();
  }

  task void processing() {
     // Tarefas

  }

  event result_t Timer.fired(){
     // Tratar o evento

     post processing();
```

```
      return SUCCESS;
   }
}
```