UNIVERSIDADE FEDERAL DO AMAZONAS

FACULDADE DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

GERAÇÃO AUTOMÁTICA DE CÓDIGO PARA REDES DE SENSORES SEM FIO USANDO COMMUNICATING X-MACHINE

MARCUS DE LIMA BRAGA

MANAUS - AM

2012

UNIVERSIDADE FEDERAL DO AMAZONAS

FACULDADE DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

MARCUS DE LIMA BRAGA

GERAÇÃO AUTOMÁTICA DE CÓDIGO PARA REDES DE SENSORES SEM FIO USANDO COMMUNICATING X-MACHINE

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

Orientador:

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

MANAUS - AM

2012

MARCUS DE LIMA BRAGA

GERAÇÃO AUTOMÁTICA DE CÓDIGO PARA REDES DE SENSORES SEM FIO USANDO COMMUNICATING X-MACHINE

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

Aprovado em 5 de janeiro de 2012.

BANCA EXAMINADORA

Prof. Dr. -Ing. Vicente Ferreira de Lucena Junior
Universidade Federal do Amazonas

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

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

Ficha Catalográfica

(Catalogação realizada pela Biblioteca Central da UFAM)

C813g Braga, Marcus de Lima

Geração Automática de Código para Redes de Sensores Sem Fio Usando Communicating X-Machine / Marcus de Lima Braga. - Manaus: UFAM, 2012.

138 p.;il. color.;

Dissertação (Mestrado em Engenharia Elétrica) – Universidade Federal do Amazonas, 2012.
Orientador: Prof. Dr. Vicente Ferreira de Lucena Junior

1. Sistemas de comunicação sem fios 2. Sistemas de transmissão de dados 3. Redes de computação I. Lucena Júnior, Vicente Ferreira de (Orient.) II. Universidade Federal do Amazonas III. Título

CDU (1997): 004.7(043.3)

Dedico este trabalho acima de tudo a Deus por tudo que Ele é, e a Nossa Senhora do Perpétuo Socorro por seu amor e consolo. Aos meus pais, in memoriam, Cleomir e Marizete Braga. À minha querida esposa Kellem Andrezza. Aos meus queridos irmãos Cleomir e Helen. As minhas tias-mães Elcy e Vânia, ao meu tio Nonato. À minha sogra Helena. Ao meu orientador Professor Vicente Lucena. A todos os amigos que contribuíram direta e indiretamente para esta pesquisa.

Agradecimentos

"Honra o Senhor com teus haveres, e com as primícias de todas as tuas colheitas"(Prov 3, 9).

A Deus por tudo que é, por nos dar a Jesus, por enviar seu Santo Espírito e por nos ter dado Maria Santíssima como mãe.

Aos meus pais, *in memorian*, Cleomir e Marizete Braga por todo amor, zelo, pelas orações, por seus exemplos de vida e dignidade que são referências para mim e, sobretudo pela honra de ser seu filho.

A minha esposa Kellem Andrezza pelo amor, companheirismo, tolerância, consolo nos momentos difíceis e incentivos em toda caminhada.

Aos meus queridos irmãos e padrinhos Cleomir e Helen, pelo amor, amizade e por serem verdadeiros irmãos e amigos, muito obrigado por tudo.

As minhas tias-mães Elcy e Vânia, pelo incentivo, amor, generosidade e consolo nos momentos difíceis.

Ao meu orientador Professor Vicente Lucena, pela oportunidade proporcionada, pelas valiosas orientações no desenvolvimento deste trabalho e no seu exemplo de pesquisador, que tanto nos motiva no desenvolvimento da ciência.

Ao meu grande amigo, irmão e colega de mestrado Alyson de Jesus dos Santos, meus sinceros agradecimentos por todos os momentos de companheirismo e amizade, pelas contribuições relevantes para o desenvolvimento deste trabalho, e pela força nos momentos mais difíceis da minha vida, o meu muito obrigado.

Aos meus colegas de mestrado da Engenharia Elétrica: Lincoln, Pedro Ivan, Luís Filipe, Fábio, Wilison, Rubens e Lucélia pela ajuda nos estudo das disciplinas.

Aos meus amigos do DCC/UFAM Daniel Patrick e Roger pelos momentos de valiosa aprendizagem e no desenvolvimento de pesquisas que contribuíram para meu aperfeiçoamento.

A Universidade Federal do Amazonas, ao CETELI e todos os professores do Curso de Mestrado de Engenharia Elétrica por suas valiosas contribuições.

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

Aos amigos do Tribunal de Justiça: Rodrigo Choji, Ricardo Maia, Breno Corado, Roberto Rocha e Ailton Cortez, por suas contribuições e incentivos.

Nada te perturbe,
Nada te espante,
Tudo passa,
Deus não muda,
A paciência tudo alcança;
Quem a Deus tem Nada lhe falta: Só Deus basta.
Eleva o pensamento,
Ao céu sobe,
Por nada te angusties,
Nada te perturbe.
A Jesus Cristo segue Com peito grande,
E, venha o que vier,
Nada te espante.
Vês a glória do mundo?
É glória vã;
Nada tem de estável, Tudo passa.
Aspira às coisas celestes,
Que sempre duram;
Fiel e rico em promessas, Deus não muda.
Ama-O como merece,
Bondade imensa;
Mas não há amor fino Sem a paciência.
Confiança e fé viva Mantenha a alma,
Que quem crê e espera Tudo alcança.
Do inferno acossado Muito embora se veja,
Burlará os seus furores Quem a Deus tem.
Advenham-lhe desamparos, Cruzes, desgraças;
Sendo Deus o seu tesouro, Nada lhe falta.
Ide, pois, bens do mundo,
Ide, ditas vãs;
Ainda que tudo perca,
Só Deus basta.
Santa Teresa D'Ávila

Resumo

Rede de Sensores Sem Fio (RSSF) tem despertado o interesse de pesquisadores no desenvolvimento de aplicações que atuem no monitoramento e controle de fenômenos físicos, apoiando-se em sua autonomia e flexibilidade, e em sua natureza distribuída e pervasiva. Se por um lado, esse interesse proporciona o desenvolvimento de novas aplicações, por outro, eleva sua complexidade e consequentemente seus custos. Isto é facilmente entendido devido suas características restritivas, que fazem com que os estágios cruciais do desenvolvimento de software (modelagem, verificação e testes) se tornem tarefas não triviais. A programação é realizada muito próxima ao sistema operacional (baixo nível), favorecendo a distração na aplicação da lógica e exigindo, portanto, ao desenvolvedor maior concentração, além de formação técnica rara entre os especialistas. Este cenário propicia a criação de metodologias e ferramentas que deem suporte ao desenvolvimento nesta plataforma. Este trabalho tem por objetivo utilizar o método formal Communicating X-Machine na construção de aplicações RSSF, oferecendo uma abordagem intuitiva e um desenvolvimento modular, provendo ao programador elevação no nível de abstração, facilitando a construção de aplicações até a geração de código na linguagem de programação nesC (plataforma TinyOS). Portanto, o presente trabalho pretende facilitar o desenvolvimento de aplicações, desde sua modelagem até a geração de código, proporcionando ao desenvolvedor o aumento no nível de abstração, melhor entendimento do problema através de seu particionamento e facilidade na construção de aplicações em RSSF.

Palavras-chave: Redes de Sensores Sem Fio, Métodos Formais, Communicating X-Machine, nesC.

Abstract

Wireless Sensor Network (WSN) has attracted the interest of researchers in applications development that operate on monitoring and control of physical phenomena, relying on its autonomy and flexibility, and on its distributed nature and pervasive. If for one hand, this interest provides the development of new applications, on the other it increases its complexity and therefore costs. This is easily understood due their restrictive features, which make the crucial stages of software development (modeling, verification and testing) become a non-trivial task. The programming is carried out very close to the operating system (low level), favoring the distraction in the application of logic and thus requiring more concentration of the developer, as well as technical training, rare among experts. This scenario enables the creation of methodologies and tools that support development on this platform. This study aims to use the formal method called Communicating X-Machine in the construction of WSN applications, offering an intuitive approach and a modular development, providing to the programmer a higher level of abstraction, making it easier to build applications until the code generation in nesC language programming (TinyOS platform). So, this paper aims to facilitate the development of applications, from their modeling until the code generation, providing to the developer an increase in the level of abstraction, a better understanding of the problem through its partitioning and ease of building WSN applications.

Keywords: Wireless Sensor Network, Formal Method, Communicating X-Machine, nesC.

Lista de Siglas

AFN Autômato Finito Não-Deteminístico

API Application Programming Interface

APD Autômato com Pilha Determinístico

APN Autômato com Pilha Não-Determinístico

AWT Abstract Windows Toolkit

CXM Communicating X-Machine

EMF Eclipse Modeling Framework

GEF Graphical Editing Framework

GME Generic Modeling Environment

GMF Graphical Modeling Framework

GMT Generative Modeling Tools

IDE Integrated Development Environment

JET Java Emitter Templates

JMI Java Metadata Interface

JNI Java Native Interface

JSP Java Server Pages

MDD Model-Driven Development

MDA Model-Driven Architecture

MDE Model-Driven Engineering

MDR MetaData Repository

MDSD Model-Driven Software Development

MEMS Micro-Electro-Mechanical System

MVC Model-View-Controller

OMG Object Management Group

RSCH Rede de Sensores do Corpo Humano

RSSF Rede de Sensores Sem Fio

SWT Standard Widget Toolkit

SXM Stream X-Machine

UML Unified Modeling Language

XM X-Machine

XMI XML Metadata Interchange

XML Extensible Markup Language

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Sumário

Capítulo 1
Introdução

Redes de Sensores Sem Fio (RSSF) são compostas por uma grande quantidade de dispositivos autônomos denominados de nós sensores, com capacidade de processamento, sensoriamento e comunicação sem fio de múltiplos saltos e que trabalham de modo cooperativo (AKYILDIZ *et al.*, 2002; ROSSETO, 2006; NAKAMURA, 2007; MOTTOLA, 2008). Essa tecnologia apresenta características restritivas em relação ao processamento, armazenamento, energia e comunicação (MOTTOLA, 2008; BAKSHI; PRASANNA, 2008). As RSSF, apesar destas limitações possuem um grande potencial para aplicações de monitoramento e controle de fenômenos físicos, onde os nós sensores cooperam entre si para percepção de um fenômeno ou evento estudado. Sua comunicação com a estação de monitoramento é feita por meio de um nó sensor diferenciado denominado como nó sorvedouro (*sink node*). Este tem por função transmitir os dados coletados na região estudada via comunicação mais robusta (por exemplo, satélite) (OLIVEIRA, 2008). A Figura 1.1 ilustra este funcionamento. Normalmente, as RSSF são implantadas em ambientes de difícil acesso, inóspitos na maioria das vezes, o que pode favorecer a imprecisão e redução de cobertura em seu sensoriamento e dificultar suas comunicações, bem como podendo vir a ser inutilizado devido à falta de energia ou ainda por defeitos de fabricação (NAKAMURA, 2007).

A evolução tecnológica proporcionada pela microeletrônica e MEMS[1] (*Micro-Electro-Mechanical System*) favorece o barateamento do hardware, além de prover vários tipos de sensoriamento, tornando viável a construção de aplicações de diferentes dimensões utilizando diversos tipos de sensores. Todavia, este cenário implica no aumento do nível de complexidade das aplicações e, consequentemente, de sua programação, fazendo com que o programador tenha que trabalhar próximo ao sistema operacional (baixo nível), obrigando-o a ter foco não somente na lógica da aplicação, mas também exigindo que possua um conhecimento técnico da plataforma raramente encontrada entre especialistas do domínio da aplicação (MOTTOLA; PICCO, 2011).

[1]MEMS tecnologia que integra elementos mecânicos, eletrônicos e sensores em um chip programado.

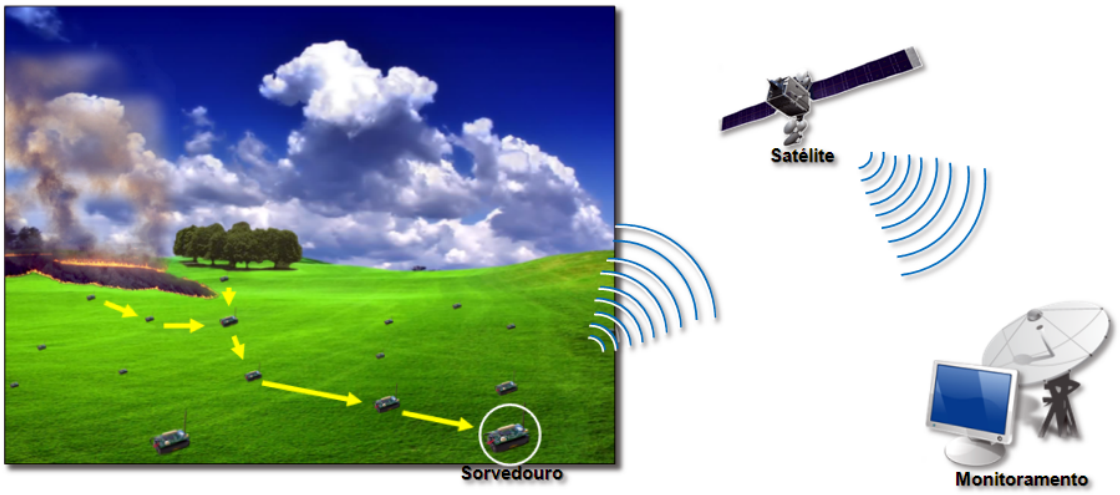


Figura 1.1: RSSF: cenário de detecção de eventos, onde o nó sorvedouro é o responsável por enviar as informações coletadas à estação de monitoramento

Terfloth (2009) relata que o desenvolvimento de aplicações para RSSF é complexo e tedioso, tendendo a erros devido à escassez de recursos, restrições de tempo e a um modelo de comunicação assíncrona, inerente a dispositivos embarcados e que estão expostos ao programador, fazendo com que o mesmo seja forçado a tomar uma perspectiva orientada ao sistema em vez de orientado ao problema, tornando esta circunstância uma desvantagem para especialistas do domínio (futuros utilizadores) que não são desenvolvedores de software.

A comunidade de pesquisadores em RSSF tem demonstrado uma crescente conscientização a respeito deste problema e um crescente número de abordagens têm sido propostas. Martins *et al.* (2008) sugere que uma plataforma ideal para RSSF permitiria escrever uma aplicação em uma linguagem de alto nível, depurar o código em um computador pessoal e implantá-lo automaticamente em um grande número de dispositivos sensores dotados de comunicação sem fio. Todavia, mesmo fornecendo um conjunto amplo e diversificado de funcionalidades, tais propostas não conseguem satisfazer todas as características e exigências das aplicações de RSSF, pois é necessário o claro entendimento das necessidades da aplicação e das diferenças básicas entre as abordagens de programação para a escolha da melhor plataforma.

Uma estratégia para solucionar este problema, é o aumento no nível de abstração com intuito de simplificar a programação, sem sacrificar a eficiência. Um método formal possibilita o aumento nos níveis de abstração, possibilitando ao desenvolvedor a liberdade de se concentrar no que realmente importa, não se detendo em detalhes irrelevantes ao problema.

O termo método formal remete ao uso de técnicas matemáticas na concepção e análise de software e hardware, podendo usar linguagens de especificação formal para descrever o comportamento de um sistema, além de utilizar técnicas matemáticas de análise para demonstrar que o sistema satisfaz propriedades críticas (CHECHIK; GANNON, 2001; CLARKE; WING, 1996). A utilização de um método formal no processo de desenvolvimento de software, permite a descrição de sistemas complexos a partir de entidades abstratas (independentes de implementação), construção por refinamento e verificação de sistemas com o objetivo de obter sistemas mais confiáveis apesar da complexidade. Métodos formais não garantem corretude aos sistemas, porém aumentam a compreensão de um sistema revelando ambigüidades e inconsistências que poderiam passar despercebidos (CLARKE; WING, 1996).

Esta pesquisa tem como cerne a criação de uma ferramenta que utilize um método formal na criação de aplicações em RSSF, desde sua modelagem até a geração de código. O método formal oferece uma maneira intuitiva na criação de modelos de sistemas, proporcionado pelo aumento nos níveis de abstração, favorecendo ao desenvolvedor a incumbência da lógica da aplicação, sem deter-se aos detalhes que não sejam relevantes ao sistema, além de proporcionar refinamentos com intuito de diminuir ambigüidades. A possibilidade do uso da técnica de verificação formal, a facilidade de se modelar sistemas complexos através de seu particionamento e a geração de código ao fim do processo, são pontos que merecem destaque na utilização do método proposto.

1.1 Motivação

A crescente diversidade de aplicações em RSSF tem favorecido proporcionalmente o aumento de complexidade nas mesmas, outro componente adicionado a esta relação, envolve a heterogeneidade dos nós sensores, bem como suas características restritivas e sua natureza distribuída. Outro agravante é o fato de sua programação ser muito próxima ao sistema operacional (baixo nível), exigindo conhecimento técnico do desenvolvedor. Facilitar a programação é um importante desafio para tornar mais viável as RSSFs, tornando as aplicações mais flexíveis e de fácil implantação. Contudo, fica evidente a necessidade de criação de ferramentas, métodos e/ou metodologias que dêem suporte ao desenvolvedor, com intuito de atingir os objetivos impostos pela tecnologia (RöMER; MATTERN, 2004).

Atualmente, existem diversas propostas de modelos de programação com objetivo de ajudar o desenvolvimento de aplicações RSSFs, bem como linguagens de programação

que proporcionam visões da aplicação, fazendo com que o programador se concentre no comportamento da rede como um todo ao invés de focar no comportamento de um simples nó. No entanto, tais soluções não suportam a heterogeneidade necessária a estas redes (KALLOE, 2008).

Uma estratégia para facilitar a programação é o aumento no nível de abstração, através de um método formal com intuito de proporcionar ao desenvolvedor a identificação de possíveis erros em tempos mais curtos (fase de projeto), bem como a eliminação de ambigüidades. Essencialmente a complexidade tende a ser reduzida pelo aumento da abstração, pois deixa de lado detalhes irrelevantes e enfatiza aspectos importantes do sistema investigado. Especificações formais têm sido o foco de pesquisas em engenharia de software há muitos anos e são utilizadas em várias aplicações, porém seu uso na área industrial ainda é limitado, apesar das perspectivas de crescimento (LAMSWEERDE, 2000) (Lamsweerd, 2000). Como métodos formais fornecem notações e técnicas, estes atributos favorecem a construção de modelos matemáticos capazes de sofrerem verificação de propriedades que o sistema precisa satisfazer. Uma propriedade pode ser qualitativa, se o interesse for o comportamento do sistema (exemplo, sistema sem *deadlocks*), ou pode ser quantitativa, caso interesse seja o desempenho do sistema (exemplo, *throughput* desejado) (MAN *et al.*, 2009).

Modelar um sistema RSSF através de um método formal permite um nível de refinamento de modo a conter todas as características de interesse e proporcionar apoio ao desenvolvedor a partir da definição do projeto, com intuito de alcançar bons resultados como, corretude, desempenho, eficiência e custo da aplicação (MAN *et al.*, 2009). A utilização de um método formal, em princípio, não garante um sistema correto, mas pode aumentar muito o entendimento do mesmo através da revelação de inconsistências, ambigüidades e informações incompletas que poderiam passar despercebidas se tais métodos não retornem nada (CLARKE; WING, 1996).

Embora métodos e metodologias de engenharia de software tenham sido concebidos para lidar com o desenvolvimento de sistemas complexos, não há evidências que qualquer um deles, além dos métodos formais, produzam sistemas corretos, livres de erro. A especificação de software usando métodos formais, ainda não foi explorada de modo a facilitar todas as fases de desenvolvimento de software: especificação formal para análise de requisitos, refinamento para concepção, síntese para codificação, prototipagem para validação e prova para verificação (KEFALAS; ELEFTHERAKIS; KEHRIS, 2003b, 2003a; FIGUEIREDO *et al.*, 2002).

A motivação principal deste trabalho é a utilização do método formal Communicating X-Machines (CXM) na criação, modelagem e geração de código para aplicações RSSF. O método utiliza estruturas matemáticas como tuplas, conjuntos, relações e funções para descrever o comportamento do sistema, com intuito de proporcionar uma abordagem intuitiva que facilite a construção de sistemas complexos de forma modular, técnicas de verificação formal favorecem o desenvolvimento de um sistema correto com aumento nos níveis de previsibilidade e diminuição de probabilidade a falhas.

1.2 Objetivos e Contribuições

Este trabalho apresenta uma ferramenta para criação de aplicações para RSSF baseada no método formal CXM, visa ainda, facilitar o desenvolvimento de aplicações complexas por meio da decomposição do problema e conseqüentemente facilitando seu entendimento. O método formal utilizado oferece ao desenvolvedor o aumento no nível de abstração de modo a facilitar a construção de aplicações para RSSF. O trabalho foi desenvolvido em duas partes, sendo que a primeira busca validar a construção das aplicações em CXM através da criação de modelos até a geração de código (*nesC* da plataforma *TinyOS*, para este trabalho). A segunda parte refere-se à automatização do processo, a criação de um *plug-in* para a plataforma Eclipse, possibilitando a geração de código para o nó sensor.

Objetivo Geral: Construir uma ferramenta para auxiliar a criação de aplicações para RSSF que utilize o método formal CXM na modelagem e geração de código utilizando máquinas de estados que proveem serviços e usem funções de acordo com o modelo estabelecido.

Objetivos Específicos:

- Investigar frameworks e geradores de código existentes para as RSSF, com o intuito de identificar princípios, tecnologias aplicadas, metodologias de desenvolvimento, bem como nível de abstração.
- Elaborar um método de tradução das especificações modeladas no formalismo, que estarão expressas na linguagem *X-Machine Designe Language* (XMDL) para *eXtensible Markup Language XML* (XML). Em seguida, converter o código XML para *nesC*.
- Estudar o framework *Graphical Model Framework* (GMF), para construir um editor

gráfico na plataforma Eclipse, proporcionando um ambiente de desenvolvimento mais amigável ao usuário.

- Integrar o método de tradução formal através de um *plug-in* na plataforma Eclipse, possibilitando a geração automática de código para o nó sensor.

São esperadas as seguintes contribuições para este trabalho:

- Levantamento bibliográfico na área RSSF, bem como análise de *frameworks* e geradores de código atualmente disponíveis;
- Aplicação de um método formal CXM para modelagem e desenvolvimento de aplicações para RSSF independentes da arquitetura.
- Proposição de um procedimento de tradução de especificação formal expresso em XMDL para XML. E, conseqüentemente, tradução XML para a linguagem de programação dos sensores, *nesC*.
- Criação de uma ferramenta que automatize a metodologia proposta, auxiliando desde a modelagem do sistema até a geração de código para o nó sensor.

1.3 Metodologia

A metodologia para o desenvolvimento do presente trabalho consistiu em cumprir etapas especificadas anteriormente, de modo a atingir cada objetivo específico descrito na Seção 1.2. Inicialmente foram realizadas pesquisas em diversas fontes literárias relacionadas às tecnologias e a modelos de programação relacionadas à RSSF, modelagem e especificação de software usando métodos formais e desenvolvimento orientado a modelos, com intuito de verificar o estado da arte sobre os referidos temas e, por conseguinte, adquirir embasamento teórico para o desenvolvimento do trabalho.

Em seguida, foi concebida uma solução para construção de aplicações RSSF usando uma abordagem intuitiva proporcionando um desenvolvimento modular. Para isto, foram criadas modelagens CXM que contemplam a funcionalidade da aplicação proposta, expressas em máquina de estado (notação diagramática) e em linguagem descritiva própria do método formal. Na etapa seguinte, foi definido um método de conversão da linguagem descritiva do autômato para uma linguagem intermediária (XML), que, por sua natureza permite sua conversão para outras linguagens de programação. Nesta etapa

foram propostos casos de uso para validar a solução proposta, os quais foram gerados suas respectivas modelagens expressas em autômatos que se comunicam e em linguagem descritiva.

Após isto, desenvolveu-se um *toolkit* denominado *Ufam Sensor CommX*, automatizando o processo de desenvolvimento de aplicações em RSSF usando o método formal CXM, dando suporte ao usuário desde a modelagem até a geração de código em *nesC*.

Posteriormente, o *plug-in* foi validado através da implementação dos casos de uso propostos anteriormente, como medir a temperatura e umidade, além do funcionamento de um *led* no sensor. Por fim, foram elaborados artigos e esta dissertação contendo descrição e os resultados alcançados.

1.4 Organização da Dissertação

Esta dissertação está dividida em sete capítulos, além do capítulo supracitado. No Capítulo 2 são abordados alguns conceitos pertinentes às áreas de RSSF, Métodos Formais e Desenvolvimento Orientado a Modelos, elementos de fundamental importância para compreensão e aplicação dos resultados obtidos.

No Capítulo 3 são descritos trabalhos relacionados a esta pesquisa, apresentando frameworks que dão suporte exclusivamente para o *TinyOS* e outros que utilizam o método formal para criação de aplicações. Análises dos trabalhos, bem como características desejadas para uma ferramenta de suporte à programação serão discutidas.

Com base nas referências bibliográficas será apresentado um método de modelagem e geração de código para RSSF no Capítulo 4, isto é feito através de um desenvolvimento modular descrito em três etapas, (1) modelagem em componentes expressos no método formal; (2) comunicação entre os componentes; e (3) geração de código através de tradução da linguagem do autômato. Serão ainda apresentados casos de uso para validar o método proposto.

Em seguida serão descritos os plug-ins, o ambiente de desenvolvimento utilizado e as etapas de construção do *toolkit* de modelagem e geração de código *Ufam Sensor CommX*. Serão apresentados ainda, a arquitetura e os componentes do plug-in proposto para automação do processo de construção de aplicações RSSF no Capítulo 5.

O processo de validação do toolkit é constituído por três cenários que utilizam da ferramenta para construção de aplicações e os resultados alcançados ao final dessas

validações serão detalhadas no capítulo 6.

Por fim, no Capítulo 7 serão apresentadas as principais conclusões sobre os estudos realizados e sobre os resultados obtidos no trabalho, além de possíveis melhorias, dificuldades encontradas e algumas sugestões para trabalhos futuros.

Capítulo 2
Fundamentação Teórica

Este capítulo apresenta um arcabouço teórico para análise dos temas relevantes à pesquisa desenvolvida, Redes de Sensores Sem Fio (RSSF), Métodos Formais e Desenvolvimento Orientado a Modelos (MDD). Inicialmente, será apresentada RSSF com sua arquitetura, *hardware*, aplicações e modelos de programação. Para o segundo tema, serão abordados modelos computacionais e suas fundamentações matemáticas, modelagens, verificações e, por conseguinte, será apresentado o método proposto. Por fim, MDD é especificado através de seus atributos e funcionalidades, padrões e transformações sobre modelos.

2.1 Redes de Sensores Sem Fio

O objetivo deste trabalho refere-se à construção de uma ferramenta baseada em um método formal para auxiliar o desenvolvedor na criação de aplicações RSSF, desde sua especificação até a geração de código. No entanto, para atingir esta meta, é necessário conhecer as técnicas de programação atualmente utilizadas e o tipo de *software* empregado, visto que ambos servem de referência para a escolha do *hardware* da aplicação modelada.

Deste modo, a fim de proporcionar melhor compreensão dos requisitos, características e restrições impostas pelas RSSF serão apresentadas peculiaridades da área desde sua arquitetura até os modelos de programação atualmente utilizados.

As RSSFs têm despertado o interesse de muitos pesquisadores devido sua ampla gama de aplicações. Todavia, vale ressaltar que as características exatas de uma RSSF estão diretamente ligadas ao contexto da aplicação. Segundo Kasten (2007), existem algumas características que são compartilhadas pela maioria das RSSFs, como, (a) monitoramento de fenômenos físicos por meio de sensores de amostragem de nós individuais, (b) sensores são implantados próximos ao fenômeno, (c) após sua implantação funcionam sem interferência humana e (d) a comunicação entre os nós sensores é sem fio. Tais carac-

terísticas gerais somadas às características específicas da aplicação servem de base para a especificação dos requisitos técnicos de *hardware* e *software* a serem empregados em um projeto de RSSF.

Atualmente, o que tem sido feito para o desenvolvimento de *software* para RSSF são adaptações de modelos do desenvolvimento clássico. Uma estratégia proposta para facilitar o desenvolvimento de *software* RSSF é aumentar o nível de abstração para o desenvolvedor, com intuito de proporcionar modelos de programação que facilitem o projeto de novas aplicações, alicerçados em sistemas operacionais que possibilitem apoio a estes novos modelos de programação e um *hardware* inovador dos nós sensores (KASTEN, 2007).

Como alguns requisitos de *software* são derivados diretamente das características da aplicação e do *hardware* dos nós sensores, será apresentado um estudo sobre a arquitetura dos nós sensores (*hardware* e *software*), suas plataformas, tipos de aplicações e por fim serão descritos os modelos de programação utilizados e suas características.

2.1.1 Arquitetura RSSF

Segundo Mottola (2008), a arquitetura RSSF proporciona que as fronteiras entre abstrações de programação e o *software* em execução em um nó sensor sejam difícieis, favorecendo a interligação com os níveis de serviço do sistema (por exemplo, roteamento) e aplicativos, *hardware* e sistema operacional construído em cima das abstrações.

O sistema operacional limita-se a fornecer mecanismos básicos, como *scheduling* e fina camada de abstração de *hardware*, enquanto que a aplicação específica compreende apenas as camadas de MAC e roteamento (MOTTOLA, 2008). A Figura 2.1, ilustra esta arquitetura RSSF, a descrição de cada um dos componentes será feita a seguir estabelecendo seu contexto nas abordagens de programação.

Camada de Aplicação - permite que um aplicativo forneça dados que o usuário final pode utilizá-los diretamente, diferindo dos serviços do sistema que são mecanismos que não proveem qualquer informação útil por si só, porém são necessários para dar suporte a aplicações específicas.

Camada de Abstração de Programação - camada que permite ao desenvolvedor expressar em alto nível suas construções em diversas formas de processamento, a partir do próprio aplicativo até os serviços dos sistemas, como por exemplo, os protocolos de roteamento.

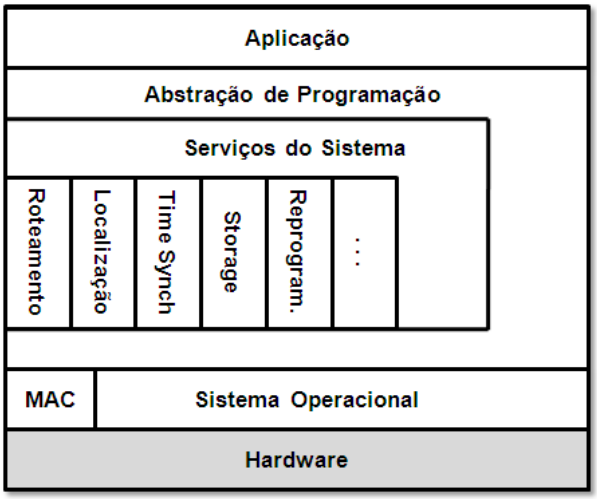


Figura 2.1: Arquitetura de referência RSSF em camadas (MOTTOLA; PICCO, 2011).

Camada de Serviços do Sistema - serviços do sistema são úteis em suporte de aplicações enquanto que a camada de aplicação entrega dados úteis diretamente ao usuário final, podemos citar como exemplos: a) Mecanismos de localização; b) Tempo de sincronização de protocolos; c) Serviços de armazenamento distribuído; d) Reprogramação e d) Protocolos de roteamento. Serviços do sistema são construídos em cima da funcionalidade fornecida pelo sistema operacional, usando sua linguagem (*nesC*, por exemplo) ou abstrações de programação.

Camada Media Access Control (MAC) - deve garantir eficientemente a comunicação enquanto gerencia a energia do nó sensor. Sendo que este último objetivo é conseguido programando o rádio para o modo de baixo consumo de energia. Outra importante característica em relação a outras plataformas sem fio é que geralmente a funcionalidade MAC é realizada em *hardware* e aqui é implementada na maior parte em *software*, utilizando linguagem de baixo nível associado ao sistema operacional.

Sistema Operacional (RSSF) - tem por funcionalidade básica esconder da aplicação detalhes de uso e acesso aos componentes de *hardware* do nó sensor, proporcionando uma interface clara aos usuários (BRANCO; RODRIGUEZ, 2010).

Camada de *Hardware* - A Figura 2.2 ilustra a visão típica de *hardware* de um nó sensor. O *hardware* do nó sensor é composto por (a) memória volátil, utilizada para guardar dados em tempo de execução de um programa (2 KB até 512 KB); (b) memória de programa, utilizada para armazenamento do código do programa, cujo tamanho varia de 32 KB a 128 KB; (c) memória de armazenamento externo, geralmente memória flash cujo tamanho pode variar de 128 KB até GB (dependendo da especificação); (d) trans-

ceiver (rádio) trabalhando geralmente na banda ISM 2,4 GHz; (e) CPU, a maioria das plataformas utiliza processador de 16 bits MSP430, da Texas Instruments ou processador de 8/16 bits da Atmega Atmel, com exceção nas plataformas IMote 2 e SunSPOT que utilizam os processadores da Intel PXA e ARM920T; e, (f) sensores e/ou atuadores. Esses recursos que compõe o *hardware* do nó sensor são levados em consideração na escolha da plataforma através dos critérios de processamento, comunicação e interação com o meio ambiente.

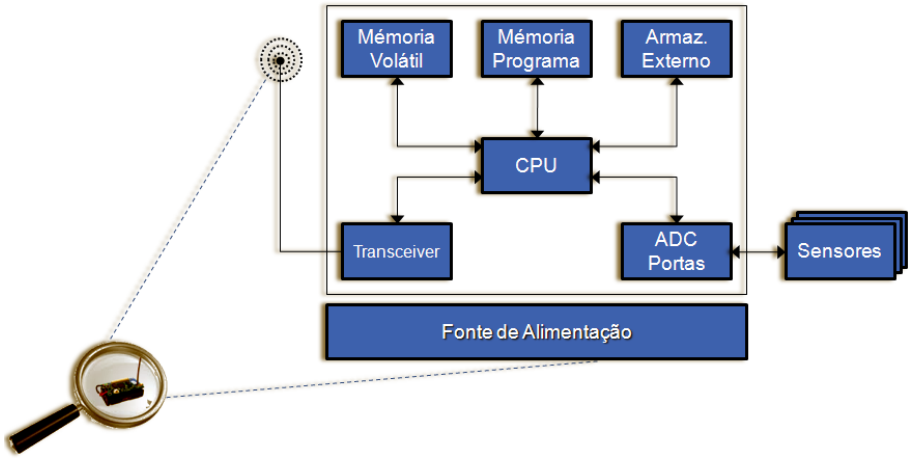


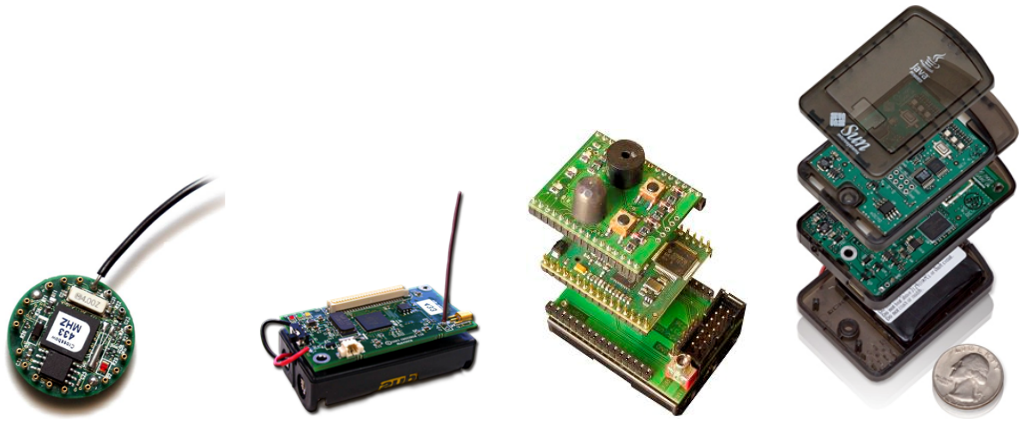
Figura 2.2: Diagrama esquemático de *hardware* do nó sensor.

2.1.2 Hardware

O *hardware* do nó sensor consiste tipicamente de uma CPU com uma memória flash para carregar o sistema operacional e o programa executável e uma memória RAM para armazenar os dados em tempo de execução. Sua fonte de energia é geralmente pilhas convencionais. Esses dispositivos são dotados de sensores que coletam dados ambientais como: temperatura, luz, umidade, entre outros e comunicam entre si através de um transceptor anexado ao *hardware*, formando uma rede composta de centenas a milhares de nós sensores. Existem diversos nós sensores disponíveis comercialmente. A Figura 2.3 apresenta alguns exemplos, (a) Mica2Dot, (b) MicaZ e (c) MSB430 utilizam a plataforma TinyOS, enquanto que (d) Sun SPOT é baseado em Java.

Hill *et al.* (2004) propôs uma classificação para as plataformas de desenvolvimento de RSSFs e as agrupou em quatro grupos baseados no *hardware* utilizado e no uso da energia. No entanto, Evers (2010), renomeou-os e os atualizou conforme apresentado na Tabela 2.1. São eles chips únicos, baixa potência, alta velocidade e outros dispositivos.

- Chip único - é utilizado para tarefas simples, esta classe é feita sob medida para



(a) Mica2Dot (b) MicaZ (c) MSB430 (d) Sun SPOT

Figura 2.3: Sensores disponíveis comercialmente.

proporcionar a utilização de pouca energia devido aos limitados recursos computacionais. Trata-se de um único circuito integrado que comporta: computação, comunicação e sensoriamento. Como o *hardware* (memória RAM e CPU) é muito limitado e seu rádio é de baixa potência, sua comunicação é somente com dispositivos mais poderosos.

- Baixa potência - são plataformas de baixo custo, pois são desenvolvidos a partir de componentes simples e baratos e fornecem recursos computacionais suficientes para realizar cálculos sobre os dados coletados e criar redes ad-hoc, além de poder carregar um pouco mais de energia.

- Alta velocidade - esta classe se baseia em dispositivos mais rápidos (processadores de 32 bits) que consomem mais energia e, por conseguinte mais caros. Estes dispositivos são indicados onde as tarefas necessitem de mais recursos computacionais e onde o fator custo não é um limitador.

Tabela 2.1: Plataformas de desenvolvimento RSSF (EVERS, 2010).

Mote	CPU	Memória	Rádio	Alimentação	Vida (hrs)
Chip Único					
Spec (2003)	8 bits Custom ISA 4-8 MHz	3 KB RAM	On-chip 916 MHz 50-100 kbps	3 mA ação 3 μA *idle*	1933
Baixa Potência					
WeC (1998)	8 bits Atmel AVR 4 MHz	0,5 KB RAM 8 KB Flash 32 KB ext. Flash	TR1000 916 MHz 10 kbps	18.8 mA ação 45.7 μA *idle*	131
Mica (2001)	8 bits Atmel AVR 4 MHz	4 KB RAM 128 KB Flash 512 KB ext. Flash	TR1000 916 MHz 40 kbps	18.8 mA ação 35.7 μA *idle*	166
Mica2 (2002)	8 bits Atmel AVR 7.37 MHz	4 KB RAM 128 KB Flash 512 KB ext. Flash	CC1000 916 MHz 38.4 kbps	26.63 mA ação 15.86 μA *idle*	354
MicaZ (2004)	8 bits Atmel AVR 7.37 MHz	8 KB RAM 128 KB Flash 512 KB ext. Flash	CC2420 802.15.4 250 kbps	29.83 mA ação 16 μA *idle*	347
Tmote (2004)	16 bits TI MSP 430 8 MHz	10 KB RAM 48 KB Flash 1 MB ext. Flash	CC2420 802.15.4 250 kbps	21.8 mA ação 5.1 μA *idle*	977
Alta Velocidade					
Btnod v3 (2003)	8 bits Atmel AVR 8 MHz	10 KB RAM 128 KB Flash 180 KB ext. SRAM	Bluetooth / TR 100 38.4 kbps	41 mA ação 151 μA *idle*	40
XYZ (2005)	32 bit ARM Thumb 180 MHz	32 KB RAM 256 KB Flash 256 KB ext. SRAM	CC2420 802.15.4 250 kbps	72.28 mA ação 390 μA *idle*	181
Sun SPOT (2006)	32 bit ARM Thumb 180 MHz	512 KB RAM 4 MB Flash	CC2420 802.15.4 250 kbps	98 mA ação 390 μA *idle*	160
Imote 2 (2006)	32 bit ARM XScale 13-416 MHz	256 KB RAM 32 KB Flash 32 MB ext. SDRAM	CC2420 802.15.4 250 kbps	66 mA ação 390 μA *idle*	16

2.1.3 Aplicações RSSF

RSSFs têm sido empregadas em diversos monitoramentos, utilizando um ou mais tipos de nós sensores. Tal diversidade se deve a diferentes cenários e requisitos, e conseqüentemente, precisam ser satisfeitos pelo desenvolvedor. Em meio a tais variedades, é possível identificar características comuns de modo a facilitar o mapeamento de abstrações de programação. Mottola e Picco (2011) apresentam uma visão de alto nível sobre estas dimensões (Figura 2.4). A seguir será abordado cada item que compõe a taxonomia.

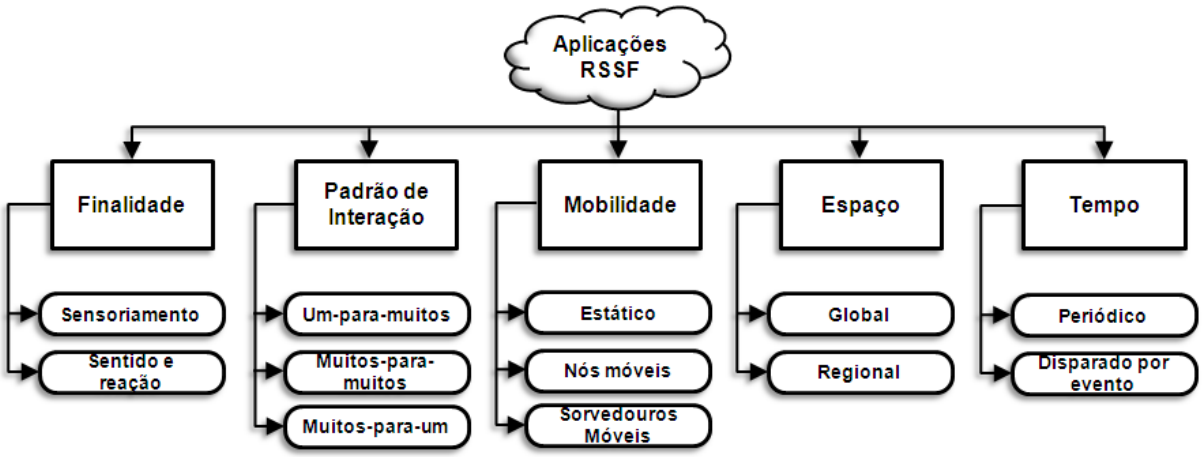


Figura 2.4: Taxonomia das aplicações RSSF (MOTTOLA; PICCO, 2011).

Finalidade - o sistema interage com o ambiente de duas formas: somente sensoriando, sentindo e reagindo aos dados coletados. A introdução do atuador neste cenário altera o contexto da aplicação, pois os dados coletados podem ser reportados ao nó sorvedouro e em seguida receber e transmitir os comandos para o sensor que detectou o fenômeno. Em outras palavras, o nó sorvedouro além de ser a ponte de ligação com a estação de monitoramento, agora também detém o controle dos atuadores utilizados na rede (MOTTOLA; PICCO, 2011 apud AKYILDIZ; KASIMOGLU, 2004).

Padrão de Interação - refere-se à maneira como os nós participantes da rede interagem com os demais nós, sendo três os modos de interação: a) Um-para-muitos; b) Muitos-para-muitos e c) Muitos-para-um. A maioria das aplicações utiliza muitos-para-um, pois os dados são canalizados através dos nós coletores até um ponto de coleta central. As aplicações mais recentes são caracterizadas por interações muitos-para-muitos ou um-para-muitos, onde é necessário enviar comandos de configuração para os demais nós participantes da rede, como exemplo, mudança na freqüência de amostragem ou no conjunto de sensores ativos (economia de energia na rede) (MOTTOLA, 2008; MOTTOLA; PICCO, 2011).

Mobilidade - RSSFs têm como característica a dinamicidade em suas topologias, algumas aplicações necessitam oferecer suporte a dispositivos móveis, introduzindo desta forma um maior grau de dinamismo à aplicação. Portanto, mobilidade pode ou não acontecer, sendo que o mais comum é o estático, onde nem os nós sensores nem os sorvedouros se movimentam. Por outro lado, existem aplicações que demandam mobilidade (MOTTOLA, 2008; MOTTOLA; PICCO, 2011) podendo se concentrar em:

- Nós móveis - quando utilizam nós móveis ligados a entidades móveis ou capazes de se mover de forma autonoma, por exemplo, monitoramento de animais.
- Sorvedouros móveis - este caso é caracterizado quando o nó sorvedouro é móvel e se aproxima dos demais nós para coletar os dados e indifere se eles são móveis ou estáticos.

Espaço - as aplicações RSSFs interagem com ambiente através das grandezas de espaço e tempo, expressando dessa forma o fenômeno monitorado. O comportamento do aplicativo em relação ao espaço pode ser (MOTTOLA, 2008; MOTTOLA; PICCO, 2011):

- Global - se aplica quando o processamento envolve toda rede, devido o fenômeno abranger a totalidade geográfica da RSSF implantada
- Regional - ocorre quando o fenômeno monitorado corresponde a uma área limitada de interesse.

Tempo - o comportamento da aplicação em relação ao tempo pode ser:

- Periódica - refere-se às aplicações onde o sensoriamento é contínuo, onde a leitura dos sensores é feita periodicamente coordenada com outras partes, para uma possível tomada de ação.
- Disparado por evento - aplicação que executa algum processamento quando uma determinada condição específica é atendida, caracteriza-se por dois estados no sensor (i) em repouso, quando o evento não é detectado e (ii) disparado, quando o evento é detectado.

A Figura 2.5 ilustra alguns exemplos de aplicações tomando espaço e tempo ortogonais, abrangendo as combinações dessas dimensões. Por exemplo, detecção de intrusos

e monitoramento de inundações são aplicações onde os fenômenos de interesse são esporádicos, porém os dados devem ser coletados e processados através do tempo, para que especialistas possam compreender o fenômeno com base nas condições ambientais possivelmente distantes do fenômeno. Automação de prédios, por sua vez, deve ter tratamento contínuo e limitado a regiões específicas, como por exemplo, controlar a temperatura de uma sala, onde geralmente é necessário empregar sensores de umidade e temperatura na região a ser controlada.

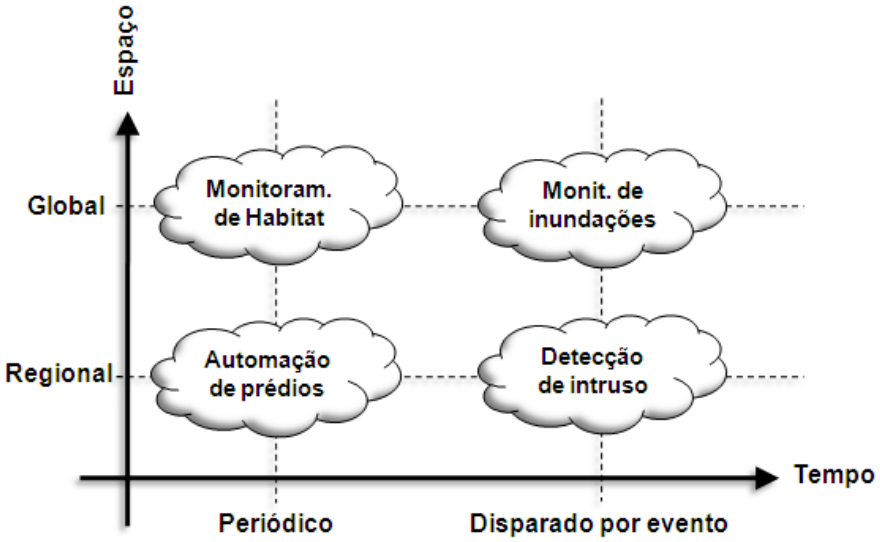


Figura 2.5: Exemplos de aplicações RSSF em espaço e tempo (MOTTOLA; PICCO, 2011).

2.1.4 Programação RSSF

Atualmente o desenvolvimento de uma aplicação RSSF requer programar individualmente para nós sensores, e essa prática é feita diretamente pelos desenvolvedores. Mesmo de posse de middlewares distribuídos e linguagens de configuração de alto nível para nós sensores, a maior parte do código da aplicação ainda é especificada pelo programador. Para dar suporte às aplicações têm sido propostos vários *frameworks*, porém tais artefatos fornecem uma estrutura padrão para o desenvolvimento de aplicações, geralmente para um domínio de aplicação específico. *Frameworks* de programação para nós sensores consistem em ambiente de execução de código e permitem reutilização de componentes de *software*, linguagens de programação e ferramentas de criação e debug de programas executáveis. O fundamento conceitual de um *framework* de programação é o modelo de programação. No modelo de programação são definidos elementos conceituais tais como: abstração do sistema operacional (*threads*, processos, concorrência, manipula-

ção de memória, entre outros), abstração de linguagem (funções, variáveis, procedures e parâmetros de envio e retorno). O modelo de programação reflete onde o programador pensa e estrutura suas aplicações (Figura 2.6).

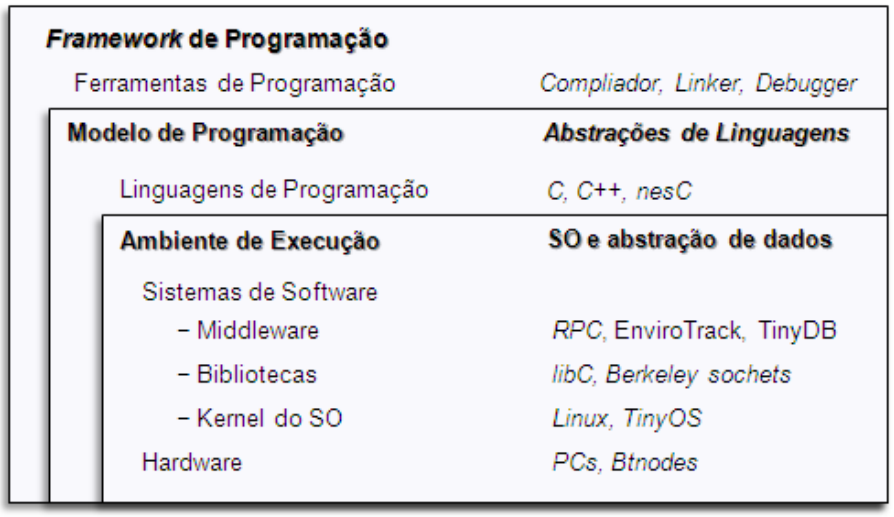


Figura 2.6: Visão em alto nível do *framework* de programação (KASTEN, 2007).

O ambiente de uma RSSF é diferente em vários aspectos do ambiente de uma computação tradicional, pois existem diversos modelos de programação. Para um melhor entendimento sobre os métodos de programação e os dispositivos que fazem parte dessas combinações, Sugihara e Gupta (2008) propuseram dividir em três características que agrupam dispositivos e redes de acordo com suas capacidades apresentadas (Figura 2.7).

- Classe Nó - identifica as plataformas de *hardware* de cada ambiente de computação, agrupando por seu consumo de energia, que fornece indicação de suas capacidades de acordo com sua ordem: a) Ordem W - referem-se à PCs, estações de trabalho e servidores; b) Ordem mW - refere-se a dispositivos portáteis impulsionada por bateria; c) Ordem μW - dispositivos ultra pequenos que se carregam usando fontes de energia externa (por exemplo, luz solar, campo eletromagnético).

- Nós Observáveis - refere-se à programação de propósito geral que se baseia na manipulação de dados e endereços (conteúdo de memória), tempo e espaço com o objetivo de dar exatidão e precisão para atender exigências dos programas. Fornece aproximação da capacidade de raciocínio e requisitos de validação que a aplicação final pode usar, como, (a) Dados+Endereço (DE); (b) DE + Tempo (DET); e (c) DET + Espaço (DETE).

- Tamanho da rede - refere-se à quantidade dos nós implantados, a partir da ordem de

alguns nós, centenas e milhares de nós. Existe uma grande diversidade de tamanhos das RSSF dependendo das necessidades da aplicação, custos e outras considerações.

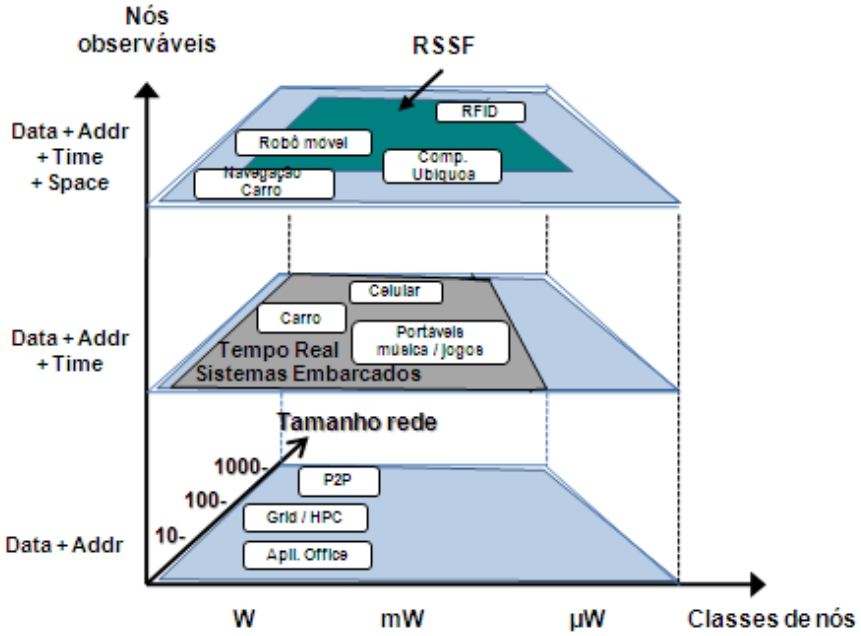


Figura 2.7: Visão abstrata dos nós sensores em relação as grandezas de aplicações (Classes X nós observáveis X Rede) (SUGIHARA; GUPTA, 2008).

2.1.4.1 Modelos de Programação RSSF

Os nós sensores são sistemas reativos, sua programação segue um padrão aparentemente simples, permanecem em modo de espera até ocorrer algum evento pré-determinado, para então realizar um processamento definido e após seu término voltar o modo de espera. Sugihara e Gupta (2008) classificaram os modelos de programação em RSSF em baixo nível (*Platform-Centric*) e alto nível (*Application-Centric*), a Figura 2.8 ilustra essa taxonomia.

- Modelos de programação baixo nível (*Platform-Centric*) - estão relacionados ao nível de nós, abstraindo o *hardware* de modo a permitir o controle flexível dos nós. TinyOS com *nesC* é um dos primeiros exemplos desta classe, servindo de padrão para programação em RSSF (Sugihara e Gupta, 2008). Outra abordagem no baixo nível é a utilização de uma máquina virtual que fornece um ambiente de execução de scripts que são muito menores que os códigos binários do TinyOS. Essa abordagem favorece a programação em situações onde a rede precisa ser dinamicamente reprogramada após sua implantação, utilizando um canal de comunicação sem fio.

- Modelos de programação de alto nível (*Application-Centric*) - têm uma visão centrada em aplicações de modo a favorecer a lógica para facilitar a programação, provendo flexibilidade e otimizando o desempenho do sistema através da promoção da colaboração entre sensores. Uma abordagem característica é fornecer um conjunto de operações para um grupo definido através de critérios diversos. Tais operações se referem à agregação e ao compartilhamento de dados com intuito de que os programadores possam descrever o processamento de dados colaborativos e utilizá-los. Os modelos de programação de alto nível são divididos em dois grupos: Abstração no Nível de Grupo e Abstração no Nível de Rede.

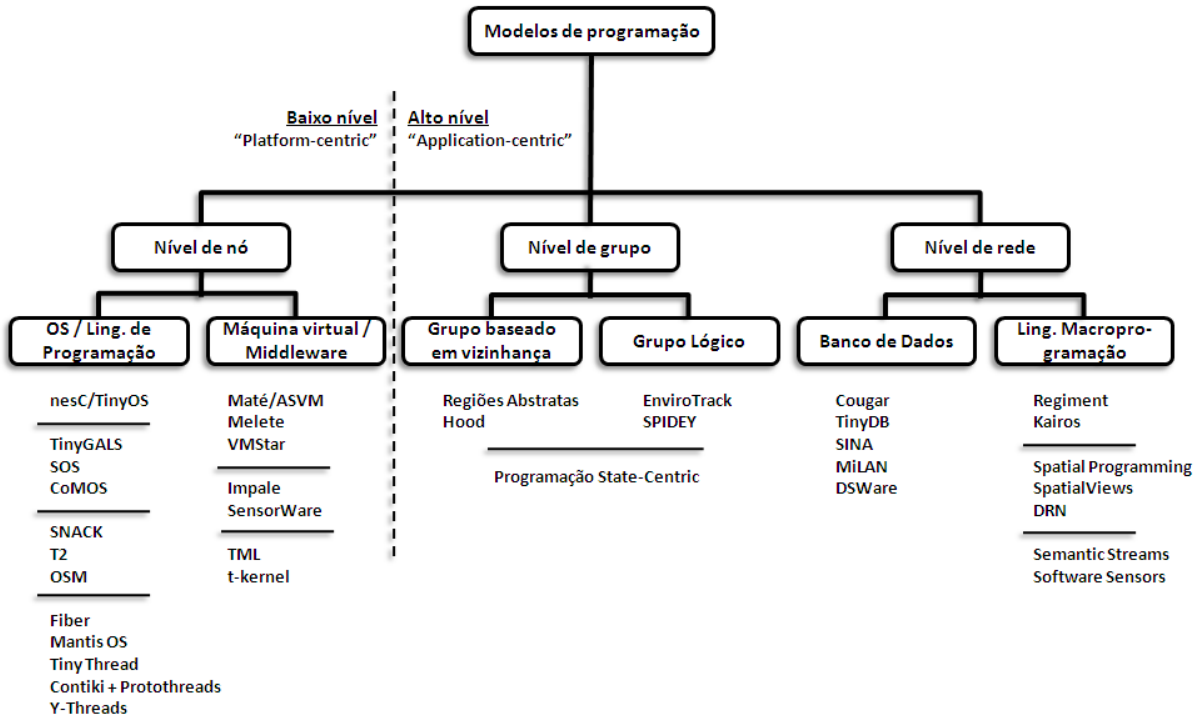


Figura 2.8: Taxonomia de modelos de programação (SUGIHARA; GUPTA, 2008).

2.1.4.2 Sistemas Operacionais

O projeto de sistemas operacionais para RSSF é diferente da concepção tradicional devido suas características peculiares restritivas, além de agregar novas características como: arquitetura, reprogramação, scheduling, modelo de execução e gerenciamento de energia (BAKSHI; PRASANNA, 2008; FRöEHLICH; WANNER, 2008). Em uma aplicação RSSF, os requisitos do aplicativo conduzem o projeto de *hardware*, desde sua capacidade de processamento até a banda de rádio a ser utilizada. De fato, uma plataforma projetada para uma aplicação, não pode ser portada para outra a menos que tal plataforma ofereça mecanismos abstratos de encapsulamento adequados para a plataforma em questão.

Existe uma grande quantidade de sistemas operacionais para RSSFs, que oferecem ao programador abstração conveniente e segura para acessar os recursos de *hardware* como timers, memória, operações básicas primitivas de comunicação e sensoriamento. Atualmente existem muitos sistemas operacionais projetados para atuarem em RSSFs, pode-se citar Mantis (ABRACH *et al.*, 2003), Contiki (DUNKELS; GRONVALL; VOIGT, 2004), SOS (HAN *et al.*, 2005) e TinyOS (RUIZ *et al.*, 2004). Este último é o mais difundido e por isso é tomado como referência para os experimentos neste trabalho.

Mantis OS - Multimodal Networks of In-situ Sensors

Foi desenvolvido na Universidade do Colorado em um projeto de mesmo nome. Trata-se de um sistema operacional *multithread* de código aberto escrito na linguagem C e provendo suporte para as plataformas Mica e Telos (MANTIS, 2006). O projeto prevê que o núcleo do sistema seja composto por um escalonador (schedulling) e drives de baixo nível para comunicação com dispositivos sensores. O escalonador utiliza o algoritmo *Round-Robin* com prioridades para controlar um subconjunto de *threads* POSIX (*Portable Operating System Interface*), e é acionado periodicamente por um temporizador dedicado ou por operações em semáforo. A estrutura de kernel do sistema operacional é composto por uma tabela de *threads* que especifica seu tamanho em tempo de execução. As entradas da tabela contêm um ponteiro para pilha de *threads* (*stack pointer*), ponteiros para função de início, limites inferior e superior da pilha, prioridade, além de um ponteiro para próxima thread na fila do escalonador. As políticas de gerência de consumo de energia têm como ponto de entrada uma *thread* denominada de *idle* que é criada na inicialização do sistema e é executada quando todas as outras *threads* estão bloqueadas.

Contiki

Contiki é um sistema operacional de código aberto, multi-tarefa e altamente portável, sendo que sua configuração típica necessita de 2 Kb de RAM e 40 Kb de ROM. As aplicações são carregadas e descarregadas de maneira dinâmica em tempo de execução e seu *kernel* é baseado em eventos. O Contiki utiliza *protothreads* sobre o seu *kernel*, fornecendo um estilo de programação semelhante às *threads*, mas internamente é totalmente baseado em eventos.

O carregamento dinâmico dos programas possibilita uma maior flexibilidade na reprogramação das RSSF. Sendo que este carregamento pode ser feito sem fio ou por outra fonte como uma EEPROM, sendo mais rápido que atualização via cabo, pois o programa é em média 1 a 10% menor que a imagem do sistema proporcionando uma solução ideal para reprogramação de RSSF (GONçALVES, 2008).

SOS

Sistema operacional para RSSF desenvolvido no *Networked and Embedded System Lab* na Universidade da Califórnia em Los Angeles, tendo como principal objetivo tratar reconfiguração dinâmica dos serviços nas RSSF. O sistema é organizado como um conjunto de módulos binários que implementam tarefas ou funções específicas, ou seja, uma aplicação é composta por módulos que interagem entre si através de interfaces de métodos e passagem de mensagens, podendo ser comparável ao funcionamento do *TinyOS*. A passagem de mensagens funciona em modo assíncrono e é controlado por um escalonador que controla as mesmas em uma fila ordenada por prioridade, pois a mensagem é passada através de uma função tratadora específica para o módulo de destino. Todos os módulos implementam tratamento para mensagens init e final, utilizadas para carga e finalização dos módulos pelo kernel. O modelo de reconfiguração dinâmica adotado pelo SOS requisita memória e sobrecusto considerável, porém segundo os autores, estas características são aceitáveis para a maioria das aplicações RSSF (WANNER, 2006).

TinyOS - Tiny Microthreading Operating System

Inicialmente desenvolvido por Jason Hill, em seu projeto de Mestrado na Universidade de Berkeley, o *TinyOS* é um sistema operacional simples e compacto, baseado em eventos, desenvolvido para atender alguns dos requisitos das RSSF como operações de concorrência com requisitos mínimos de *hardware* e economia de energia. O ambiente *TinyOS* é um ambiente de desenvolvimento com código aberto, um modelo e uma linguagem de programação que permite construir um sistema operacional para cada aplicação (RUIZ *et al.*, 2004). O *TinyOS* é organizado como uma coleção de componentes de *software* composto por interfaces de duas fases, mecanismo para controlar execução de chamadas de procedimentos e sua comunicação é baseada em eventos. Cada componente de *software* declara os comandos a que responde e eventos que sinaliza. Suas composições criam camadas de componentes onde os de mais baixo nível respondem a comandos e sinalizam para os componentes de mais alto nível.

Um programa em *TinyOS* pode ser descrito como um grafo de componentes de *software* baseados em três abstrações de programação:

- Comandos - são denominados métodos não bloqueantes e são usados para requisitar serviços, por exemplo, um envio de uma mensagem. Sendo que o código associado a um comando armazena os parâmetros de uma requisição e condicionalmente escalona uma tarefa para ser executada posteriormente.

- Eventos - são sinalizadores de término de um determinado serviço, por exemplo ocorrência de um evento de *hardware* ou ainda o envio de uma mensagem. Um tratador de eventos pode escalonar tarefas, sinalizar outros eventos, depositar informações no ambiente ou ainda chamar um comando. Componentes de baixo nível possuem tratadores conectados diretamente às interrupções de *hardware*, por exemplo, uma recepção de mensagem ou uma temporização.

- Tarefas - são chamadas de métodos com execução adiada. São unidades básicas de execução e permitem postergar a execução das chamadas de procedimentos. O *TinyOS* executa tarefas e os tratadores de eventos de interrupção em uma única aplicação, uma vez escalonada, uma tarefa executa até concluir, em outras palavras, não existe preempção entre tarefas. Por outro lado, os tratadores de eventos de *hardware*, são executados quando uma interrupção ocorre e executam até terminar, porém podem interromper as tarefas e outros tratadores de interrupção. Para que a execução de uma tarefa seja garantida e não postergue indefinidamente em detrimento a outras tarefas, seu código executado deverá ser curto, ou seja, para operações longas deverão ser particionadas. É necessário que uma tarefa possa terminar para que a próxima seja tratada, ou seja, o código de uma tarefa não pode bloquear ou ficar em espera ocupada (*busy wait*). O *TinyOS* implementa a política de filas FIFO (*First In First Out*), escalonar tarefas para execução sempre que o processador estiver disponível.

Linguagem nesC - network embedded systems C

A linguagem *nesC* é uma linguagem de programação C estilizada, projetada para incluir os conceitos estruturais e modelos de execução do TinyOS. Aplicativos escritos em *nesC* são compostos por componentes, que podem ser construídos e combinados para formar uma aplicação, aumentando a modularidade e reusabilidade de código. Em *nesC*, o comportamento de um componente é especificado em termos de um conjunto de interfaces. Interfaces são bidirecionais, e informam o que um componente usa e o que ele provê (GAY *et al.*, 2003). As interfaces definem a interação entre os componentes que usa e o que provê. O provedor de uma interface implementa comandos enquanto que seu usuário implementa eventos. Rosseto (2006) apresenta um exemplo dessa funcionalidade utilizando a interface *SendMsg*, que define um comando *send* para enviar uma mensagem e um evento *sendDone* para sinalizar o seu envio (Código 2.1).

```
1 interface SendMsg {
2    command result_t send (uint16_t addr, uint8_t length,
```

```
3                                   TOS_MsgPtr msg);
4     event result_t sendDone (TOS_MsgPtr msg, result_t success);
5 }
```

Código 2.1: Comando send envia uma mensagem e um evento *sendDone* utilizando a interface *SendMsg* (ROSSETO, 2006).

2.1.4.3 Componentes de Software

Componentes são estaticamente ligados um ao outro via interfaces. O fluxo de informação pode ocorrer nas camadas inferiores (via comandos) ou nas camadas superiores (via eventos). As aplicações em *nesC* consistem de um ou mais componentes ligados entre si.

Componente - especifica um conjunto de interfaces pelo qual está conectado a outros componentes, de forma a prover e/ou usar um conjunto de interfaces providos/usados por/para outros componentes. Existem dois tipos de componentes em *nesC*:

- Modules - contém o código da aplicação, implementa uma ou mais interfaces, bem como seu comportamento interno.
- Configuration - configura as ligações entre os componentes, ou seja, ligam componentes para formar um novo componente.

Interface - São bidirecionais, especificando o conjunto de funções disponíveis: comandos (podem ser chamados) e eventos (quais precisam ser tratados), a Figura 2.9 ilustra este cenário.

- Componentes são ligados através das interfaces;
- A interface pode prover ou usar um componente.
- Quando proveem, (a) todos os comandos têm que ser implementados e (b) todos os eventos deverão ser chamados, por exemplo, interface Leds e componente LedsC.
- Quando usam, (a) todos os comandos podem ser chamados e (b) todos os eventos tem que ser implementados, por exemplo, interface Timer e o componente Blink
- Ligação das interfaces, por exemplo, a ligação M.x -> N.y, que significa que a interface x usada pelo componente M é implementada pela interface y provida pelo componente N.

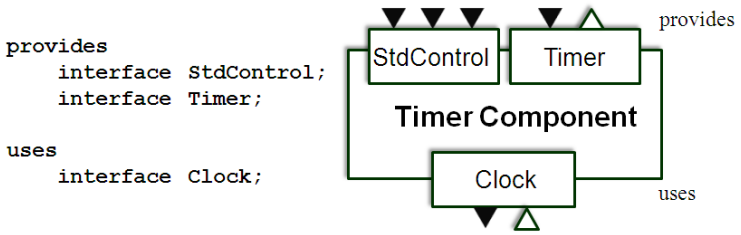


Figura 2.9: Taxonomia de modelos de programação (SUGIHARA; GUPTA, 2008).

A Tabela 2.2 descreve as convenções de nomes aplicadas à linguagem:

Tabela 2.2: Convenções de nomes em *nesC* (RUIZ *et al.*, 2004).

Identificador	Regras para nomes	Exemplos
Interface	Verbos ou substantivos.	ADC, SendMsg.
Componentes	Substantivo com terminação:	
	• C - componente, provendo interfaces:	TimerC
	• M - módulos, implementações.	TimerM
Arquivos	Com sufixo .nc	TimerC.nc
		TimerM.nc

A seguir serão descritos os estágios de desenvolvimento de aplicações para RSSF, utilizando o modelo formal proposto. A proposta parte da modelagem de componentes expressos XMs, em seguida adiciona-se comunicação entre esses componentes fazendo com que o sistema seja descrito completamente.

2.2 Modelo de Computação Geral: X-Machine

Segundo Silva (2007), o conceito de função computável é o marco do nascimento de um novo ramo da matemática denominado de teoria da computação, que tem por finalidade a resolução de um problema de forma automática empregando uma função que utiliza determinados procedimentos denominados modelos computacionais. A teoria da computação abrange o estudo destes modelos e seu respectivo poder computacional que corresponde a classes de problemas e suas respectivas representações. Como o modelo computacional é incapaz de compreender diretamente a língua humana (falada ou escrita), devido sua grande variedade de significados e/ou acepções de uma mesma palavra, foram criadas as linguagens de programação. Tais linguagens têm por objetivo a diminuição do distanciamento entre a linguagem natural (humana) e a linguagem de máquina, além de buscar a eliminação das ambigüidades.

A teoria da computação provê conceitos e princípios que ajudam a entender a

natureza geral da computação, isto é possível por meio da construção de modelos de computadores abstratos para resolução de determinados problemas. Para modelagem é necessário a introdução do conceito de um autômato, que se refere a uma construção com todas as características de um computador digital: entrada, saída, armazenamento temporário e tomada de decisão. O autômato é capaz de reconhecer uma linguagem, capaz de identificar se uma cadeia de símbolos faz parte da referida linguagem ou não, essas linguagens são denominadas de linguagens formais. As linguagens formais são abstrações das características de uma linguagem de programação, sendo assim, possuem um conjunto de símbolos, regras de formação de sentenças, tem uma sintaxe bem definida e uma semântica precisa de modo a não apresentarem ambigüidades ou sentenças sem significado (VIEIRA, 2006).

Em 1936, Alan Turing descreveu os termos matemáticos precisos como um sistema formal automático com regras simples de operação, pode ser poderoso. Turing uniu matemática e lógica através de uma máquina abstrata, tornando possível sistemas processadores de símbolos, sua teoria abriu novas perspectivas no esforço de formalização da matemática, além de marcar fortemente a história da computação. Na seção seguinte serão descritos alguns métodos formais utilizados na modelagem formal de sistemas.

2.2.1 Modelagem e Métodos Formais

A utilização de um método formal no processo de desenvolvimento de *software* permite a especificação de sistemas complexos a partir de entidades abstratas, independentes da implementação, na construção e verificação de sistemas. O aumento do nível de abstração facilita a especificação dos dados, comportamento e funções do sistema, de modo a se abster de detalhes desnecessários ao processo.

Alguns métodos formais como Máquinas de Estado Finito ou Redes de Petri, conseguem descrever a dinâmica de um sistema, porém falham para descrever como um dado é afetado em cada operação no diagrama de transição de estados. Outros métodos, como Statecharts, conseguem capturar a dinâmica e o comportamento dos dados em cada operação, porém são susceptíveis a diversas interpretações (BARNARD; WHITWORTH; WOODWARD, 1996; KEFALAS; ELEFTHERAKIS; KEHRIS, 2003a, 2003b).

X-Machine é um método formal intuitivo, que consegue descrever os tipos de dados de modo formal e as funções por meio de notação matemática conhecida, além de conseguir expressar a dinâmica e o comportamento do sistema (como um dado é afetado após uma operação). Além disso, o conjunto de X-Machines pode ser visto como componentes que

podem se comunicar e conseqüentemente formar um sistema maior através de composição, o que é uma característica interessante para este trabalho, pois as aplicações no TinyOS são combinações de componentes descritos em *nesC*.

2.2.2 X-Machines

O modelo X-Machine (XM) foi proposto por Eilenberg (1974) para ser uma máquina geral de computação, uma alternativa para autômato finito (AF), autômato com pilha (AP) e máquina de Turing (MT) e outros tipos. O termo X de seu nome refere-se ao um tipo de variável, ou seja, uma XM é uma máquina ou dispositivo para manipular tipos de objeto X. Por exemplo, uma calculadora poderia ser descrita como uma máquina de ponto flutuante, pois ela manipula números de ponto flutuante, bem como um automóvel poderia ser descrito como máquina de transporte, pois transporta passageiros de um ponto a outro (STTANETT, 2006).

No entanto, Holcombe, Holcombe e Ipate (1998) estendeu o modelo chamando a atenção da comunidade de Tecnologia da Informação (TI), demonstrando que poderia ser utilizado como ferramenta para especificação e teste de sistemas, desde então tem sido utilizado em diversos trabalhos empregando principalmente especificação e modelagem (LAYCOCH, 1993; IPATE, 1995; HOLCOMBE; HOLCOMBE; IPATE, 1998; BALANESCU; GHEORGHE; HOLCOMBE, 2001; FAIRTLOUGH *et al.*, 2005; BRAGA; SANTOS; JUNIOR, 2010a, 2010b) . Nesta abordagem, a fase de testes que tradicionalmente era adotada como fase final do desenvolvimento, foi deslocada para trabalhar em paralelo com as fases iniciais do projeto de *software*.

O modelo XM se assemelha à máquina de estado finito, porém com duas diferenças significativas, a) as transições são rotuladas com funções e b) estruturas de memória ligadas à máquina. Tais particularidades permitem ao modelo ser mais expressivo e flexível em relação à máquina de estado finito, além de permitir definir outros modelos de classe. Uma subclasse da XM proposta no Laycoch (1993) denominado Stream X-Machine (SXM), é particularmente interessante, pois consegue modelar estrutura de dados não triviais como uma tupla de memória. A classe emprega uma abordagem diagramática para modelagem dos dados e controle de um sistema de forma mais apropriada. Para tal, utiliza-se de métodos de integração, onde as transições entre estados são realizadas por meio de aplicação de funções descritas por notação formal e o controle de dados é realizado na memória da máquina. As funções recebem símbolos de entrada e a memória valores e produzem uma saída enquanto modificam os valores presentes na memória (KEFALAS;

ELEFTHERAKIS; KEHRIS, 2003a, 2003b; AGUADO, 2004).

Definição Formal - Uma XM 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 (Figura 2.10). Cada transição na XM faz com um elemento presente no *stream* de entrada σ e o adiciona ao *stream* de saída γ. A XM pode ser definida formalmente como uma óctupla M = (Σ, Γ, Q, M, ϕ, F, q_0, m_0) onde:

- Σ, Γ são os alfabetos de entrada e saída;
- Q é o conjunto finito de estados;
- M é o conjunto (possivelmente) infinito chamado memória;
- 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,
- 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 descrita como uma função de transição de estado,

$$F : QX\phi \rightarrow Q$$

- q_0, m_0 são o estado inicial e a memória inicial respectivamente.

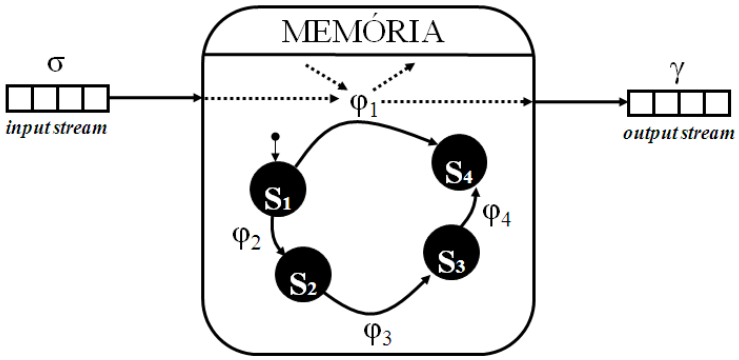


Figura 2.10: Modelo abstrado da XM (adaptado de Eleftherakis (2003).

As XMs, embora de natureza abstrata, possuem o mesmo poder computacional das Máquinas de Turing (MT) (IPATE, 1995; KEFALAS; ELEFTHERAKIS; KEHRIS, 2003b; AGUADO, 2004) e são expressivas o suficiente para representarem a implementação de

um sistema. Esta característica é bastante útil na modelagem, além de facilitar o desenvolvimento de ferramentas que utilizem a metodologia de construção de aplicações de componentes XMs mais prática.

A Tabela 2.3 apresenta um comparativo entre Máquina de Turing e X-Machine, as provas matemáticas de equivalência são mostradas nos trabalhos de Ipate (IPATE, 1995), Eleftherakis (ELEFTHERAKIS, 2003), Aguado (AGUADO, 2004) e Stannett (STTANETT, 2006). A utilização das XMs na especificação de sistemas oferece ainda uma estratégia de testes no modelo, o que permite o refino através da diminuição de ambigüidades e, por conseguinte, uma melhor descrição do *software* (KEFALAS; ELEFTHERAKIS; KEHRIS, 2001).

Tabela 2.3: Definição de uma X-Machine e Máquina de Turing (adaptado de Walkinshaw (2002)).

	X-Machine		Máquina de Turing
Σ	Alfabeto de entrada (input).	Σ	Alfabeto de entrada (input)
Γ	Alfabeto de saída (output).	Γ	Alfabeto da fila ($\Sigma \subseteq \Gamma$).
Q	Conjunto de estados finitos.	Q	Conjunto de estados finitos (sem estado de parada).
M	Memória (possivelmente) infinita.		Sem correspondente.
φ	Um conjunto finito de funções parciais que mapeiam uma entrada e um estado φ de memória numa saída e um novo estado de memória, $\varphi : \Sigma X M \rightarrow \Gamma X M$		Sem correspondente.
F	Função parcial de próximo estado.	δ	Função de transição que leva um estado e um símbolo da fita (ou branco) e leva para um estado (ou estado de parada) dada por: $\delta : QX(\Gamma \cup \Delta) \rightarrow (Q \cup h)X(\Gamma \cup \Delta)XR, L, S$
q_0	Estado inicial ($q_0 \subseteq Q$).	q_0	Estado inicial ($q_0 \subseteq Q$).
m_0	Memória inicial.		Sem correspondente.

2.2.3 Communicating X-Machines

O modelo Communicating X-Machines (CXM), ilustrado na Figura 2.11, é uma extensão da XM provido de uma ou mais portas de entrada e saída (input e output), onde cada saída é conectada a uma porta de entrada da outra máquina, permitindo a troca de mensagens entre ambas (comunicação), de forma que uma função pode ler a entrada a partir de um stream de dados de comunicação (mensagem) enviada por outra máquina. As funções podem escrever a mensagem no stream de comunicação de saída de outra máquina. Após a instanciação dos parâmetros de saída, a mensagem será enviada

ao destino. CXM é um método formal que pode ser usado para modelar o comportamento de um sistema que se comunica (BARNARD, 1998).

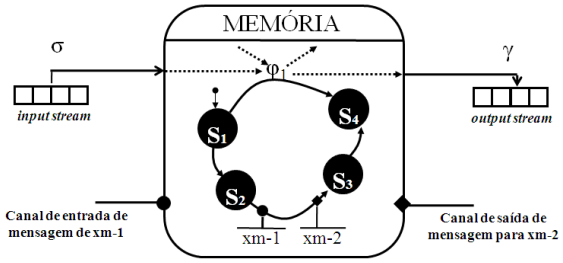


Figura 2.11: Communicating X-Machine (adaptado de Kefalas, Eleftherakis e Kehris (2003b)).

A comunicação de um sistema CXM com n componentes, Figura 2.12, pode ser descrito por meio de uma matriz de comunicação (CM):

$\mathbf{CXM_n} = ((\mathbf{XMC_i})_{\mathbf{i=1,...n}}, \mathbf{CM}, \mathbf{C_o})$, onde:

- $\boldsymbol{XMC_i}$ é um componente X-Machine do sistema;
- $\boldsymbol{CM}$ é uma matriz n X n, denominada matriz de comunicação;
- $\boldsymbol{C_o}$ é o inicio da matriz de comunicação.

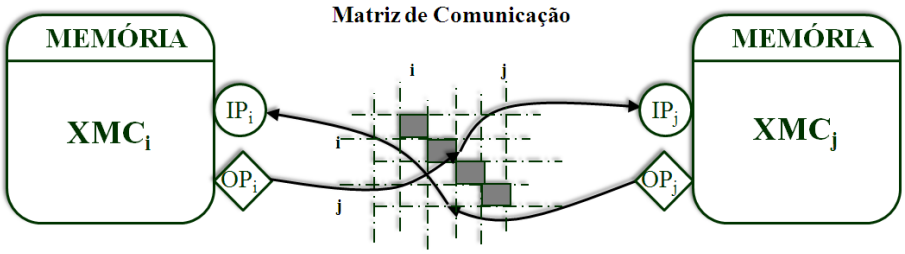


Figura 2.12: Comunicação dos componentes X-Machine XMC_i e XMC_j através das portas OP e IP (adaptado de Kefalas, Eleftherakis e Kehris (2003b)).

O componente XMC_i de CXM_n é diferente de uma XM padrão, pois se utilizam de portas IN e OUT para se comunicarem. Essas portas são ligadas a CM que atua como meio de comunicação entre as XMCs. Nas células da CM estão contidas as mensagens trocadas entre as XMC_i e XMC_j, ou seja, XMC_i lê a i-ésima coluna e escreve na i-ésima linha. A CM pode conter um valor indefinido λ, que significa que não existe mensagem, enquanto que as demais células podem estar vazias. As mensagens podem ser algum tipo definido na memória em todos os componentes XMC, de modo que o envio e recebimento de mensagens requerem um *input* na porta IP e um *output* na porta OP. Ainda nas portas IP e OP, existe um tipo especial de comunicação que se refere a estados e funções

respectivamente, denominados por estados de comunicação (*Communicating States*) e funções de comunicação (*Communicating Functions*), conforme pode ser visto na Figura 2.13. As funções de comunicação derivam dos estados de comunicação e aceitam um símbolo vazio ε como *input* e produzem um símbolo ε como *output*, sem afetar a memória. As funções de comunicação ou lêem um elemento na CM e o carregam para porta *IN*, ou escrevem um elemento que está na porta OUT para a CM:

$$cf(\varepsilon, m, in, out, c) = (\varepsilon, m, in', out', c'), onde:$$

$$m \in M, in, in' \in IN, out, out' \in OUT, c, c' \in CM$$

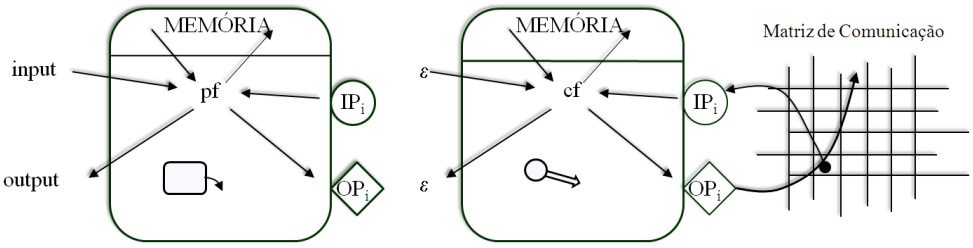


Figura 2.13: Visão da comunicação de estados e funções (adaptado de Kefalas, Eleftherakis e Kehris (2003a)).

A metodologia COMX, proposta por Barnard (1998) se utiliza da CXM para construção de sistemas de comunicação, a metodologia segue uma abordagem *top-down* e favorece a verificação de propriedades do sistema de comunicação tais como: alcançabilidade (*reachability*), *boundness*, verificação de *deadlocks*, entre outras. No entanto, não existe nenhum esforço para favorecer o reuso, preservação semântica das XM *stand-alone* (visão de componentes), nem tampouco o favorecimento de particionamento do problema com intuito de proporcionar melhor entendimento.

2.2.4 X-Machine Description Language - XMDL

O modelo proposto possui uma linguagem para expressar a XM, trata-se da X-Machine Description Language - XMDL. A XMDL é uma linguagem de marcação tipo eXtensible Markup Language - XML baseada em tags e utilizada para declaração de partes de XM como tipos, conjuntos de estados, símbolos de inputs e outputs, valores de memória, funções entre outros, a Tabela 2.4 apresenta as palavras reservadas da linguagem e seus respectivos correspondentes.

Tabela 2.4: Palavras-chave XMDL (adaptado de Eleftherakis (2003)).

Elemento XM	Sintax XMDL	Semântica Informal
M	# model nome_modelo	Atribui nome ao modelo.
Σ	# input <conjunto de *inputs*>	Descreve o conjunto de *inputs*.
Γ	# output <conjunto de *outputs*>	Descreve o conjunto de *outputs*.
Q	# states <conjunto de estados>	Define o conjunto de estados.
M	# memory <tupla de memória>	Define a tupla de memória.
φ	# fun <função>	Define uma função φ.
F	$\#transition(q, \emptyset) = q$	Define cada transição F.
q_0	#init_state <estado>	Conjunto dos estados iniciais.
m_0	# init_memory <memória>	Memória inicial.

Segundo Kefalas, Eleftherakis e Kehris (2001), a proposta da linguagem é auxiliar na construção de diversas ferramentas que utilizem os conceitos do referido modelo, apoiando-se na simplicidade das estruturas primitivas oferecidas pela mesma, podendo fazer várias combinações. Outra característica importante é de ser uma linguagem de marcação, pois remove imposições de uma linguagem posicional, além de facilitar sua transformação para outra linguagem mais conhecida. Outro ponto forte da mesma é a sua similaridade à notação matemática, que facilita a aprendizagem para alguém que não esteja familiarizado com a linguagem. A Tabela 2.5 apresenta a estrutura de dados primitiva da linguagem do autômato.

Tabela 2.5: Estrutura de Dados primitiva do XMDL (WALKINSHAW, 2002).

Estrutura	Descrição
Identifier	Ponteiro que contém um endereço de memória que pode ser uma constante ou uma variável.
Bag	Estrutura onde elementos duplicados são permitidos e sua ordem não têm importância.
Tuple	Conjunto de elementos ordenados, onde são descritos: parâmetros para funções, entradas e saídas das X-Machines.
Sequence	Estrutura onde elementos duplicados são permitidos e sua ordem é importante.
Set	Estrutura onde elementos duplicados não são permitidos e sua ordem não é importante.

Os *scripts* XMDL são conjuntos de sentenças que iniciam com uma tag e finalizam com um ponto final, seus identificadores são constantes ou variáveis, ou ainda identificadores especiais como *nil*, que indica conjunto vazio, *bag* ou sequence. Apesar de usar somente cinco estruturas de dados diferentes, é altamente expressivo devido a possibili-

dade de se construir estruturas complexas através da composição em aninhamento das estruturas primitivas.

Existem também os conjuntos embutidos que são construídos sobre as estruturas primitivas apresentadas anteriormente. Os conjuntos embutidos podem ser visualizados na Tabela 2.6.

Tabela 2.6: Conjuntos embutidos XMDL (WALKINSHAW, 2002).

Item	Descrição
Identifier	Ponteiro que contém um endereço de memória que pode ser uma constante ou uma variável.
Integer	Conjunto de inteiros.
Natural10	Conjunto de inteiros positivos incluindo o zero.
Natural	Conjunto de inteiros positivos.
Real	Conjunto de números reais.
Char	Conjunto de letras do alfabeto em latim.
Boolean	Refere-se aos valores "verdadeiro"e "falso".

As declarações em XMDL referem-se à definição matemática da tupla que representa uma XM, além de incluírem o nome do modelo em questão, comentários e a declaração de funções externas definidas em outra linguagem (KEFALAS; ELEFTHERAKIS; KEHRIS, 2001). As seguir será apresentado como se devem descrever em XMDL as especificações da XM.

- Nome do Modelo (M) - nome que especifica o modelo é designado por #model + nome_do_modelo. Ex: #model Sensor_Temperatura.
- Símbolos de *input* e *output* (Σ, Γ) - como são conjuntos simples, são definidos por expressões ou valores explícitos.
- Type (ϕ) - representa os dados em que a máquina se baseia, podendo ser especificados como coleções de dados (por exemplo, o tipo *Set*) ou ainda conjuntos embutidos (*Integer*, por exemplo). Ex: #type bell = boolean.
- States (Q, q_0) - representa os estados do modelo. Ex: a) #states = {*Init, Start, Stop*}.; b) init_state{*Init*}.
- Type Function (φ) - é usado para mapear um valor de entrada na memória para um valor de saída da memória;

- Memory (M, m_0) - como os valores de memória podem ser infinitos, suas definições são através de tuplas. Ex: #memory (fila, *nil*).
- Transition Function (F) - especifica as funções de transição do modelo. Ex: #transition(*Init, to_start*) = Start.
- Comentário - é especificado da seguinte forma: /* exemplo de comentário*/.

2.3 Desenvolvimento Orientado a Modelos

Uma abordagem denominada Desenvolvimento Orientado a Modelos (*Model-Driven Development* - MDD), surgiu como uma alternativa para os processos de desenvolvimento de *software*, com intuito de simplificá-lo e formalizá-lo através da elevação dos níveis de abstração utilizados no processo. A MDD propõe que o modelo seja uma abstração de um produto de *software* relacionada com uma linguagem de modelagem de forma explícita, privilegiando o alto nível do modelo (MIYAZAWA, 2008). Conceitos e pontos relevantes em MDD serão apresentados na seção seguinte.

2.3.1 Conceitos MDD

O MDD consiste na utilização de modelos e tecnologias relacionadas, com intuito de facilitar e documentar o processo de desenvolvimento de *software*, oferecendo ao desenvolvedor um alto nível de abstração para criação de modelos, sem precisar interagir com o código fonte, ou seja, protegendo-o das complexidades requeridas na implementação de diferentes plataformas. A geração de código é concebida através de um mecanismo automático que atua sobre os modelos criados pelo desenvolvedor. Modelos em MDD são abstrações de um produto de *software* relacionado de forma explícita a uma linguagem de modelagem (MIYAZAWA, 2008 apud HAILPERN; TARR, 2006). Esses modelos não são apenas uma referência ou um guia, mas fazem parte do *software* (LUCRéDIO, 2009).

A Figura 2.14 ilustra o processo de criação de *software* usando a abordagem. Inicialmente, o projetista de *software* utiliza ferramentas de modelagem para concepção de um modelo abstrato, para isto a ferramenta deve ser intuitiva e de fácil utilização. Em seguida são aplicadas transformações de modelos, onde são empregadas técnicas de mapeamento de modelo para modelo, ou de modelo para código que é o caso desta pesquisa.

O MDD busca reduzir a distância semântica entre o domínio do problema e sua solução através do aumento do nível de abstração, proporcionado pelos modelos de *software*

e pelas transformações que geram implementações automáticas e que refletem a solução modelada.

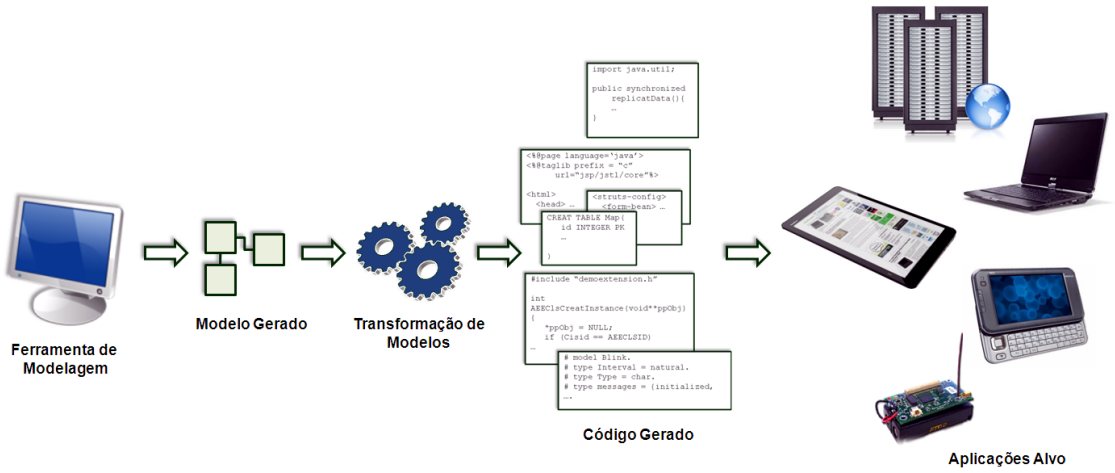


Figura 2.14: Processo de criação de *software* usando MDD.

O MDD é também conhecido pelos seguintes acrônimos: Model-Driven Engineering (MDE) (LUCRéDIO, 2009 apud SCHMIDT, 2006), Model-Driven Software Development (MDSD) (LUCRéDIO, 2009 apud VOELTER; GROHER, 2007) e MD* (LUCRéDIO, 2009 apud VOELTER, 2008). A Figura 2.15 ilustra os principais elementos que compõem a abordagem e a maneira como se combinam. O MDD é composto por:

- Ferramenta de modelagem - possibilita a criação de modelos que necessitam ser semanticamente completos e corretos para serem compreendidos pela máquina. Outra característica é que deve ser intuitiva e de fácil utilização para o engenheiro de *software*.

- Modelos - são entradas para as transformações para código-fonte ou outros modelos. Essa ferramenta deve possibilitar que as regras de mapeamento sejam construídas de maneira mais natural possível, provendo ao engenheiro de *software* construir essas regras (modelo para modelo ou modelo para código) de modo mais simples.

- Mecanismo de execução de transformações - mecanismo que aplique as transformações definidas pelo engenheiro de *software*, de maneira a manter as informações de rastreabilidade, com intuito de saber a origem de cada elemento gerado no modelo ou código-fonte.

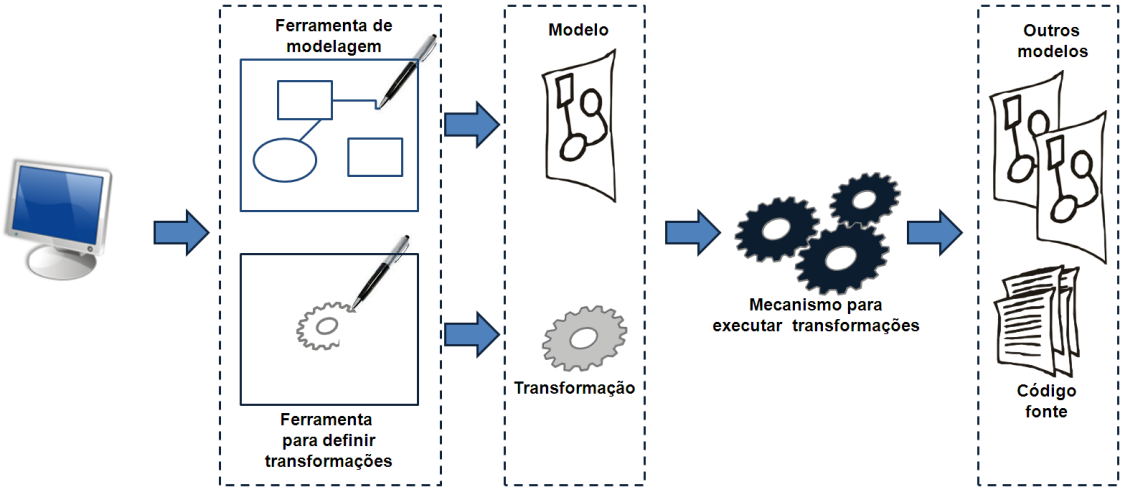


Figura 2.15: Principais elementos de MDD baseado em (LUCRéDIO, 2009).

2.3.2 Eclipse e MDD

Antes de apresentar a abordagem MDD na plataforma Eclipse é necessário conhecer o conceito de metamodelagem e sua arquitetura. Metamodelagem possibilita suporte aos fundamentos MDD por produzir modelos semanticamente corretos e completos, bem como definir e executar transformações e suporte a diferentes linguagens de modelagem (LUCRéDIO, 2009). A Figura 2.16 mostra a arquitetura clássica da metamodelagem em quatro níveis, M0 - primeiro nível refere-se aos dados, M1 - segundo nível corresponde aos metadados ou modelos, M2 - terceiro nível utilizado para definição de modelos é denominado de metamodelo, M3 - trata-se do quarto nível, é utilizado na definição de metamodelos (linguagens de modelagem). Outra característica deste nível é a possibilidade de se auto-instanciar.

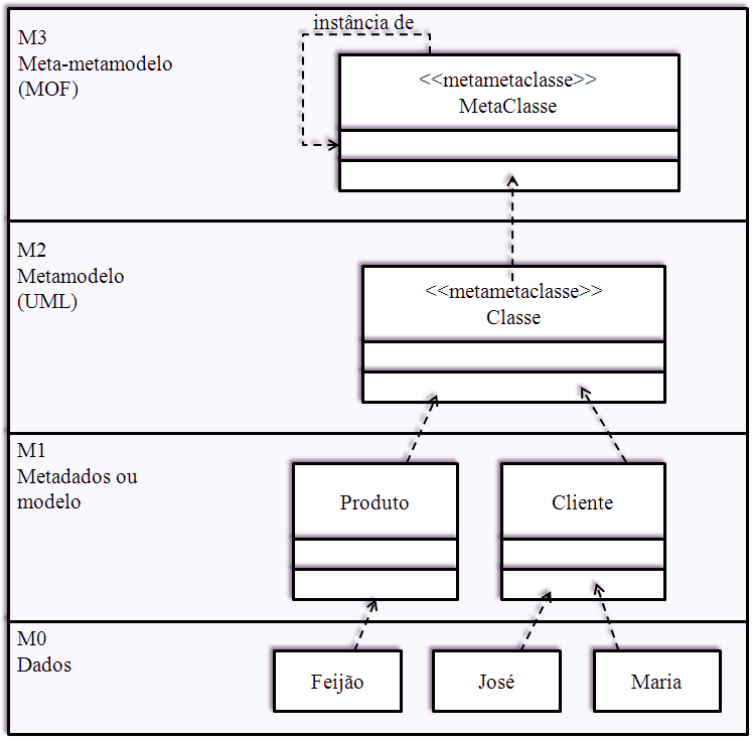


Figura 2.16: Arquitetura clássica de modelagem.

O Eclipse é uma plataforma aberta extensível e escrita na linguagem Java, na qual novos recursos podem ser agregados em qualquer momento, além de permitir a interação de desenvolvedores com a plataforma de desenvolvimento. O Eclipse trabalha com a ideia de plug-ins (Figura 2.17), pois agrega todos seus componentes através deste artifício (exceção ao núcleo da plataforma) (ANISZCZYK; GALLARDO, 2007; FERREIRA, 2009).

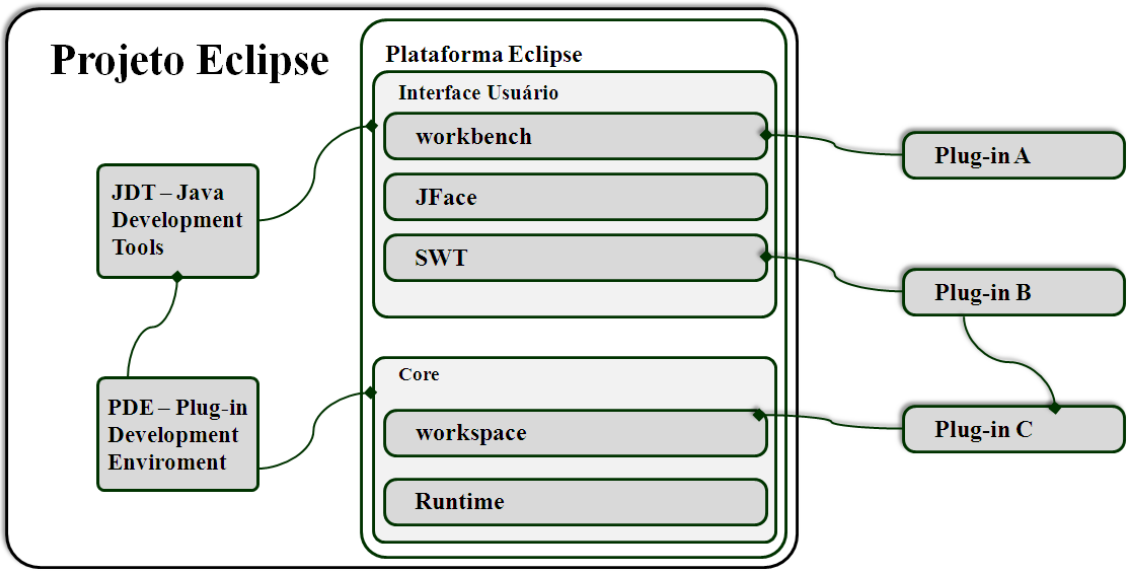


Figura 2.17: Arquitetura de plug-ins Eclipse (GUCLU, 2008).

A abordagem baseada na plataforma Eclipse é liderada pela IBM e tem por base o Eclipse Modeling Framework (EMF), que permite a manipulação de modelos de acordo com seu metamodelo correspondente. O EMF se baseia em um meta-metamodelo denominado Ecore, que por sua vez é baseado no padrão estabelecido pela Object Manegement Group (OMG), o Meta-Object Facility (MOF). O MOF é um metamodelo que serve de base para todas as linguagens de modelagem, além de proporcionar que as ferramentas de modelagem e transformação trabalhem juntas (LUCRéDIO, 2009).

2.4 Resumo

Este capítulo teve por objetivo apresentar o estado da arte sobre RSSF, Métodos Formais e MDD, como base para o desenvolvimento dos capítulos seguintes. Para isto realizou-se uma abordagem dos aspectos básicos das RSSFs, modelos de programação, sistemas operacionais mais usados e por fim, um estudo sobre a linguagem de programação *nesC*.

Em relação a métodos formais, foi abordado o tema de modelo de computação geral, no qual apresentou-se o método formal XM e sua extensão proposta neste trabalho, o CXM. Foram apresentadas as visões diagramáticas e sua linguagem de definição XMDL, esses conceitos são importantes e serão utilizados na definição dos modelos apresentados nos capítulos seguintes para construção de aplicações RSSF.

Por fim, foi apresentada a área de MDD, conceitos e utilização na plataforma Eclipse, que será a ferramenta utilizada na implementação desta pesquisa. No próximo capítulo serão apresentados alguns trabalhos correlatos com esta pesquisa. Será feita também uma comparação entre eles e as características desejáveis para a elaboração de ferramenta de criação de aplicações RSSF.

MDD será utilizado na construção da ferramenta proposta neste trabalho, permitindo dessa forma, automatizar a construção das aplicações RSSF. Para isto, no próximo capítulo serão analisados alguns trabalhos relacionados a esta abordagem, além da elaboração de uma lista de características desejáveis a uma ferramenta de apoio para construção de aplicações RSSFs.

Capítulo 3

Trabalhos Correlatos

Existem diversos projetos que visam facilitar o desenvolvimento de aplicações para RSSFs. Alguns proporcionam um ambiente de desenvolvimento para RSSF, outros permitem a comparação de possíveis soluções a partir de simulação ou emulação. Outros ainda focam no desenvolvimento de aplicações RSSFs provendo linguagens de alto nível, bibliotecas de componentes e compiladores.

Neste capítulo serão analisados alguns trabalhos relacionados com a abordagem proposta, sendo abordado sob dois aspectos: (1) específicos para o desenvolvimento de aplicações RSSF, onde serão apresentadas quatro soluções aplicadas exclusivamente para área em questão, RaPTEX, TinyDT, Viptos e GRATIS são os trabalhos escolhidos; e (2) aplicação de métodos formais para desenvolvimento de aplicações, onde será estudado o *X-Machine Toolkit* e o XMJ Tool, que são *frameworks* de modelagem e geração de código em Java. Embora não gerem código em *nesC*, tais ferramentas servem como parâmetros para o desenvolvimento do presente trabalho. Dessa forma, nas próximas seções serão descritos os trabalhos mencionados, bem como suas vantagens e desvantagens, além de identificar características desejáveis para produção de um ambiente favorável ao desenvolvimento de aplicações RSSF que utilize um método formal.

3.1 RaPTEX - Rapid Prototyping Tool for Embedded Communication Systems

O Raptex (LIM, 2007), Figura 3.1, é um projeto que proporciona um ambiente de desenvolvimento para sistemas embarcados de comunicação, sendo composto por três subsistemas, um modo gráfico que se baseia em componentes para construção de aplicações, um gerador de código que se utiliza da montagem no modo gráfico e um quadro de estimativas de desempenho. A ferramenta utiliza diagramas de blocos que fazem referência a biblioteca de componentes do *TinyOS*, de maneira a dar suporte a simulação de comunicação de sistemas embarcados com restrições de energia.

Para criar uma aplicação RSSF, o desenvolvedor utiliza-se de uma interface gráfica e simplesmente arrasta um bloco (componente) desejado e em seguida realiza sua comunicação com outro componente *nesC*. Na próxima etapa, a geração de código, se dá através de ajustes de parâmetros para customizar a aplicação corrente para em seguida compilar os códigos. Após a compilação, a ferramenta realiza análises do código produzido, de modo a determinar as características de desempenho, como velocidade do processador, tamanho de memória, consumo médio de energia e tempo estimado de vida útil da bateria. Essas informações permitem ao usuário uma melhor avaliação da aplicação, além de proporcionar resultados mais satisfatórios no desenvolvimento.

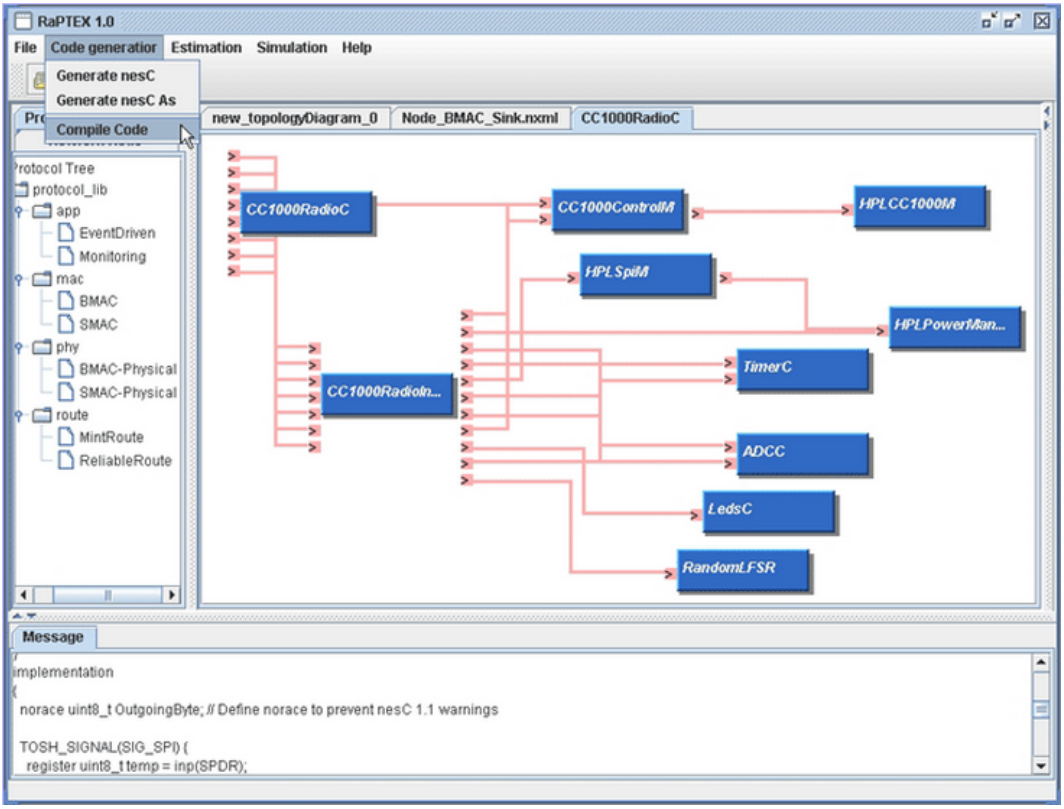


Figura 3.1: Raptex (LIM, 2007).

Apesar das facilidades e vantagens listadas anteriormente, o RaPTEX possui uma desvantagem considerável: o desenvolvedor necessita possuir algum conhecimento do *TinyOS*, informações sobre seus componentes e suas interfaces. Portanto, a ferramenta não provê suporte aos usuários não especialistas, fazendo com que o desenvolvimento de aplicações não seja uma tarefa tão trivial.

3.2 TinyDT

TinyDT (SALLAI; BALOGH; DORA, 2005), Figura 3.2, trata-se de um *plug-in* da plataforma Eclipse baseado no *TinyOS*, onde provê uma IDE (*Integrated Development Enviroment*), que auxilia na construção das aplicações, contando ainda com um analisador sintático que visa a colaborar na validação de sentenças inerentes à linguagem. O *plug-in* oferece ainda ao desenvolvedor *code completion*, suporte para várias plataformas de nós sensores, visualização do *TinyOS* em árvore, além de controle de versão.

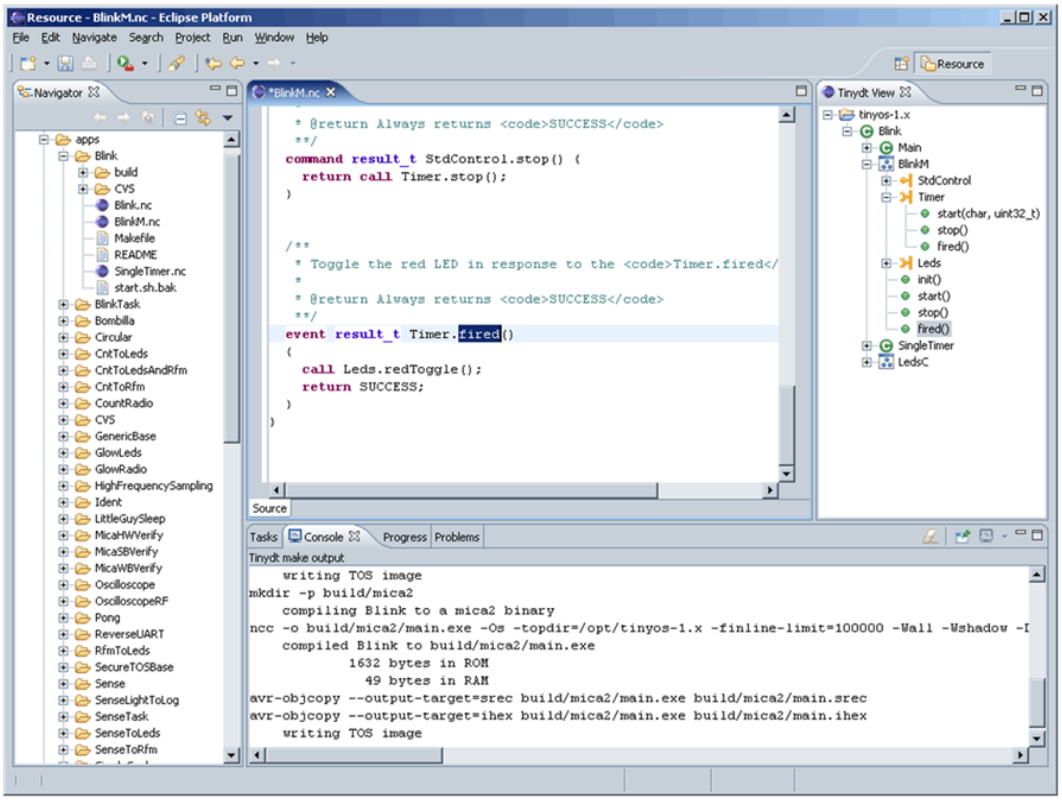


Figura 3.2: TinyDT (SALLAI; BALOGH; DORA, 2005)

Embora a IDE facilite o desenvolvimento de aplicações, a ferramenta não se baseia em nenhum método que garanta a corretude do *software* gerado. Outra desvantagem é de que o *plug-in* se baseia na versão do *TinyOS* 1.x, portanto não prevê atualização da referida plataforma de desenvolvimento.

3.3 Viptos - Visual Ptolemy and TinyOS

Viptos (CHEONG; LEE; ZHAO, 2005), Figura 3.3, fornece uma *framework* gráfico composto por blocos para o desenvolvimento de aplicações em *TinyOS*, onde os usuários

podem criar aplicações através da montagem de diagramas, que é transformado automaticamente em código *nesC*, que pode ser compilado e baixado para plataforma escolhida. Viptos provê suporte não somente ao desenvolvimento de código, mas também a simulação de ambiente. Viptos integra um simulador *TinyOS* (TOSSIM), um simulador de rede (VisualSense). O framework fornece ao desenvolvedor uma ferramenta, ncapp2moml, que converte aplicações existentes no *TinyOS* em modelos Viptos.

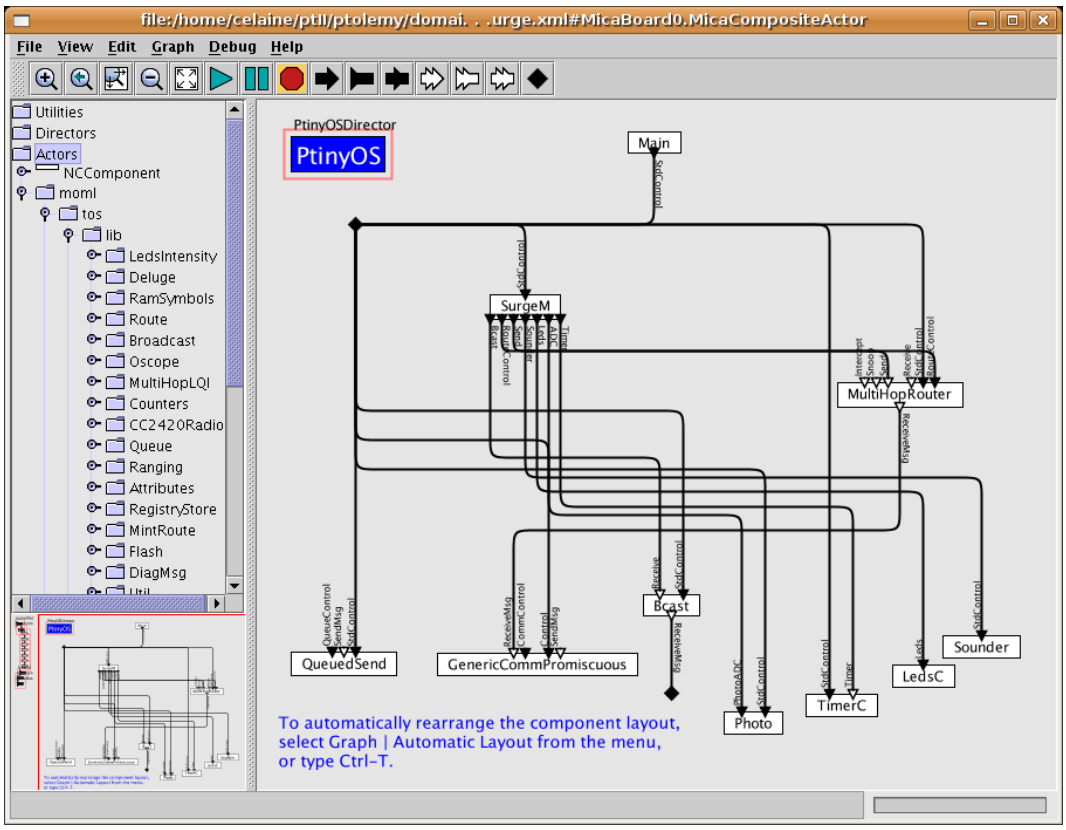


Figura 3.3: Viptos (CHEONG; LEE; ZHAO, 2005).

Apesar de oferecer recursos gráficos para modelagem de aplicações RSSF, além de um ambiente integrado de simulação, geração automática de código *nesC* e uma ferramenta de transformação de modelos, Viptos não oferece um ambiente favorável a desenvolvedores não especialistas, sendo portanto necessário ter prévio conhecimento da plataforma *TinyOS* para construção de aplicações RSSF.

3.4 GRATIS - Graphical Development Enviroment for TinyOS

GRATIS (VOLGYESI; LEDECZI, 2002) (Volgyesi e Ledeczi, 2002), Figura 3.4, é um projeto que fornece um ambiente gráfico baseado em GME (*Generic Model Enviroment*),

onde sua sintaxe é definida em metamodelos e utiliza da notação de diagramas de classe especificados em UML. Sua semântica estática é especificada pela linguagem de restrição de objetos (OCL) com finalidade de geração de código por meio de tradutores do modelo.

O principal objetivo do GRATIS consiste em auxiliar o desenvolvedor na compreensão da plataforma *TinyOS*, através da estrutura de suas aplicações e sua hierarquia apresentada em um modo gráfico. A ferramenta permite que elementos da linguagem *nesC* possam ser reconhecidos através das ligações entre componentes, permitindo expressar a aplicação através de hierarquia de componentes.

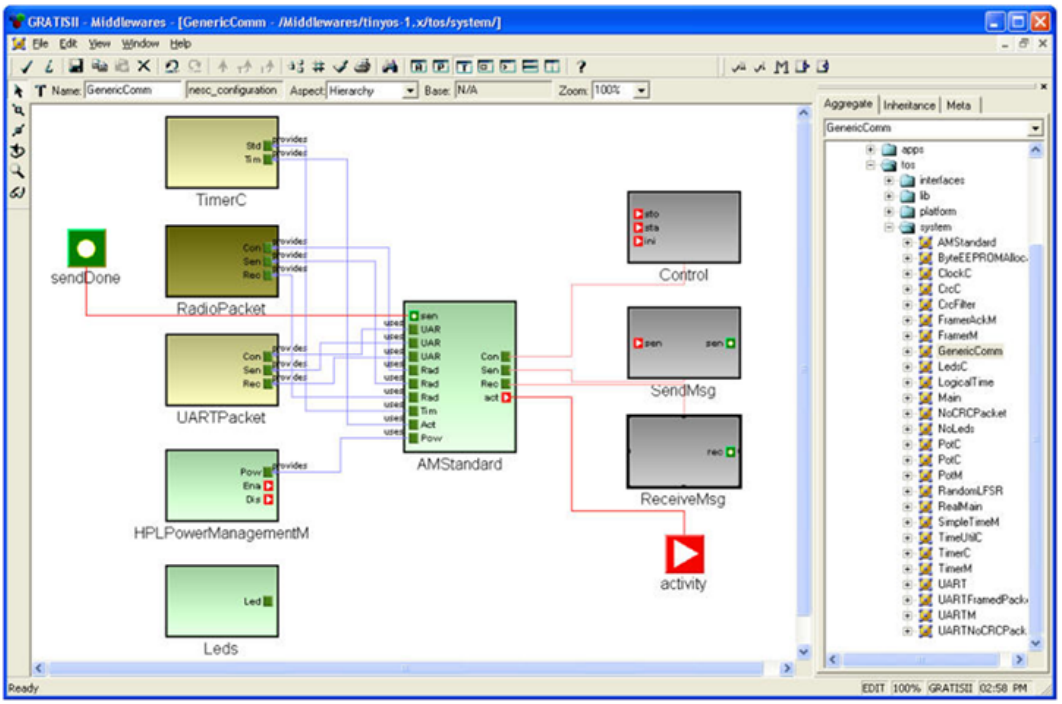


Figura 3.4: GRATIS (VOLGYESI; LEDECZI, 2002).

GRATIS tem como foco principal prover componentes *TinyOS* em modo gráfico, além de fornecer uma biblioteca no ambiente gráfico que utiliza blocos para construção de aplicações e geração de código são implementados em Python. Da mesma forma que as ferramentas citadas anteriormente, GRATIS não oferece suporte adequado para usuários não especialistas, além de não ter tido atualizações no projeto. Sendo, portanto direcionado a versão do *TinyOS* 1.x e *nesC* 1.1.

3.5 XMJ Tool

XMJ Tool (OGUNSHILE, 2005), Figura 3.5, é um projeto baseado no método formal XM e consiste em modelar sistemas utilizando sua notação XMDL e cujo código

resultante é Java. A ferramenta provê uma interface gráfica simples composta de um editor de texto, um editor XMachines e um editor TreeView. A construção de aplicações procede da seguinte forma: o editor de texto recebe especificação do sistema em XMDL, logo em seguida é aplicado um parser que constrói duas visões da aplicação, uma que mostra o modelo XM e que pode ser parametrizado para especificar uma determinada propriedade e outra que é possível visualizar a aplicação através de uma árvore de componentes, respectivamente XMachines e TreeEditor. A ferramenta gera código em Java baseado nas especificações construídas em XMDL, o mecanismo de tradução atua em duas etapas, (i) efetua a tradução do código XMDL para o XML e em seguida (ii) do XML para Java.

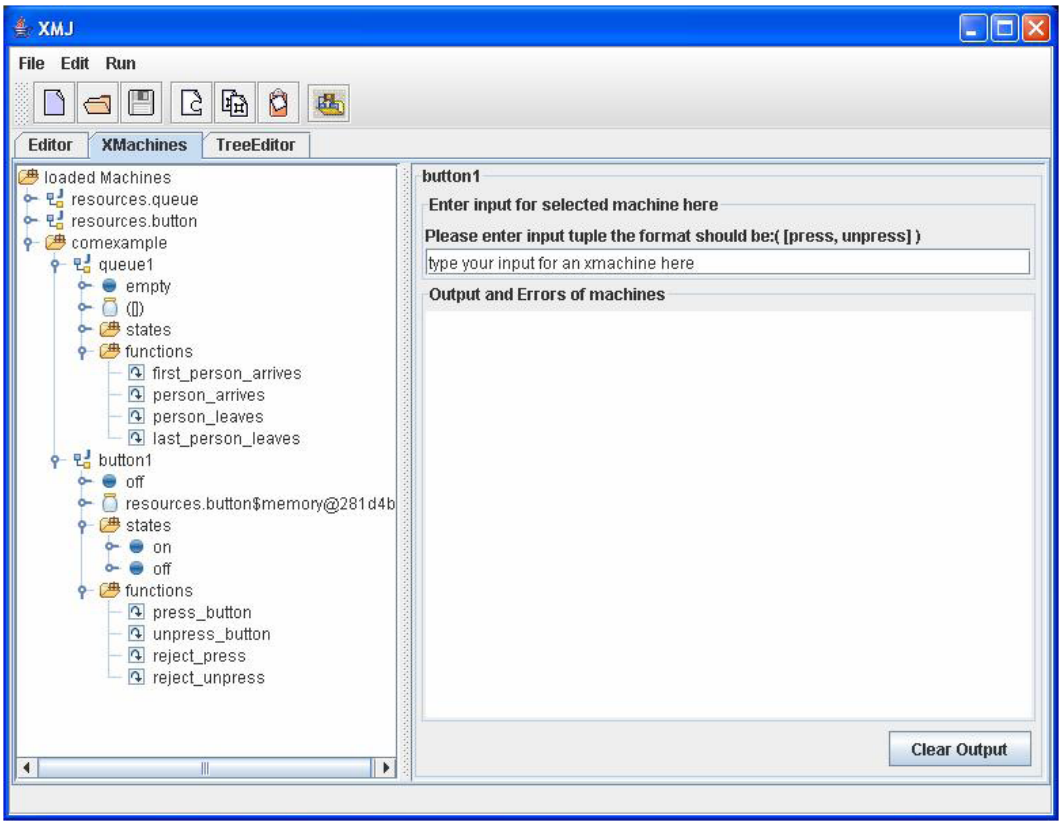


Figura 3.5: XMJ (OGUNSHILE, 2005).

A ferramenta oferece ao desenvolvedor uma interface pobre quando comparada com as já citadas neste capítulo, além de exigir do usuário conhecimento da linguagem XMDL para construção de aplicações. Apesar de ter um método formal como base de sua construção, a ferramenta não oferece suporte a um desenvolvedor não especialista.

3.6 X-Machine Toolkit

O X-Machine Toolkit (ABDUNABI, 2007), Figura 3.6, é uma ferramenta construída sob o método formal XM, é construída sob a plataforma Eclipse e permite a modelagem de aplicações através da construção de autômatos. A ferramenta oferece dois componentes básicos, States (estados) e Transitions (transições), para que o usuário possa modelar suas aplicações. Destaca-se que após a modelagem do sistema, a ferramenta provê a geração automática de código em linguagem Java.

O *toolkit* oferece ao usuário uma interface amigável, pois as aplicações podem ser modeladas através da montagem de um sistema por meio da construção de máquinas de estado, o que favorece o aumento do nível de abstração e possibilita ao desenvolvedor preocupar- se com a lógica em detrimento de detalhes que não sejam tão importantes.

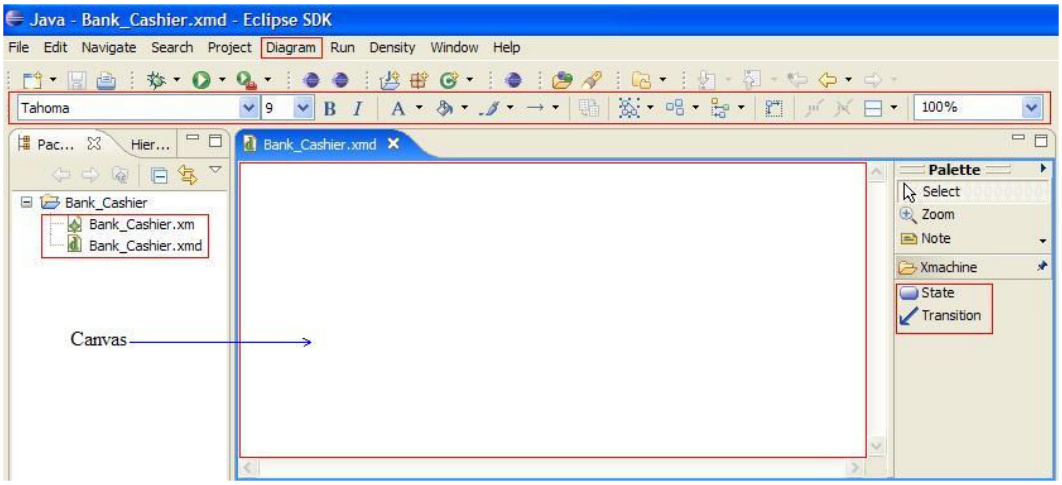


Figura 3.6: X-Machine Toolkit (ABDUNABI, 2007).

O presente trabalho tomou por base a referida ferramenta para construção de um *toolkit* para desenvolvimento de aplicações RSSF, linguagem de programação *nesC*, além da utilização do método formal CXM, que é uma extensão do XM conforme visto no Capítulo 2.

3.7 Comparativo dos Trabalhos Correlatos

Raptex (LIM, 2007), TinyDT (SALLAI; BALOGH; DORA, 2005), Viptos (CHEONG; LEE; ZHAO, 2005) e GRATIS (VOLGYESI; LEDECZI, 2002) são ambientes de desenvolvimento para RSSFs baseados no *TinyOS*. TinyDT é um plug-in do *TinyOS* para o Eclipse que implementa uma IDE, e auxilia na construção das aplicações. Embora a IDE venha

a prover facilidades para o desenvolvimento de aplicações, a ferramenta não se baseia em nenhum método que garanta a corretude do *software* gerado.

Viptos, GRATIS e Raptex são ferramentas que proveem em seus frameworks blocos para construção das aplicações, o que possibilita a usuários não-especialistas terem suporte para o desenvolvimento de *software*. Embora GRATIS e Raptex ofereçam uma visão clara da estrutura das aplicações em *TinyOS*, elas não oferecem nenhuma ferramenta de verificação de propriedades, nem tampouco estão alicerçados em nenhum método formal que sirva de guia para o processo de desenvolvimento de *software* com qualidade.

Viptos está inserido em um projeto mais amplo. Viptos provê suporte não somente ao desenvolvimento de código, mas também a simulação de ambiente. Viptos integra um simulador *TinyOS* (TOSSIM), um simulador de rede (VisualSense).

XMJ (OGUNSHILE, 2005) e X-Machine Toolkit (ABDUNABI, 2007) são ferramentas baseadas no método formal XM, sendo que a primeira oferece uma interface gráfica não tão favorável ao desenvolvedor, pois o mesmo precisa conhecer o método e sua linguagem para construir suas aplicações. A segunda ferramenta já possui uma interface mais amigável, favorecendo ao desenvolvedor no desenvolvimento de aplicações, bastando apenas ao mesmo, ter conhecimento básico sobre autômatos. Outro ponto a ser ressaltado é que ambas ferramentas, geram código em Java baseado em suas modelagens.

A seguir será apresentado um comparativo dos trabalhos relatados, explicitando características inerentes em cada um deles, além de propor características desejáveis ao *framework* tido como ideal.

3.8 Características desejadas para framework

Com intuito de identificar características mais apropriadas para uma ferramenta de apoio ao desenvolvimento de aplicações RSSF, foi feita uma avaliação dos frameworks apresentados anteriormente, tomando por base alguns parâmetros que servirão de base para avaliação. A Tabela 3.1 apresenta um comparativo entre os trabalhos apresentados de acordo com características desejadas para um framework. Os campos preenchidos por "Sim", indicam que a característica foi contemplada. Os campos preenchidos por "Sim*", indicam que a característica foi contemplada parcialmente. Os campos preenchidos por "Não", indicam que a ferramenta não contempla tal característica. A seguir serão descritos os parâmetros base que servirão para avaliar os trabalhos descritos anteriormente:

- Usabilidade - constata a facilidade de uso da ferramenta pelo usuário.
- Portabilidade - averigua a possibilidade da utilização da ferramenta em outro ambiente de execução.
- Abordagem - refere-se ao tipo de interação com o usuário.
- Geração de código - constata se a ferramenta provê a geração automática de código em *nesC*.
- Plataforma - refere-se onde a ferramenta foi desenvolvida.
- Verificação de código - refere-se a possibilidade de efetuar verificação de corretude no código gerado.

Tabela 3.1: Comparação entre os Frameworks.

	RaPTEX	TinyDT	Viptos	GRATIS	XMJ	X-Machine Toolkit	Este Trabalho
Usabilidade	Sim	Sim	Sim	Sim	Não	Sim	**Sim**
Portabilidade	Sim	Sim	Sim	Sim	Sim	Sim	**Sim**
Abordagem	Blocos	Edição de código	Blocos	Blocos	Edição de código	Autômato XM	**Autômato XM/CXM**
Plataforma	Eclipse	Eclipse	Eclipse	Python	Java	Eclipse	**Eclipse**
Geração de código	Sim*	Não	Sim*	Sim*	Não	Não	**Sim**
Verificação de código	Não	Não	Não	Não	Sim	Sim	**Sim**

A tabela 3.1 reflete os estudos das ferramentas apresentadas neste capítulo, de acordo com as características desejadas para um framework de aplicações RSSF. Pode-se notar que no campo geração de código, RaPTEX, Viptos e GRATIS atendem parcialmente esta característica, pois só geram o arquivo de configuração (configuration) em *nesC*. Para atender completamente a característica, seria necessário gerar o arquivo de módulo (module). Quanto a XMJ e X-Machine Toolkit, estes não contemplam a característica, pois ambos geram código em Java e não tem por foco aplicações RSSF. TinyDT, talvez seja a mais pobre das soluções apresentadas, pois não possue interface gráfica nem tampouco gera código, além de não ter suporte para a versão mais atual da plataforma *TinyOS* 2.x. Um ponto a ser destacado na ferramenta proposta por este trabalho, embora a referida seja uma extensão da X-Machine Toolkit, é o fato de trabalhar com dois modelos do método formal, o XM e o CXM, pois se trata de modelar aplicações como componentes XM e em seguida realizar sua comunicação entre tais componentes. Isto poderá ser melhor entendido no capítulo a seguir.

3.9 Resumo

Este capítulo analisou as ferramentas existentes para construção de aplicações RSSFs, salvo uma delas que faz parte de um projeto maior e por conseguinte possue outros objetivos, porém a ferramenta desejável é baseada na extensão desse trabalho formal. As ferramentas foram avaliadas por meio de vantagens e desvantagens das abordagens, sendo que as características foram extraídas desta análise.

No capítulo seguinte, será abordado o desenvolvimento de uma metodologia para criação e geração de código para RSSF, utilizando-se do método formal CXM. Serão apresentados estudos de caso com intuito de validar o método proposto. Este passo é importante, pois permite identificar parâmetros que servirão de referência para comparação com outras ferramentas.

Capítulo 4

Integrando X-Machine e RSSF

Este capítulo introduz a aplicação do modelo formal proposto CXM para modelagem, verificação e geração de código para RSSF. A abordagem proposta tem por premissa a composição de XMs que se comunica por meio de portas de entrada (*input*) e saída (*output*) de canais de fluxo (*streams*) e associações destes às funções. A seguir será apresentada a abordagem CXM proposta para construção de aplicações RSSF, bem como exemplos de casos de uso para ilustrar o método.

4.1 Construindo Sistemas a partir de X-Machine Independentes

A proposta de construção de sistemas de comunicação por meio de componentes XMs consiste em três etapas, (a) modelar a XM independente das outras partes do sistema, ou ainda utilizar modelos existentes como componentes do novo, (b) determinar a comunicação entre componentes XMs e (c) promover a geração de código através de definições dos modelos expressos em XMDL e em seguida efetuar sua tradução para linguagem XML, e a partir dela para outra linguagem de programação desejada. Esta abordagem permite ao desenvolvedor criar modelos a partir da composição de componentes XMs utilizando-se de uma metodologia *bottom-up* (Figura 4.1).

A abordagem apresenta vantagens ao desenvolvedor como (i) re-utilização de modelos existentes; (ii) as atividades de modelagem e comunicação são abordagens distintas, permitindo a decomposição de complexidade por meio de componentes XMs, o que torna mais fácil o seu entendimento e conseqüentemente sua modelagem; (iii) componentes; e (iv) componentes podem ser verificados em separado, além de poder usufruir de ferramentas construídas por abordagens XM ou CXM. A seguir serão apresentadas as etapas para construção das aplicações.

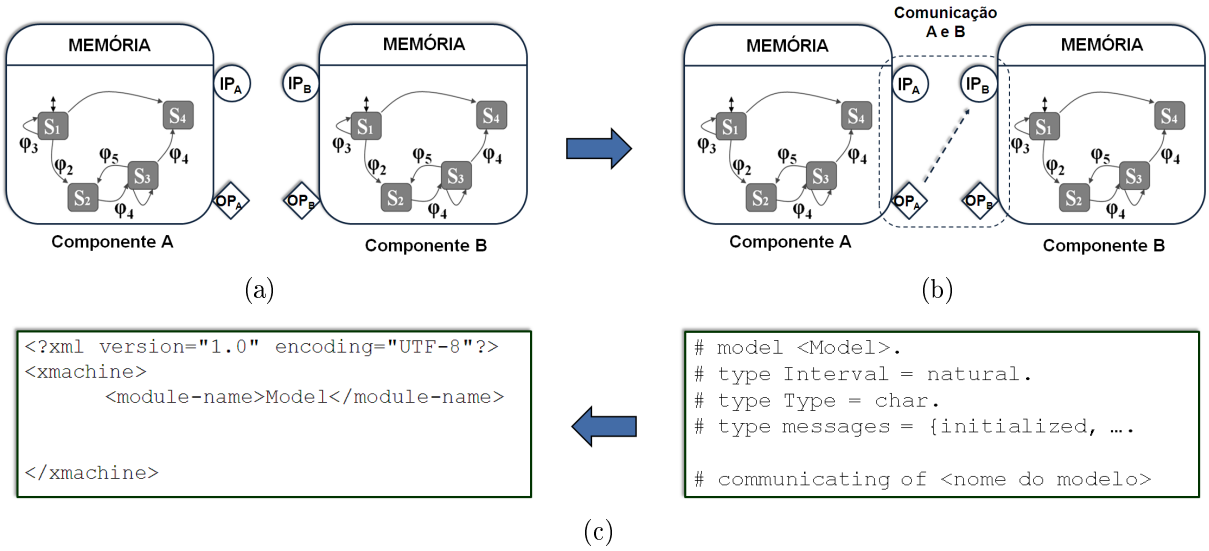


Figura 4.1: Processo de construção de aplicações RSSF.

4.1.1 Modelando um Componente X-Machine

Nesta etapa, são modelados os comportamentos de um sistema por meio de componentes XMs em separado, de modo a realizar tal operação independente do todo (sistema). Esta abordagem permite o particionamento do sistema, favorecendo um melhor entendimento e refino na modelagem. A ideia principal é o desenvolvimento orientado a componentes XMs, para que na etapa seguinte seja apenas modelada a comunicação com os demais componentes XMs.

A aplicação *Blink*[1] servirá como referência inicial para demonstração da metodologia proposta neste trabalho. Sendo que a referida é composta por dois componentes: um módulo chamado BlinkM.nc, e um configuration Blink.nc (Figura 4.2). Vale ressaltar, que todas as aplicações requerem um arquivo de configuração no nível mais alto, e que tipicamente possue o nome da própria aplicação. A aplicação pode ser encontrada no caminho "apps/Blink"da árvore do *TinyOS*.

[1] *Blink* é uma aplicação básica do *TinyOS* que faz o LED vermelho piscar a uma freqüência de 1Hz.

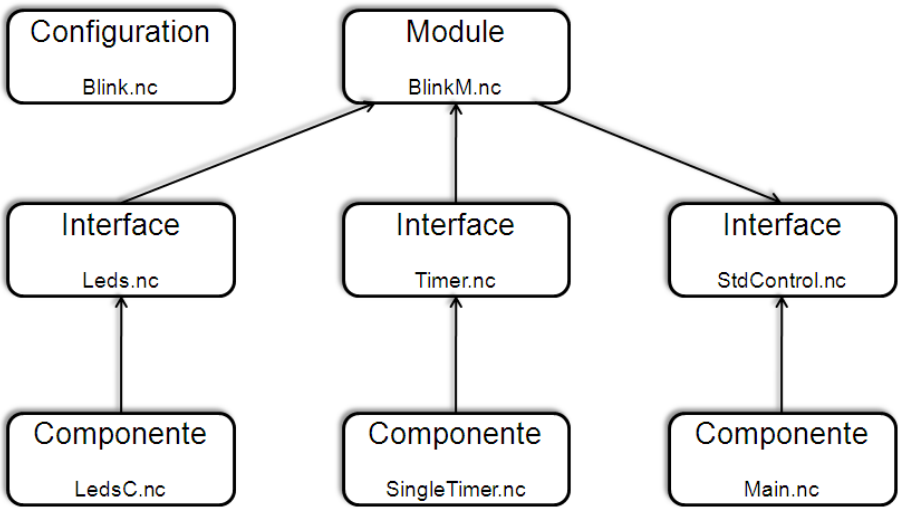


Figura 4.2: Esquema da aplicação *Blink*.

Inicialmente a aplicação é modelada por meio de componentes XMs, essa prática possibilita a modularização do sistema, com o objetivo de facilitar o desenvolvimento. Nesta fase é possível ter uma melhor compreensão dos requisitos, o nível de abstração é bem maior favorecendo a modelagem. Conforme ilustrado na Figura 4.2, pode-se perceber que o *Module Blink* é composto por dois componentes: *LedsC* e *SingleTimer*. Em seguida, a modelagem do estudo de caso do componente XM *LedsC* e *SingleTimer*. O componente em questão modela o funcionamento de Led (diodo emissor de luz) em seus dois estados: aceso e apagado. Para este estudo, será adotado o *led* de cor vermelha, os estados correspondentes e suas respectivas funções de transição redOn que sai do estado *Off* para o estado *On* e a transição *redOff* que sai do estado *On* para o estado *Off*. A Figura 4.3 mostra os modelos expressos em forma de componente XM, onde o componente *SingleTimer* implementa um temporizador e é responsável pela especificação do tipo temporizador, bem como de seu intervalo que é expresso em milisegundos. Neste exemplo, o tipo de temporizador é o TIMER_REPEAT e seu intervalo é de 1000, o temporizador continua até que o *comando stop()* seja acionado.

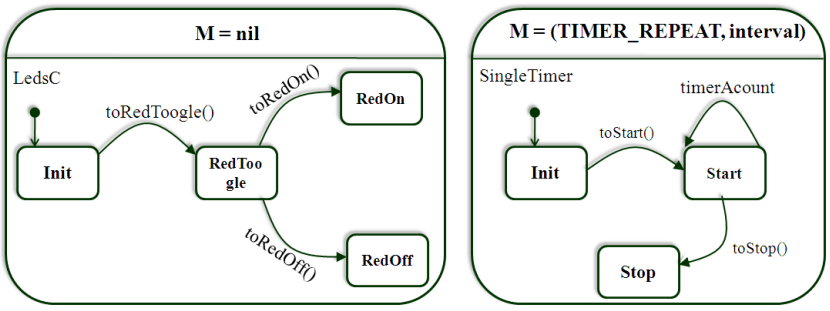


Figura 4.3: Esquema da aplicação *Blink*.

Em uma aplicação em *TinyOS*, o primeiro componente a ser executado é o *Main* por meio do comando init(), seguido pelo comando *start*(). Uma aplicação *TinyOS* precisa ter um componente *Main* em sua configuração. StdControl é uma interface que inicializa componentes *TinyOS* e define três comandos: a) init() - é chamado para iniciar um componente; b) *start*() - é chamado para o componente ser executado pela primeira vez; e c) stop() - é chamado para parar um componente. O modelo *XmBlink* resultante pode ser visto como componente XM na Figura 4.4 e apresenta os três estados descritos anteriormente: init, *start* e stop. Na seção seguinte serão descritos a comunicação entre os componentes e o código XMDL que expressa o modelo.

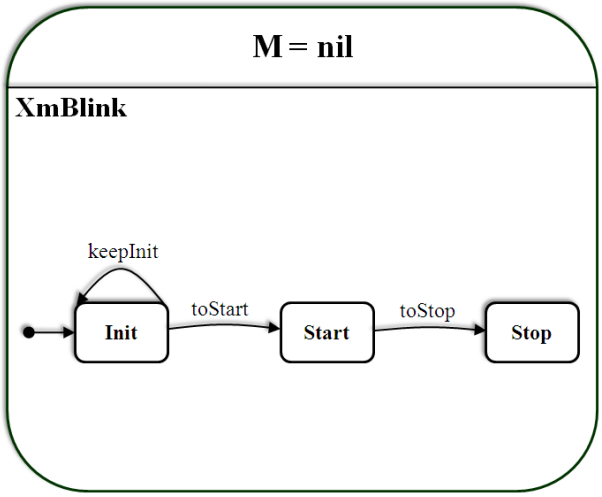


Figura 4.4: Aplicação *XmBlink* em XM.

4.1.2 Comunicação entre Componentes X-Machines

Esta etapa propõe a integração entre os componentes XMs que expressam o sistema modelado. Esta integração é dada através de comunicação entre os componentes especificados anteriormente. O modelo resultante é apresentado na Figura 4.5, explicitando a comunicação entre os componentes, sendo que é possível notar as funções de escrita (saída) caracterizadas pelo símbolo losango cheio, bem como as funções de leitura (entrada) caracterizadas pelo símbolo de círculo cheio.

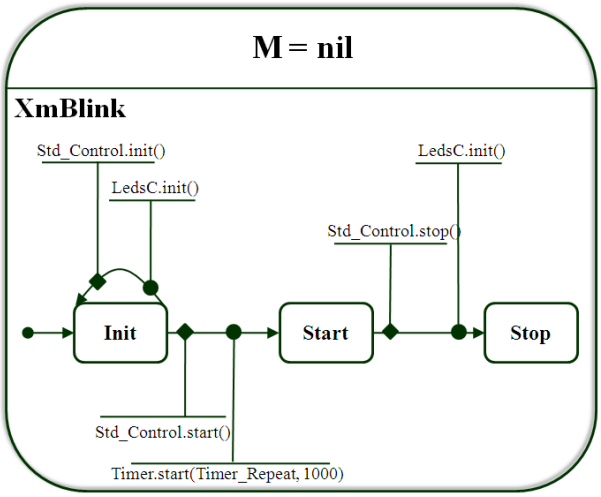


Figura 4.5: Aplicação *XmBlink* descrita através da comunicação de componentes XM.

O modelo pode ser expresso através da linguagem de notação matemática XMDL, o Código 4.1 descreve o modelo *XmBlink* e a Figura 4.6 ilustra a representação diagramática da aplicação em questão.

```
1 #model XmBlink.
2 #type Interval = natural.
3 #type Type = char.
4 #type messages = {initialized, started, stoped}.
5 #states = {Init, Start, Stop}.
6 #memory nil.
7 #type event = {fired}.
8 #input (Type, Interval).
9 #output (messages).
10 #fun start ((start), (?Type, ?Interval)) =
11       if ?Type == TIMER_REPEAT then (nil, (message)).
12 #fun stop ((stop), (nil)) = ((stop), nil).
13 #transition (Init, init) = Init.
14 #transition (Init, start) = Start.
15 #transition (Start, stop) = Stop.
```

Código 4.1: *XmBlink* expresso em XMDL.

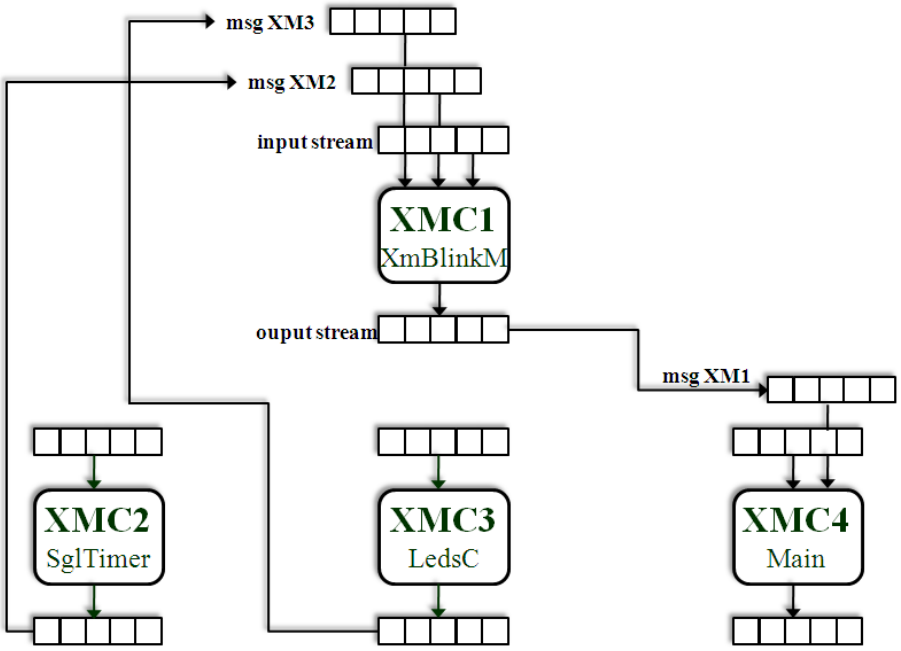


Figura 4.6: Representação diagramática do *Blink*.

A sintaxe para expressar a comunicação entre os componentes XM expressa em XMDL é dada seguindo a sintaxe do Código 4.2.

```
1 #communicating of (nome modelo):
2 (nome função) reads from (nome modelo)
3 (nome função) writes (messagem) to (nome modelo)
4 [where (expressão) from (memory | input | output)(tupla)]
```

Código 4.2: Sintaxe da comunicação em XMDL.

A seguir serão apresentados os códigos referentes à comunicação entre componentes da aplicação, *XmBlink* (Código 4.3), *Main* (Código 4.4), *SingleTimer* (Código 4.5) e *LedsC* (Código 4.6).

```
1 #communicating of XmBlink:
2  StdControl.init() writes (init()) to Main.
3     Leds.init() reads from LedsC.
4  StdControl.start writes (start()) to Main.
5     Timer.start() reads from SingleTimer.
6  StdControl.stop writes (stop()) to Main.
7     Timer.stop() reads from TimerC.
```

Código 4.3: Comunicação *XmBlink* com outros componentes.

```
1 #communicating of LedsC:
2  LedsC.init() writes (StdControl.init()) to
```

```
3       XmBlinkM.
4  LedsC.redToggle() writes (Timer.fired()) to
5       XmBlinkM.
```

Código 4.4: Comunicação do component *LedsC*.

```
1 #communicating of SingleTimer:
2 StdControl.start() writes
3       (Timer.start(TIMER_REPEAT, 1000)) to
4       XmBlinkM.
5  StdControl.stop() writes
6       (Timer.stop()) to XmBlinkM.
```

Código 4.5: Comunicação do componente *SingleTimer*.

Concluída a etapa de modelagem dos sistemas de comunicação onde são construídos os componentes XM e suas configurações expressas em XMDL, o próximo passo é a geração de código da aplicação.

4.1.3 Geração de Código

A geração de código consiste em duas etapas de tradução: a) XMDL para XML e b) XML para *nesC*. A tradução intermediária em XML se deve ao fato de ser uma linguagem que possibilita a tradução para outras linguagens como, por exemplo, Java, C, dentre outras. Dessa forma, este procedimento permitirá uma grande quantidade de oportunidade de migração do sistema modelado, a seguir será explicitado o processo de tradução dessas etapas.

4.1.3.1 XMDL para XML

A partir da descrição do modelo em XMDL é aplicado o método de tradução para XML proposto por Kefalas e estendido por Ogunshile (2005) Ogunshile (2005), o Código 4.6 ilustra parte desse procedimento, o código completo será apresentado em anexo no trabalho.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xmachine>
3     <module-name>XmBlink</module-name>
4
5         <type>
6       <name>Interval</name>
```

```
7       <type-definition>
8         <built-in-type>natural</built-in-type>
9       </type-definition>
10     </type>
11
12     <type>
13       <name>Type</name>
14       <type-definition>
15         <built-in-type>char</built-in-type>
16       </type-definition>
17     </type>
18         ...
19 </xmachine>
```

Código 4.6: Aplicação *XmBlink* em XMDL.

O código XML resultante desta operação, permite a tradução para outras linguagens de programação, sendo que *nesC* (plataforna *TinyOS*) é a linguagem escolhida para validar este trabalho, por se tratar de uma das linguagens mais utilizadas na área de RSSF, conforme visto no Capítulo 2.

4.1.3.2 XML para *nesC*

8 A tradução do código XML para *nesC* obedece algumas características provenientes da linguagem em questão. A aplicação MyBlink é composta por dois componentes: arquivos de configuração (configuration) e arquivos de módulo (module). Configuração é responsável pelas conexões entre componentes, além de especificar as interfaces que usam e provêem. Módulo é onde está a definição da implementação da aplicação, é designado por: nome da aplicação + M.nc, no caso estudado, XmBlinkM.nc, sendo que o sufixo nc refere-se à linguagem de programação *nesC*. Essa divisão de configuração e módulo possibilita ao programador o desenvolvimento mais ágil de aplicações por meio de conexão de módulos (Ruiz et al, 2004).

a) Arquivo de configuração - A criação do arquivo de configuração da aplicação *XmBlink* é dada pelos componentes que são utilizados na aplicação, neste caso, *XmBlinkM*, *Main*, *LedsC* e *SingleTimer*. Esta informação pode ser extraída do modelo através das informações de comunicação XM: *communicating of XmBlink*, *communicating of LedsC*, *communicating of SingleTimer*. O componente *Main*, faz parte de uma aplicação *TinyOS*, sendo que é o primeiro a ser executado conforme mencionado anteriormente.

A tradução da especificação XML para linguagem *nesC* segue os seguintes procedimentos:

- O nome do arquivo de configuração é extraído da tag <module-name> expressa no código: <module-name>XmBlink</module-name>;
- Os componentes que serão utilizados na aplicação são extraídos dos outros modelos/componentes (máquinas) aliados com o componente *Main* (sempre usado);
- As ligações entre os componentes e seus respectivos métodos podem ser extraídos das tags XML expressas nos modelos de comunicação: *communicating of XmBlink*, *communicating of LedC*, *communicating of SingleTimer*, também aliado ao *StdControl* (que referencia *Main*). A interface *StdControl* é usada para inicializar e as aplicações *TinyOS*.

  O código *nesC* gerado a partir dos procedimentos descritos anteriormente é apresentado no Código 4.7.

```
1 configuration XmBlink {
2 }
3 implementation {
4   components Main, XmBlinkM, SingleTimer, LedsC;
5   Main.StdControl -> XmBlinkM.StdControl;
6   Main.StdControl -> SingleTimer.StdControl;
7   XmBlinkM.Timer -> SingleTimer.Timer;
8   XmBlinkM.Leds -> LedsC;
9 }
```

Código 4.7: Código de configuração.

b) Arquivo módulo - é onde estão implementadas as especificações do arquivo de configuração XmBlink.nc. A tradução da especificação em XML para *nesC* segue os seguintes procedimentos:

- Da mesma forma que o arquivo de configuração, o nome é extraído da tag XML <module-name>;
- A interface StdControl possui três métodos: *init()*, *start()* e *stop()*. Dessa forma grande maioria das aplicações terão que implementar estes métodos, e de acordo com as ligações fornecidas no modelo de comunicação, serão descritos os componentes e as interfaces que os implementam;

- As interfaces que o módulo usa e provê são retirados dos modelos de comunicação XM da seguinte forma, para aqueles que lêem a partir do componente XM (reads) referenciam as interfaces que serão usadas (uses). As que têm a clausula writes a partir do componente XM, serão as que serão providas (cláusula provides);
- Os comandos são implementados pelas interfaces e são descritos nos modelos de comunicação, o Código 4.8 reflete os procedimentos relatados.

A descrição do processo de obtenção do código em *nesC* pode ser observado através do diagrama (Figura 4.7), onde pode-se notar o mapeamento das aplicações descritas em XM, sendo postos através de componentes que se comunicam para modelagem do sistema. Em seguida é aplicado o método de tradução para o XML, para em seguida ser traduzido para a linguagem *nesC*.

```
1 module XMBlinkM {
2   provides {
3     interface StdControl;
4   }
5   uses {
6     interface Timer;
7     interface Leds;
8   }
9 }
10 implementation {
11
12   command result_t StdControl.init() {
13     call Leds.init();
14     return SUCCESS;
15   }
16
17   command result_t StdControl.start() {
18     // Start a repeating timer that fires every 1000ms
19     return call Timer.start(TIMER_REPEAT, 1000);
20   }
21
22   command result_t StdControl.stop() {
23     return call Timer.stop();
24   }
25
26   event result_t Timer.fired()
27   {
28     call Leds.redToggle();
```

```
29        return SUCCESS;
30    }
31 }
```

Código 4.8: Código do Módulo *XmBlink*.

O processo de geração de código se inicia por meio de componentes XM modelados e expressos em XMDL. A verificação e os testes podem ser feitos neste momento. O passo seguinte é a modelagem da comunicação entre componentes, neste momento são construídos os relacionamentos por meio de funções de transições entre os componentes. De posse do modelo gerado (componentes + comunicação) em XMDL, a tradução do código XMDL para XML é realizada na terceita etapa. Por fim, a tradução para a linguagem escolhida, nesta pesquisa, a linguagem *nesC*.

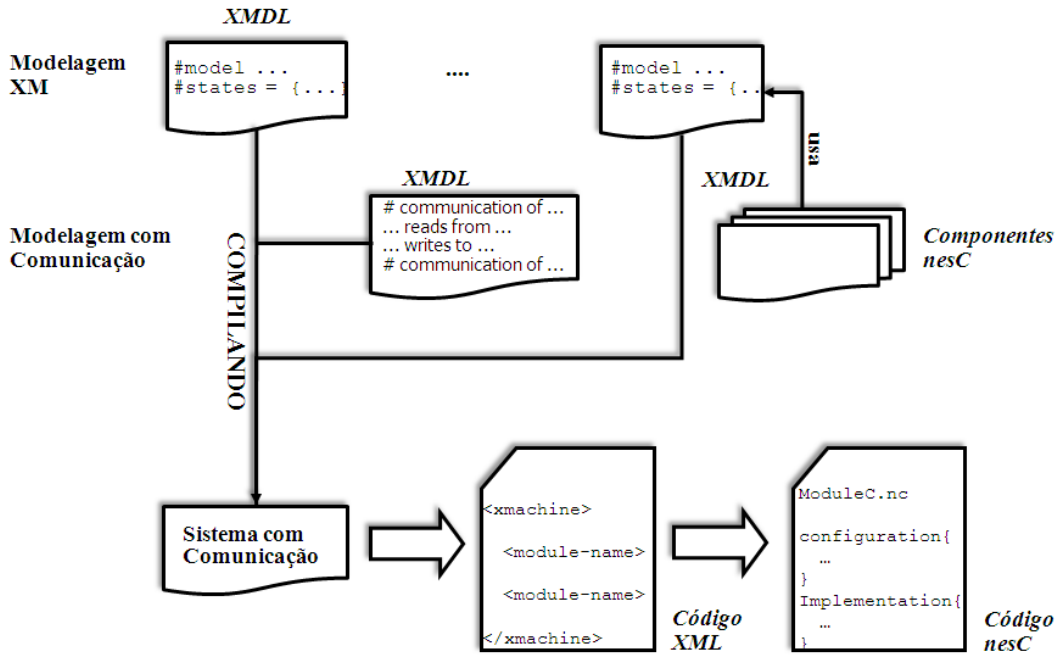


Figura 4.7: Processo de geração de código *nesC*.

4.2 Casos de Uso

Nesta seção são apresentados dois casos de uso que podem ser aproveitados em diversos tipos de aplicações. O caso de uso proposto é voltado para área médica, por meio de um sistema de monitoramento de pacientes. O referido caso de uso faz parte de um domínio de possíveis aplicações em RSSF, que é denominado de Rede de Sensores para o monitoramento do Corpo Humano (RSCH). A utilização das RSCHs vem a ser

uma solução inovadora e de forte impacto social, pois é possível fazer o acompanhamento do paciente em qualquer lugar e em qualquer momento, bastando ter a infraestrutura adequada para tal (internet, notebook com comunicação wireless, smartphone, dentre outros).

A conseqüência da utilização dessa tecnologia é a economia gerada, pois possibilitará diminuir o número de internações e procedimentos ambulatoriais desnecessários devido à disponibilidade dos dados do paciente em qualquer tempo e espaço. Outro ganho é a diminuição de erros médicos devido à interação proporcionada ao profissional da área (BARBOSA, 2008 apud LYMBERIS; DITTMAR, 2007). O cenário proposto trata da integração de sensores que atuarão no corpo do paciente e no ambiente onde o mesmo se encontra, sendo, portanto, dividido em sensores corporais e sensores ambientais respectivamente. Nos sensores corporais são monitorados a temperatura do paciente, e nos sensores ambientais, são monitorados umidade e temperatura, sendo que será aproveitado o mesmo modelo de temperatura usada no corporal, porém em contexto diferente. A Figura 4.8 ilustra o caso de uso proposto com seus sensores corporais e ambientais descritos anteriormente, os modelos diagramáticos e códigos XMDL, bem como as comunicações entre os componentes da aplicação serão apresentados a seguir, servindo de base para a geração de código em *nesC* meta deste trabalho.

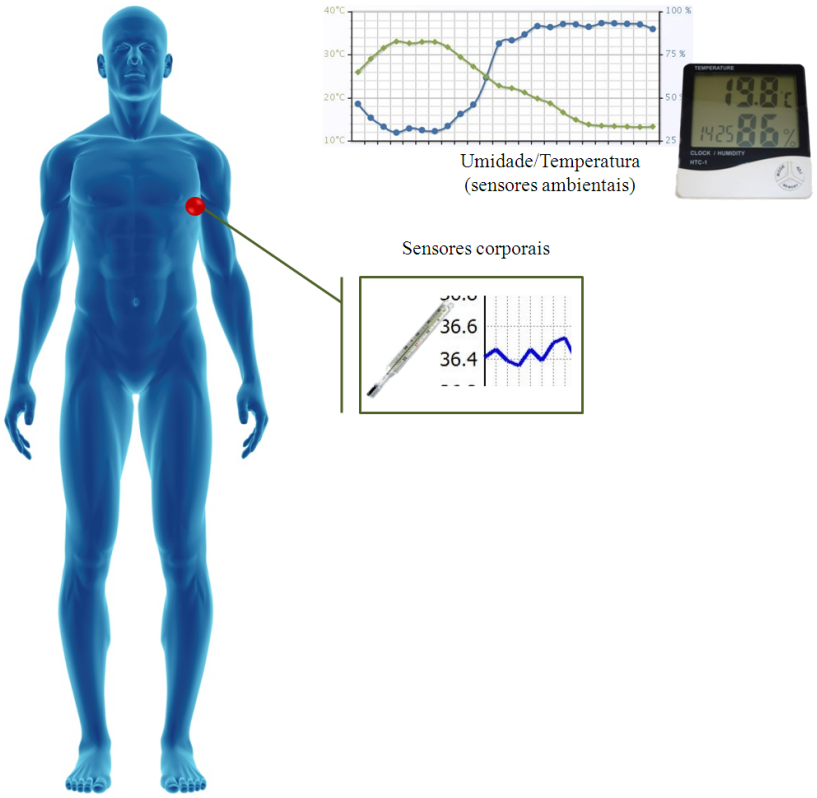


Figura 4.8: Aplicação RSCH - sensores ambientais e corporais.

4.2.1 Sensoriamento de Umidade

Esta aplicação faz parte do sensoriamento ambiental do estudo de caso proposto e é responsável por medir a umidade do ambiente onde os pacientes estão alojados. A aplicação é uma adaptação de *XmBlink* apresentada anteriormente, porém com a inserção de um componente de sensoriamento, neste caso, o sensor de umidade. A aplicação é composta por quatro componentes, o componente *Main* que compõe todas as aplicações no *TinyOS*, o componente Timer, o componente *LedsC* que controla o funcionamento dos leds em cada valor lido pelo sensor e a aplicação proposta, *XmUmidade*.

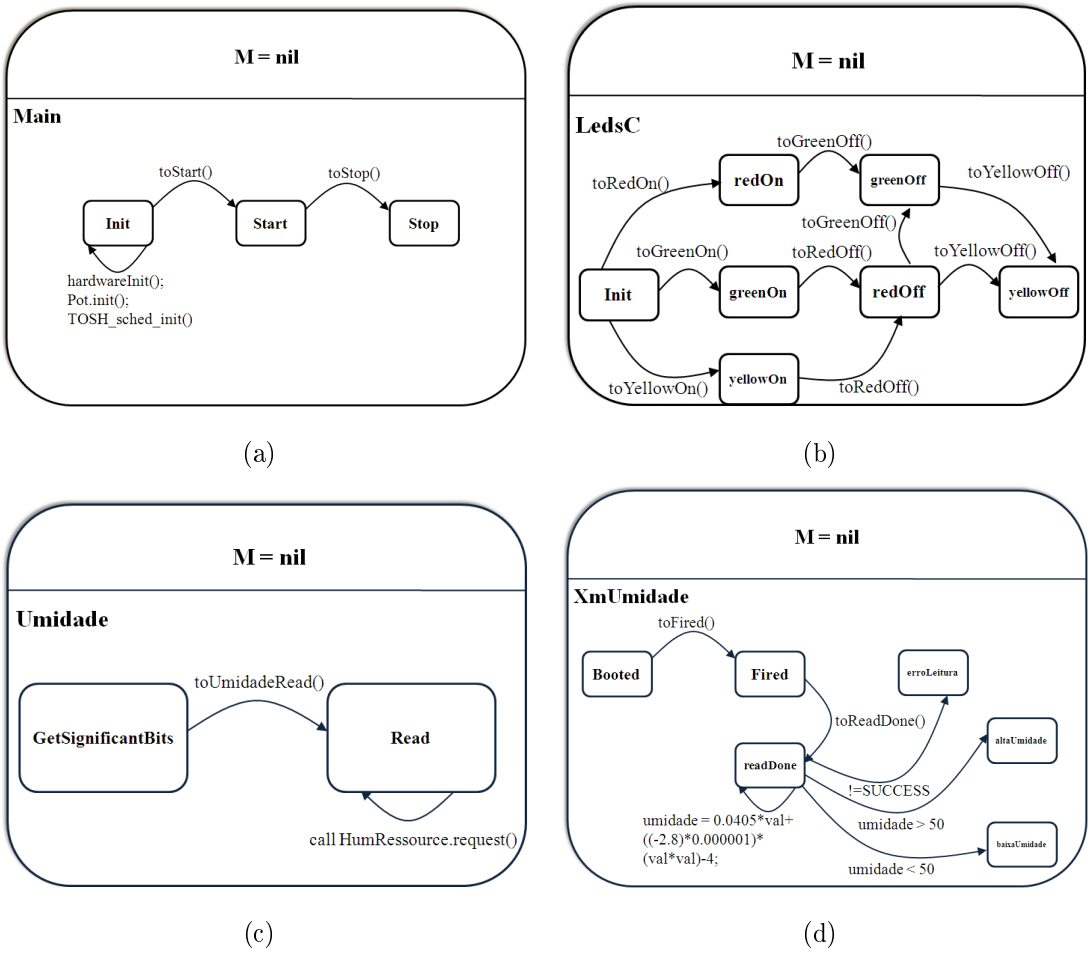


Figura 4.9: Componentes que formam a aplicação *XmUmidade*.

A Figura 4.9 apresenta a modelagem diagramática da aplicação expressa em CXM, e seus respectivos componentes, (a) *Main* presente em todos os códigos das aplicações em *nesC*, sendo que para este caso só foi modelado uma parte do mesmo, a de inicialização do hardware (sensor); (b) *LedsC* responsável pelo funcionamento dos leds de acordo com suas ações, neste caso cada cor referencia-se a uma ação da aplicação, ou seja, *led* verde indica

umidade alta, *led* amarelo indica umidade baixa e *led* vermelho indica erro na leitura do sensor; O terceiro componente é (c) Umidade que referencia o sensor de mesmo nome; e (d) *XmUmidade*, que representa a aplicação proposta, onde pode-se notar o cálculo da umidade, bem como das ações que cada *led* vai assumir.

A comunicação entre os componentes da aplicação pode ser vista na Figura 4.10 de onde são explicitadas as leituras e escritas no componente principal da aplicação conforme técnica referenciada anteriormente.

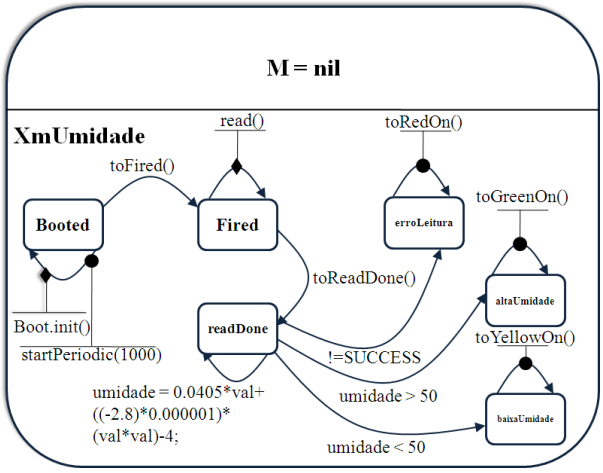


Figura 4.10: Comunicação entre os componentes na aplicação *XmUmidade*.

Os códigos a seguir descrevem a aplicação em XMDL e suas respectivas comunicações com os demais componentes. O Código 4.9 refere-se à aplicação principal *XmUmidade*, sua comunicação com os demais componentes é mostrado no Código 4.10. Os códigos 4.11, 4.12 e 4.13 referem-se a comunicação dos componentes *LedsC*, *SenseTimer* e Umidade respectivamente.

```
1 #model XmUmidade.
2 #type Interval = natural.
3 #type Type = char.
4 #type messages = {booted, fired, stoped, readDone,!=SUCCESS, umidade>50,
      umidade<50}.
5 #states = {Booted, Fired, ReadDone, erroLeitura, AltaUmidade,
     BaixaUmidade}.
6 #memory nil.
7 #init_state {Booted}.
8 #type event = {booted, fired, readDone}.
9 #input (Type, Interval).
10 #output (messages).
11 #fun booted ((booted), (?Type, ?Interval)) =
12       if ?Type == StartPeriodic then (1000, (message)).
```

```
13 #fun fired ((fired), (?Type, ?Interval)) =
14       if ?Type == fired then (nil, (readUmidade)).
15 #fun readumidade ((readDone), ((?Type, uint16_t val)) =
16       if ?Type == error_t_result then
17          ?umidade <-0.0405*?val+((-2.8)*0.000001)*
18                    (?val*?val)-4.
19       if ?umidade != SUCCESS then redOn().
20       if ?umidade >50 then call greenOn().
21       if ?umidade <50 then call yellowOn().
22 #transition (Booted, Fired) = Fired.
23 #transition (Fired, ReadDone) = ReadDone.
24 #transition (ReadDone, ErroLeitura) = ErroLeitura.
25 #transition (ReadDone, Umidade>50) = AltaUmidade.
26 #transition (ReadDone, Umidade<50) = BaixaUmidade.
```

Código 4.9: *XmUmidade* em XMDL.

```
1 #communicating of XmUmidade:
2  Boot.booted() writes (booted()) to Main.
3     SenseTimer.startPeriodic(1000) reads from SenseTimer.
4  SenseTimer.fired() writes (start()) to SenseTimer.
5     ReadUmidade.read() reads from ReadUmidade.
6  ReadUmidade.readDone() writes (read()) to Umidade.
7     Umidade.read() reads from Umidade.
```

Código 4.10: Comunicação *XmUmidade*.

```
1 #communicating of LedsC:
2  LedsC.redOn writes (readDone()) to
3     XmUmidadeM.
4 LedsC.greenOn writes (readDone()) to
5     XmUmidadeM.
6 LedsC.yellowOn writes (readDone()) to
7     XmUmidadeM.
```

Código 4.11: Comunicação *LedsC*.

```
1 #communicating of SenseTimer:
2 Boot.booted() writes
3     (SenseTimer.startPeriodic(1000)) to
4     XmBlinkM.
```

Código 4.12: Comunicação *SenseTimer*.

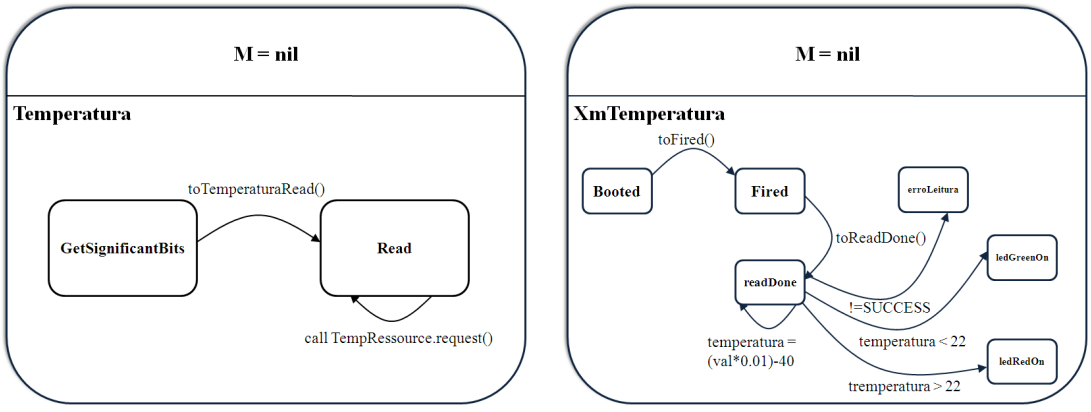
```
1 #communicating of Umidade:
2 Umidade.read() writes
3       (read()) to XmUmidadeM.
```

Código 4.13: Comunicação Umidade.

O passo seguinte é a tradução de XMDL em XML e em seguida *nesC* gerando os arquivos de configuração e módulo, conforme método estudado na seção anterior. Os códigos XML e *nesC* gerados serão apresentados nos capítulos a seguir.

4.2.2 Sensoriamento de Temperatura

A aplicação Temperatura é uma adaptação da modelagem anterior (Umidade) com uma diferença, a troca do componente de sensoriamento, temperatura no lugar do sensor de umidade. Portanto, a aplicação possue os mesmos componentes descritos anteriormente com a ressalva do sensor em questão. A Figura 4.11 apresenta a modelagem diagramática da aplicação em questão, onde o componente (a) Temperatura referencia o sensoriamento da temperatura, sendo que foi modelado somente parte do mesmo. O componente (b) *XmTemperatura*, reflete a aplicação proposta onde é expresso o cálculo da temperatura expressa em graus Celsius, bem como suas ações de acordo com o valor calculado, tomando como base a temperatura de 22° C.



(a) (b)

Figura 4.11: Componentes da aplicação *XmTemperatura*.

A comunicação entre os componentes da aplicação poderá ser visualizado na Figura 4.12, onde são mostradas as interrações, leituras e escritas, do componente principal com os demais.

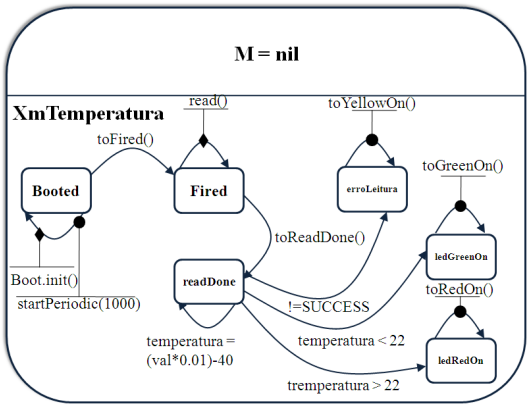


Figura 4.12: Comunicação entre os componentes da aplicação *XmTemperatura*.

A seguir serão apresentados os códigos em XMDL que descrevem a aplicação, bem como os códigos que representam a comunicação entre componentes. Os códigos 4.14 e 4.15 descrevem a aplicação principal e sua comunicação com os demais componentes respectivamente. O demais códigos, 4.16, 4.17 e 4.18 refletem a comunicação dos componentes *LedsC*, *SenseTimer* e Umidade respectivamente.

```
1 #model XmTemperatura.
2 #type Interval = natural.
3 #type Type = char.
4 #type messages = {booted, fired, stoped, readDone, !=SUCCESS,
      temperature<22, temperature>22}.
5 #states = {Booted, Fired, ReadDone, erroLeitura, AltaTemperatura,
      BaixaTemperatura}.
6 #memory nil.
7 #init_state {Booted}.
8 #type event = {booted, fired, readDone}.
9 #input (Type, Interval).
10 #output (messages).
11 #fun booted ((booted), (?Type, ?Interval)) =
12       if ?Type == StartPeriodic then (1000, (message)).
13 #fun fired ((fired), (?Type, ?Interval)) =
14       if ?Type == fired then (nil, (readTemperatura)).
15 #fun readtemperatura ((readDone),
16       ((?Type, uint16_t val)) =
17       if ?Type == error_t_result then
18          ?temperatura <- (?val*(0.01)-40.
19       if ?temperatura != SUCCESS then call yellowOn().
20       if ?temperatura < 22 then call greenOn().
21       if ?temperatura > 22 then call redOn().
22 #transition (Booted, Fired) = Fired.
```

```
23 #transition (Fired, ReadDone) = ReadDone.
24 #transition (ReadDone, !=SUCCESS) = ErroLeitura.
25 #transition (ReadDone, temperatura<22) = ledGreenOn.
26 #transition (ReadDone, temperatura>22) = ledRedOn.
```

Código 4.14: *XmTemperatura* em XMDL.

```
1 #communicating of XmTemperatura:
2  Boot.booted() writes (booted()) to Main.
3     SenseTimer.startPeriodic(1000) reads from SenseTimer.
4  SenseTimer.fired() writes (start()) to SenseTimer.
5     ReadTemperatura.read() reads from ReadTemperatura.
6  ReadTemperatur.readDone() writes (read()) to
7     Temperatura.
8     Temperatura.read() reads from Temperatura.
```

Código 4.15: Comunicação *XmTemperatura*.

```
1 #communicating of LedsC:
2 LedsC.redOn writes (readDone()) to
3     XmTemperaturaM.
4 LedsC.greenOn writes (readDone()) to
5     XmTemperaturaM.
6 LedsC.yellowOn writes (readDone()) to
7     XmTemperaturaM.
```

Código 4.16: Comunicação *LedsC*.

```
1 #communicating of SenseTimer:
2 Boot.booted() writes
3     (SenseTimer.startPeriodic(1000)) to
4     XmTemperaturaM.
```

Código 4.17: Comunicação *SenseTimer*.

```
1 #communicating of Temperatura:
2 Temperatura.read() writes
3     (read()) to XmTemperaturaM.
```

Código 4.18: Comunicação Temperatura.

De posse destes artefatos gerados na modelagem dos estudos de caso apresentados, basta aplicar os passos de tradução dos códigos XMDL para XML e em seguida para *nesC*, conforme apresentado seção 4.1.

4.3 Resumo

Neste Capítulo estudou-se a integração de um método formal para construção de aplicações RSSF, que resultou em um método que acompanha o desenvolvedor desde a modelagem até a geração de código. Para isto, foram propostos casos de uso para ilustrar a aplicação do método apresentados na criação de aplicações RSSF. Os casos de uso foram: (a) *XmBlink*, aplicação que controla a freqüência com que o *led* pisca; (b) *XmUmidade* e (c) *XmTemperatura*, ambos fazendo parte de uma aplicação de monitoramento de pacientes em um hospital.

Os casos de uso apresentados neste capítulo serão utilizados nos capítulos seguintes, onde será utilizada a ferramenta *Ufam Sensor CommX* na construção e geração de código para RSSF, de modo a testar e validar a mesma. Também serão apresentados os código e as visões destes, a fim de favorecer ao desenvolvedor melhor compreensão do problema.

Capítulo 5
Automatizando a Geração de Código

Neste capítulo é apresentado o processo de desenvolvimento do *plug-in* utilizando o método formal CXM proposto para a plataforma Eclipse. O presente capítulo aborda tecnologias e conceitos utilizados na construção da ferramenta, bem como a arquitetura utilizada. Outro ponto destacado são as características principais do *plug-in*, modelo gerador de código descrito em XSD, além da arquitetura empregada no projeto.

5.1 Arquitetura do toolkit

Eclipse é uma plataforma de desenvolvimento integrado (IDE - *Integrated Development Environment*) criada pela IBM em 2001 e que hoje é administrada pela comunidade open-source com intuito de construir uma plataforma de desenvolvimento aberta e extensível que auxilie no desenvolvimento e gerenciamento de *software* em todo seu ciclo de vida (ECLIPSE, 2008). Atualmente, o Eclipse possui uma extensa comunidade de colaboradores, desde desenvolvedores individuais até grandes organizações como: Oracle, Borland, Red Hat, entre outras. A característica principal do Eclipse é sua arquitetura baseada em *plug-ins*, onde se pode agregar um pequeno conjunto de serviços por meio de um componente e trabalhar em conjunto com os demais. Existem dois *plug-ins* que são a base da plataforma Eclipse, pois provêem pontos de extensão para a maioria dos *plug-ins* nativos, são eles: Workspace e Workbench (Figura 5.1). O primeiro permite a interação do usuário com os recursos da plataforma como arquivos e projetos. Já o segundo, permite que outros *plug-ins* estendam a interface do Eclipse com menus, barras de ferramentas, criação de janelas, entre outros (FERREIRA, 2009).

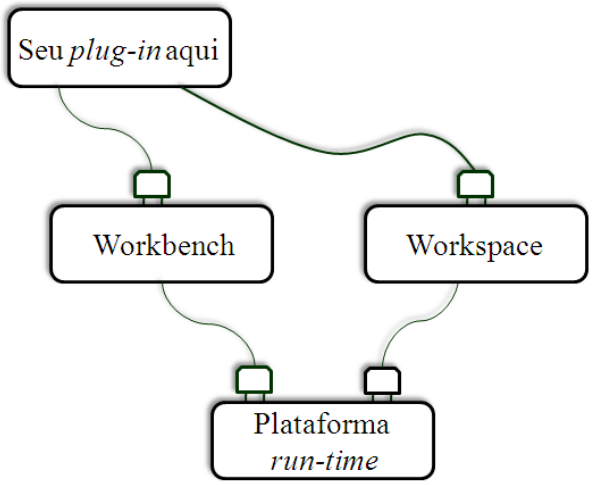


Figura 5.1: *Plug-ins* essenciais do Eclipse (adaptado de Gallardo (2002)).

Em uma visão mais macro, a plataforma Eclipse pode ser dividida em dois módulos: módulo Core e o módulo Interface do Usuário. O módulo Core examina os *plug-ins* instalados e os carrega para serem utilizados no sistema quando o Eclipse é inicializado, porém isso só é feito se os mesmos estiverem configurados. O Core é composto pelo runtime e workspace, sendo que o runtime é responsável pela inicialização, funcionamento e gerenciamento dos *plug-ins* dentro da plataforma. O workspace, como dito anteriormente, administra os recursos do usuário.

O módulo Interface do Usuário é responsável pela entrada e saída de dados e é dividido em três grupos: workbench, JFace e SWT. Devido a sua arquitetura (Core e interface), possibilita o uso dos seus recursos por duas ferramentas focadas em desenvolvimento: PDE (Plug-in Development Enviroment) e JDT (Java Development Tool) (Figura 5.2). O PDE é um *plug-in* nativo e corresponde ao conjunto de ferramentas para desenvolvimento de *plug-ins* para o Eclipse, sendo composto por três componentes: a) PDE UI - agrega ferramentas de criação, desenvolvimento, testes e depuração de *plug-ins*, é o principal componente do PDE; b) PDE Build - automatiza as funções de compilação e execução, isto é, montagem do *plug-in*; c) PDE API Tools - componente que disponibiliza o acesso aos recursos do PDE para os desenvolvedores de *plug-ins*.

Pode-se notar que PDE provê recursos e facilidades específicas para a criação de *plug-ins*, o arquivo manifesto plugin.xml é o principal recurso utilizado neste processo. É através deste arquivo que o usuário pode especificar dependência entre *plug-ins*, especificar o classpath dos projetos e gerar os builds de um *plug-in* em desenvolvimento, em outras palavras, ele é a base para o desenvolvimento de componentes do Eclipse (FONSECA, 2008).

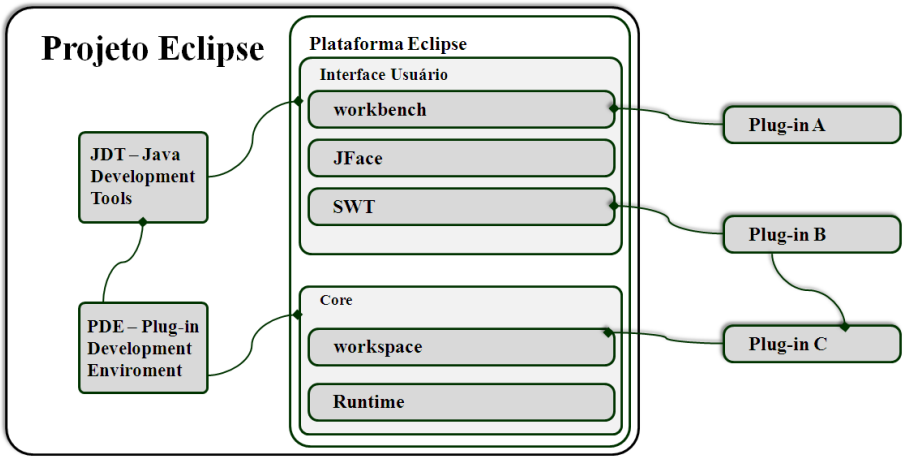


Figura 5.2: Arquitetura de um *plug-in* Eclipse (adaptado de Ferreira (2009)).

Para a geração de código, a tecnologia MDD é empregada, conforme visto no capítulo 3, sendo que resultados relevantes têm sido obtidos pela comunidade open source Eclipse, mais precisamente dentro de um projeto denominado Modeling. O projeto em questão, visa à evolução e a promoção da tecnologia MDD, a partir da unificação de ferramentas, frameworks e padrões de implementação (FERREIRA, 2009). O objetivo do projeto Modeling é concentrar todos os projetos relacionados com a tecnologia de modo a promover a colaboração, unificação e a interoperabilidade entre os projetos (ECLIPSE, 2008). Alguns projetos participantes são listados a seguir (FERREIRA, 2009): a) Eclipse Modeling Framework (EMF) - framework para geração de código a partir de modelos; b) Graphical Editing Framework (GEF) - framework para criação de editores gráficos genéricos; c) Graphical Modeling Framework (GMF) - framework para geração de editores gráficos baseado em modelos de domínio, utilizando EMF e GEF. Na seção seguinte são descritos os *plug-ins* que serão utilizados para construção da ferramenta proposta.

5.1.1 Plug-ins Utilizados

Nesta seção são descritos os *plug-ins* utilizados na construção da ferramenta. Em princípio o trabalho tomou por base o Projeto Modeling, bem como uma ferramenta XML denominada XML Schema Definition (XSD). A seguir serão apresentados projetos e os conceitos empregados na construção do *toolkit* proposto.

5.1.1.1 EMF - Eclipse Modelling Framework

O EMF é um framework de geração de código com capacidade de gerar *plug-ins* Eclipse e outras aplicações baseadas em um modelo estruturado de classe simples,

utilizando uma API reflexiva para manipular esses metamodelos (BUDINSKY, 2004; FONSECA, 2008). A utilização do EMF proporciona ao desenvolvedor a possibilidade de se concentrar no modelo sem ter que se preocupar com os detalhes da implementação, pois o *plug-in* facilita a concepção e a implementação de um modelo estruturado proporcionando a geração de código. O EMF metamodelo ou Ecore - é o modelo usado para representar modelos, sendo que eles podem ser expressos em XML, código Java ou ainda documento XML Metadata Interchange (XMI). A Figura 5.3 mostra um subconjunto do modelo Ecore que contém quatro classes necessárias para representar o modelo, são elas:

i) **EClass** - é usada para representar uma classe no modelo;

ii) **EAtribute** - representa um atributo modelado, possui nome e tipo;

iii) **EReference** - representa a associação entre classes, possui nome, tipo de referência e um sinalizador booleano sobre existência de contenção;

iv) **EDataType** - representa um tipo de atributo, podendo ser primitivo ou um tipo de objeto.

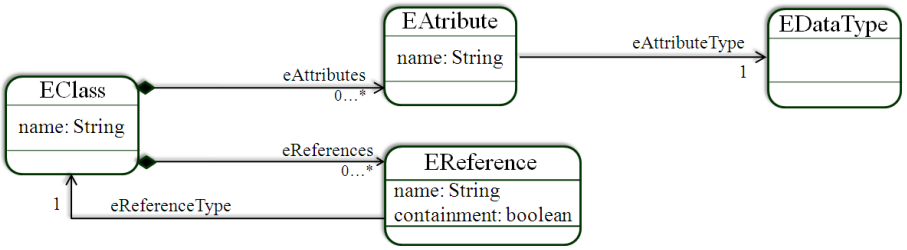


Figura 5.3: Principais elementos do EMF (adaptado de Budinsky (2004)).

5.1.1.2 GEF - Graphical Editing Framework

O GEF é um projeto open-source baseado no padrão de arquitetura de *software* Model-View-Controller (MVC), sendo utilizado para o desenvolvimento de editores gráficos no Eclipse. O editor fornecido pelo GEF é baseado no *plug-in* Draw2d que provê um conjunto de ferramentas para construção e renderização de objetos gráficos baseados em Standard Widget Toolkit[1] (SWT). Os Edit-Parts são objetos que fazem parte do núcleo do GEF e fazem o controle ou o mapeamento entre views e models, são os controllers da arquitetura MVC (Figura 5.4).

[1]SWT é um *toolkit* gráfico utilizado na plataforma Java, sendo uma alternativa para o Abstract.

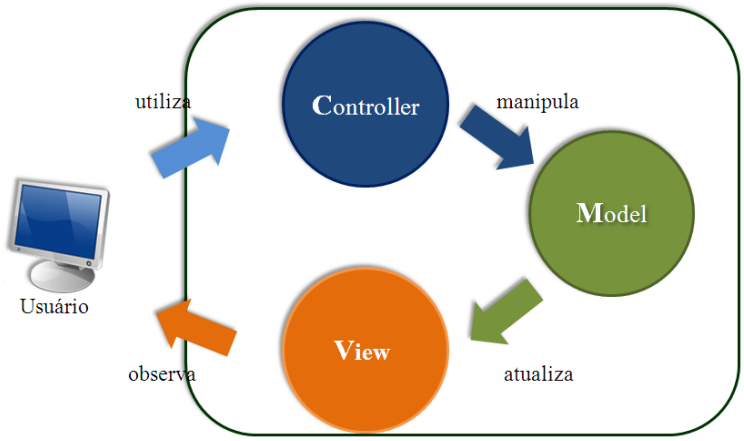


Figura 5.4: Esquema da arquitetura MVC.

O editor gráfico do GEF cria e edita instâncias do modelo, permitindo ações comuns como: copiar, colar, desfazer, refazer. O Draw2D fornece figuras de formas não retangulares que podem ser compostas através de encaixe, de modo a formar novas figuras mais complexas, também oferece outras características comuns como: painel de visão geral, régua, grades, paletes e visualização das propriedades (MOORE *et al.*, 2004).

No entanto, a implementação dos objetos do GEF, views e models é construída sem o auxílio de nenhum elemento gráfico que facilite o processo, isto é, construir uma figura customizada por meio de uma classe, não garante como esta vai se comportar, o que tende a prejudicar a produtividade do *plug-in* (FONSECA, 2008).

A Figura 5.5 ilustra um diagrama produzido por GEF (GEF, 2011), onde pode ser visto uma paleta de componentes, um editor gráfico com uma aplicação modelada utilizando os componentes da paleta em questão. Pode-se notar a representação em árvore da aplicação na lateral esquerda da referida figura.

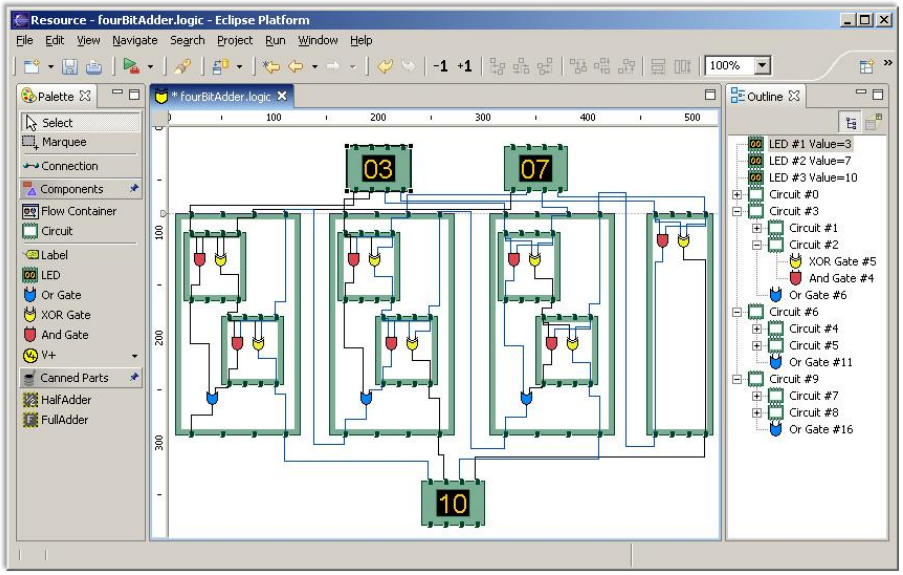


Figura 5.5: Diagrama produzido por GEF (GEF, 2011).

5.1.1.3 GMF - Graphical Modeling Framework

O GMF é um *plug-in* da plataforma Eclipse utilizado para construção de editores gráficos de modelos. O GMF integra os *plug-ins* EMF e GEF na forma visual, possibilitando facilidade na implementação de ferramentas. O GMF basea-se em dois componentes principais (GMF, 2011):

a) Tooling - consiste em uma abordagem orientada a modelos que descreve notações, semânticas e ferramentas para geração de editores gráficos e geração de código referente à implementação;

b) Runtime - disponibiliza ferramentas e componentes básicos para o editor, provê recursos de persistência e sincronização entre modelo e notação visual, de modo a interligar elementos EMF e GEF em tempo de execução.

O diagrama da arquitetura GMF ilustra o funcionamento (Figura 5.6), onde um editor gráfico é gerado e depende diretamente do componente GMF runtime que agrega as ferramentas EMF (que instancia e manipula modelos) e GEF (framework MVC que crie elementos gráficos do Editor) (FERREIRA, 2009).

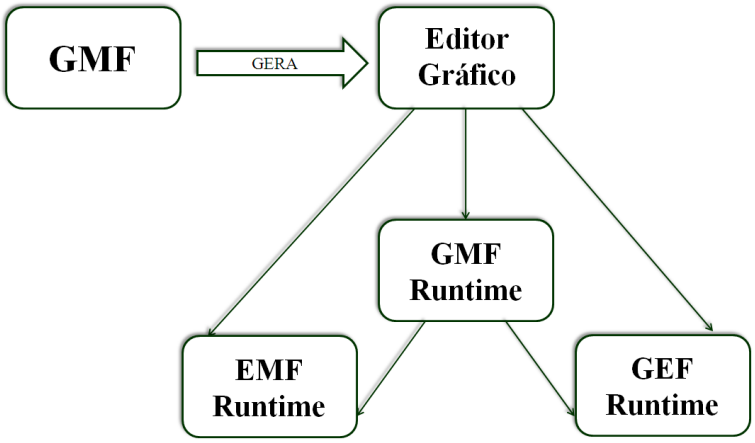


Figura 5.6: Arquitetura GMF (GMF, 2011).

A seguir serão descritos os editores pertencentes ao componente tooling e para melhor compreensão, será descrita a Figura 5.7, que apresenta os componentes e modelos utilizados no desenvolvimento de uma aplicação GMF e que facilita a integração do EMF e GEF:

- *Graphical Definition Model* (GMFgraph) - editor que define os atributos gráficos que serão visualizados em um ambiente baseado no GEF, tais como retângulo e conexões;
- *Tooling Definition Model* (GMFtool) - editor onde são projetados barras de ferramentas, menus e palettes (define chamadas de criação de elementos no diagrama);
- *Mapping Model* (GMFmap) - é o principal editor do GMF, efetua a relação entre os componentes GMFgraph, GMFtool e o metamodelo. No editor são definidos os elementos que compõe o metamodelo no diagrama, o relacionamento entre esses elementos, além de auditoria na edição do diagrama verificada em tempo de execução;
- *Generator Model* (GMFGen) - é o responsável pela geração de código do diagrama, além de possibilitar a edição de preferências em relação à implementação do *plug-in*. É o último modelo criado, é utilizado para gerar o *plug-in* final.

Devido basear seus *plug-ins* nos diagramas do Eclipse, o GMF proporciona o aumento de produtividade, já que permite a integração dos *plug-ins* EMF e GEF e conseqüente reuso (com pouca ou nenhuma alteração) desses editores, que podem trabalhar em domínios diferentes (FONSECA, 2008).

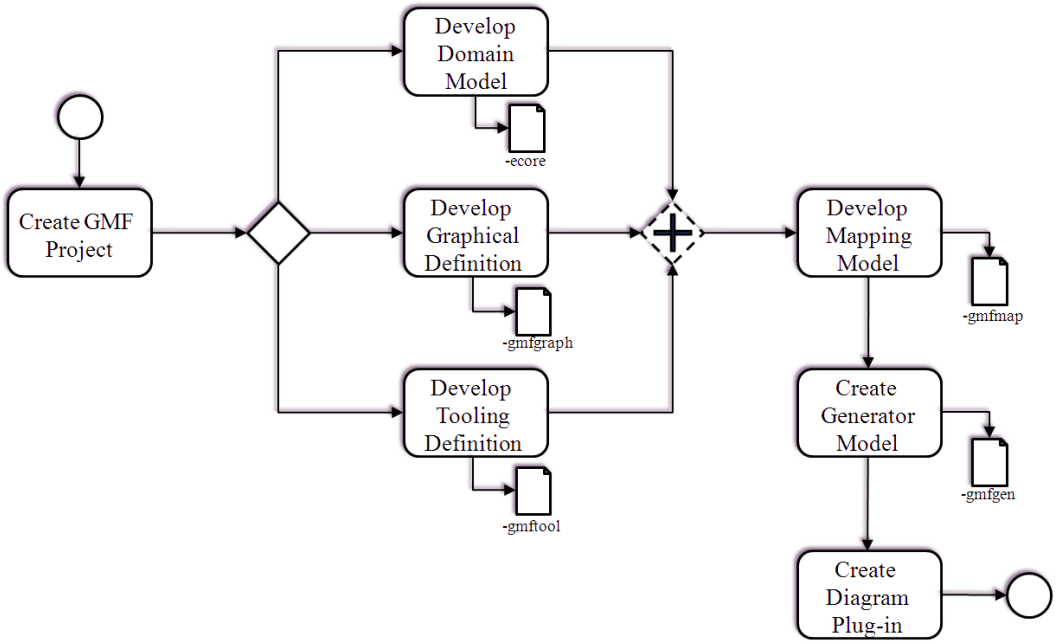


Figura 5.7: Visão dos modelos no desenvolvimento baseado em GMF (adaptado de GMF (2011)).

5.1.1.4 Java Emitter Templates (JET)

O JET é um *plug-in* gerador de código para a plataforma Eclipse. O JET produz código e outros recursos do Eclipse (arquivos, pastas, projetos) a partir de uma instância do modelo abstrato (documentos XML simples ou ecore) (ELDER, 2005; ANISZCZYK; MARZ, 2006; JET, 2011). O JET se utiliza de *templates* para gerar código, atuando através de substituição de campos pré-estabelecidos por conteúdo dos atributos do modelo que está em transformação. A utilização do JET permite a implementação de geradores baseados em uma linguagem específica, além de favorecer a formatação e a heterogeneidade do código gerado (FONSECA, 2008).

A criação de modelos abstratos e sua transformação em código é possível sem a utilização do MDD. No entanto, sua utilização viabiliza a automação da geração de código e a torna uma boa prática da engenharia de *software*. A interação e o funcionamento dos componentes do JET são ilustrados na Figura 5.8, sendo que seus componentes principais são: a) JET *template* - script que contém o mapeamento de elementos e trechos de código para um código alvo; b) JET genlet - trata-se de uma classe Java que contém o método generate, que retorna uma string contendo o código gerado através de um argumento (um modelo); c) JET generator - é uma aplicação que carrega um modelo a partir de um arquivo e invoca o JET genlet e salva o código gerado em arquivos (FERREIRA, 2009).

* JET generator - aplicação que carrega o modelo a partir de um arquivo e, em seguida, invoca o JET genlet e salva o código gerado em arquivos.

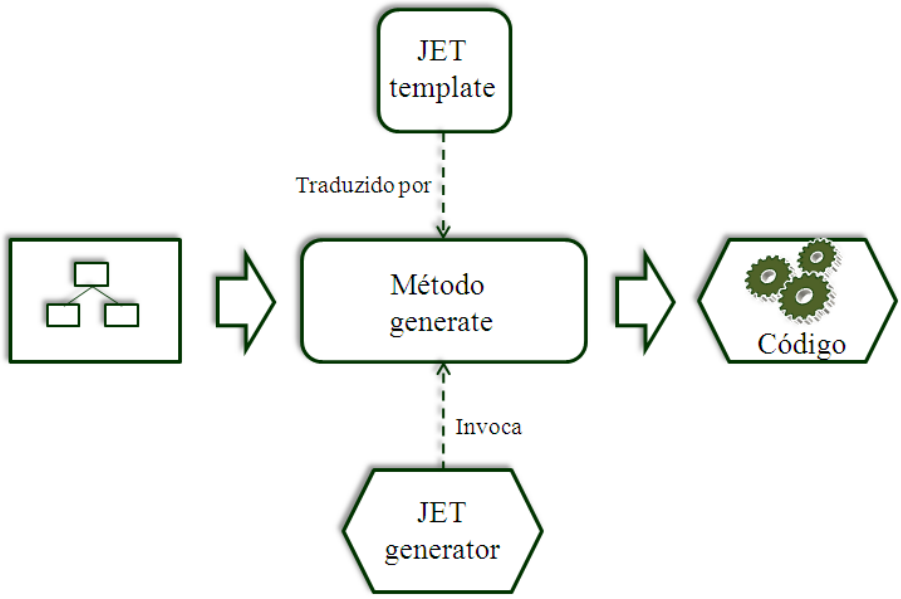


Figura 5.8: Funcionamento do JET (adaptado de JET (2011)).

5.1.2 Definição do modelo CommX

Para melhor compreensão e entendimento do problema apresentado, o nível de abstração será elevado com intuito de melhor expressar sua solução, a fim de que a mesma seja inclinada a automatização, baseando-se em sua freqüência de utilização e conseqüente facilidade de definição. A modelagem tem por objetivos específicos, aumentar o nível de abstração e gerar um produto através da especificação modelada (DIAS, 2009).

O modelo proposto é baseado em (ABDUNABI, 2007), no qual foi definido o modelo XM que possui entradas que servem para comunicação entres as XMs participantes do modelo. O modelo proposto neste trabalho é uma extensão do referido trabalho, onde são adicionados mais dois canais de comunicação, permitindo que as XMs troquem mensagens entre si, flexibilizando ainda mais seu trabalho e permitindo a criação de componentes XM conforme definido no Capítulo 3. O modelo é expresso em XSD e serve de base para o metamodelo para produção do Ecore. A Figura 5.9 apresenta o modelo CommX proposto e em seguida o Código 5.1 que representa o modelo.

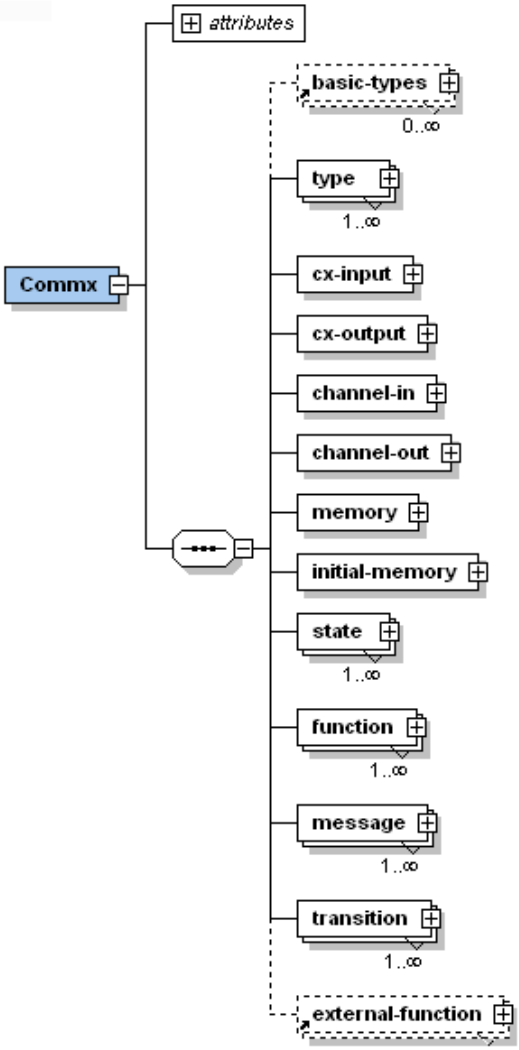


Figura 5.9: Diagrama (ecore) CommX em XSD.

```
1 <!-- Definição do Commx -->
2 <xsd:element name="Commx">
3         <xsd:complexType>
4 <xsd:sequence>
5         <xsd:element ref="basic-types" minOccurs="0" maxOccurs="unbounded
              "/>
6         <xsd:element name="type" type="type" maxOccurs="unbounded"/>
7         <xsd:element name="cx-input" type="cx-input"/>
8         <xsd:element name="cx-output" type="cx-output"/>
9         <xsd:element name="channel-in" type="channel-in"/>
10        <xsd:element name="channel-out" type="channel-out"/>
11        <xsd:element name="memory" type="memory"/>
12        <xsd:element name="initial-memory" type="initial-memory"/>
13        <xsd:element name="state" type="state" maxOccurs="unbounded"/>
14        <xsd:element name="function" type="function" maxOccurs="unbounded
              "/>
```

```
15          <xsd:element name="message" type="message" maxOccurs="unbounded"
                />
16          <xsd:element name="transition" type="transition" maxOccurs="
                unbounded"/>
17          <xsd:element ref="external-function" minOccurs="0" maxOccurs="
                unbounded"/>
18      </xsd:sequence>
19  <xsd:attribute name="model-name" type="allowed-string" use="required"/>
20      <xsd:attribute name="initial-state" type="xsd:IDREF" use="required"
            ecore:reference="state"/>
21      </xsd:complexType>
22  </xsd:element>
```

Código 5.1: Definição da CommX em XSD.

A inclusão de mais dois canais de comunicação, channel-in e channel-out, em relação ao trabalho de Abdunabi (2007) pode ser notado, proporcionando dessa maneira, um desenvolvimento modular de cada componente. O meio de envio das comunicações por streams, message é outro adicional ao modelo CommX proposto. A Figura 5.10 ilustra a extensão de state em relação ao modelo XM citado anteriormente e a criação do message, o Código 5.2 referencia as respectivas extensões do modelo.

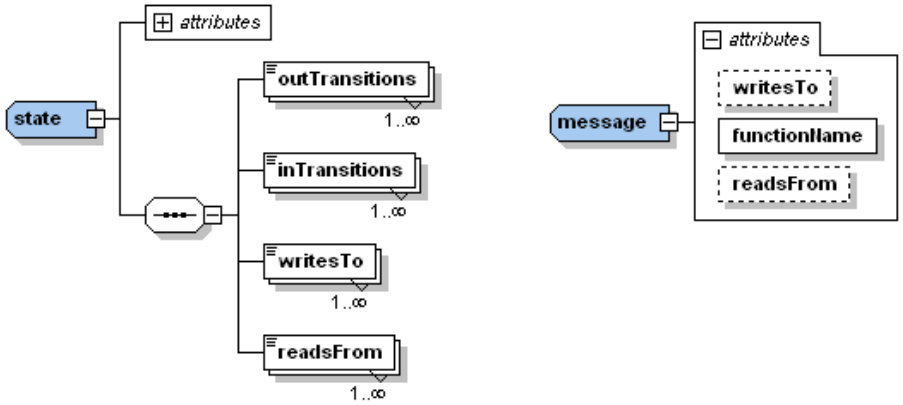


Figura 5.10: Diagramas state e message XSD.

```
1  <!-- Definição do state -->
2  <xsd:complexType name="state">
3          <xsd:sequence>
4              <xsd:element name="outTransitions" type="xsd:IDREF" maxOccurs="
                   unbounded" ecore:reference="transition" ecore:opposite="
                   fromState" ecore:transient="true"/>
5              <xsd:element name="inTransitions" type="xsd:IDREF" maxOccurs="
                   unbounded" ecore:reference="transition" ecore:opposite="
```

```
                toState" ecore:transient="true"/>
6           <xsd:element name="outMessages" type="xsd:IDREF" maxOccurs="
                unbounded" ecore:reference="message" ecore:opposite="writesTo
                " ecore:transient="true"/>
7         </xsd:sequence>
8         <xsd:attribute name="name" type="id-name" use="required"/>
9 </xsd:complexType>
10 . . .
11 <!-- Definição do message -->
12 <xsd:complexType name="message">
13         <xsd:attribute name="writesTo" type="xsd:IDREF" ecore:reference="
               state" ecore:opposite="outMessages"/>
14       <xsd:attribute name="functionName" type="allowed-string" use="
              required"/>
15       <xsd:attribute name="readsFrom" type="xsd:IDREF" ecore:reference="
              state" ecore:opposite="inMessages"/>
16 </xsd:complexType>
```

Código 5.2: Definição de state e message em XSD.

5.2 Ufam Sensor CommX Toolkit

O *toolkit* é composto por quatro *plug-ins*, sendo que cada um difere dos demais pelos recursos oferecidos, característica esta que permite que modificações sejam feitas bem como adição de novas funcionalidades para o futuro. *Ufam Sensor CommX* é uma ferramenta open source construída na plataforma Eclipse para criação de aplicações em RSSF e geração de código em *nesC*, usa um método formal através de uma máquina de estados denominado CXM.

A ferramenta oferece ao programador suporte ao desenvolvimento para RSSF através de um editor gráfico e uma paleta de componentes, que poderá modelar suas aplicações utilizando uma interface amigável e de fácil manuseio. *Ufam Sensor CommX* provê ainda, a visualização da implementação em duas formas, XML e em árvore de componentes (*treeview*). Essas possibilidades de visualização serão melhor descritas na seções seguinte, onde serão mostrados os códigos implementados nas duas visões. Outro ponto a ser destacado é a geração automática de código, isto é possível através da criação de mecanismos que permitem a transformação de modelos. Esses mecanismos de transformação são denominados templates, seu funcionamento será detalhado nas seções seguintes. A Figura 5.11 ilustra o processo de criação de aplicações utiliando o *plug-in Ufam Sensor*

CommX.

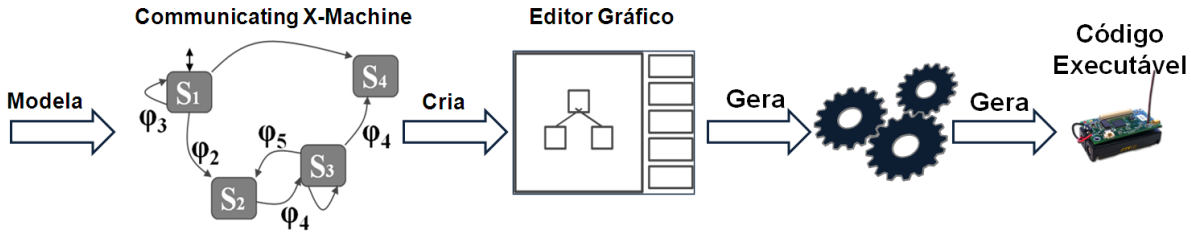


Figura 5.11: Processo de criação de aplicações RSSF usando *Ufam Sensor CommX*.

5.2.1 Características do toolkit

O *Ufam Sensor CommX* foi desenvolvido na plataforma Eclipse utilizando o projeto Modelling, inicialmente na versão 3.2.2 (codinome Calixto) e em seguida migrada para a versão 3.4.2 (codinome Ganymede), devido a atualização de *plug-ins* disponibilizados na plataforma mais recente. A seguir serão listados os *plug-ins* utilizados de acordo com cada versão da plataforma Eclipse.

- Calixto (IDE 3.2.2) - 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), bem como adicionando as dependências dos respectivos *plug-ins*.

- Ganymede (IDE 3.4.2) - GMF (2.1.0), EMF (2.4.0), GEF (3.3.0), EMF validation framework (1.2.0), XML Schema Infoset Model XSD (2.4.0) e JET (1.1.0), além de adicionar as dependências aos respectivos *plug-ins*.

O *plug-in* proporciona além da modelagem de aplicações a geração de código a partir das especificações criadas no editor gráfico. Isto é possível devido o uso de *templates* para linguagem de programação *nesC*. O *Ufam Sensor Toolkit* é composto por cinco *plug-ins*, Model CommX (*br.edu.ufam.pgee.commx*), Edit CommX (*br.edu.ufam.pgee.commx .edit*), Tree Editor CommX (*br.edu.ufam.pgee.commx.editor*), Graphical Editor CommX (*br.edu .ufam.pgee.commx.diagram*) e codeGenerator CommX (*br.edu.ufam.pgee.commx .codeGenerator*), conforme pode ser visto na Figura 5.12. Cada *plug-in* provê características que se distinguem dos demais, fazendo com que as modificações futuras possam ser implementadas facilmente. A seguir serão descritos cada um dos *plug-ins* que compõem a ferramenta.

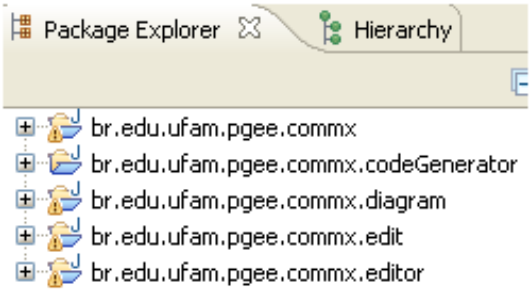


Figura 5.12: *Plug-ins* que compõem *Ufam Sensor CommX*.

5.2.2 Model CommX

Model é criado com base no modelo CommX definido em XSD. Ele serve de entrada para criação do metamodelo (ecore), base para criação de outros *plug-ins*. O Model é baseado no padrão de projetos MVC e provê dois modelos, o Commx.ecore e Commx.genmodel, conforme pode ser visto na Figura 5.13. O primeiro é obtido através da EMF a partir do modelo em XSD, já o gerador de modelos é obtido através do processo de derivação do ecore.

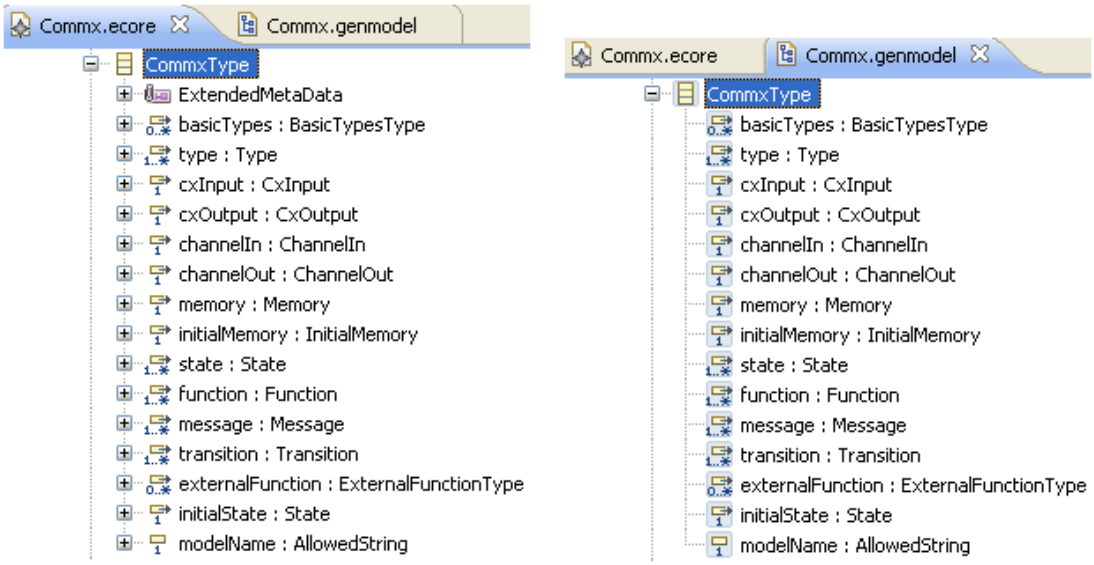


Figura 5.13: *Commx.genmodel* e *Commx.ecore* gerados a partir de Commx XSD.

5.2.3 Edit CommX

O *plug-in* Edit CommX, é o segundo *plug-in* criado. Ele faz parte do view do modelo MVC e é o responsável por mostrar ícones, nomes dos objetos XMs e a estrutura de árvore das aplicações. O framework EMF .Edit é utilizado na geração deste *plug-in*

valendo-se do Commx.genmodel do modelo.

5.2.4 Tree Editor CommX

O *plug-in* Tree Editor CommX faz parte do view e do controler do modelo MVC, responsável por fornecer a exibição da estrutura de árvore e as propriedades de suas folhas, permitindo interação do usuário com elementos não gráficos, sendo que o framework EMF .Editor é utilizado para gerar o editor de código através do Commx.genmodel do modelo. A Figura 5.14 apresenta uma aplicação fazendo uso do editor em questão, onde pode ser notado na raiz da árvore o nome XmBlink.cx, onde o sufixo cx se refere ao método CXM como forma de identificar o tipo de modelo construído. Em seguida, são apresentados os demais componentes que fazem parte da aplicação apresentada, pode-se notar mais uma vez expresso em uma de suas folhas, a alusão ao referido modelo, os demais componentes são formados por estados (*states*), mensagens (*messages*) e transições (*transitions*). O *plug-in* tem ainda a possibilidade de expressar o modelo em XML, o que proporciona a transformação do modelo em outras linguagens de programação existentes.

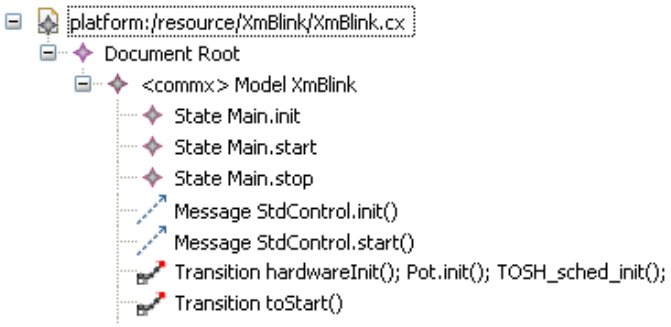


Figura 5.14: Aplicação XmBlink expressa no Tree Editor CommX.

5.2.5 Graphical Editor CommX

O editor gráfico faz parte da view e do controller do modelo MVC, fornecendo uma interface gráfica que permite ao usuário desenhar um diagrama de estado XM. De modo a utilizar o modelo GMF para criação do *plug-in* são necessários o modelo Commx.gmfgraph, responsável pela definição de figuras, componentes (estados) e as ligações entre os estados (*transitions* ou *messages*). Além do modelo de definição de ferramentas Commx.gmftool, responsável por especificar a paleta de componentes (*pallete*), onde são organizados por ícones os elementos gráficos componentes (neste trabalho, estados) e as ligações que podem ser transitions ou messages. Outro modelo necessário é o mapeamento Commx.gmfmap, responsável pela ligação dos três modelos anteriormente

descritos (Commx.ecore, Commx.gmfgraph e Commx.gmftool) e finalmente, o modelo de geração de código Commx.gmfgen, que é utilizado na geração de código.

5.2.6 Code Generator CommX

O *plug-in* Code Generator, oferece a facilidade de geração de código a partir da instância do modelo criado pela árvore do editor gráfico, utilizando-se do JET2 para criação do mesmo. O *plug-in* utiliza das especificações descritas em XML modeladas no Tree Editor, e a partir daí efetua a geração automática de código para uma linguagem alvo. A seguir serão apresentados casos de uso para ilustrar a utilização da ferramenta e a consequente geração de código para cada um destes.

5.3 Criação do Toolkit

O processo de criação da *Ufam Sensor CommX* se deu através da criação de um metamodelo (ecore) baseado na modelagem CommX descrita em XSD, conforme visto na seção 5.1.2. O processo de criação tomou como base o guia de construção de editores GMF, denominado de Dashboard. O referido guia, ilustrado na Figura 5.15, é composto por um workflow de cinco processos bem definidos com intuito de ajudar o desenvolvedor a efetuar cada etapa de maneira correta. A seguir serão descritas as etapas proporcionadas pelo GMF Dashboard.

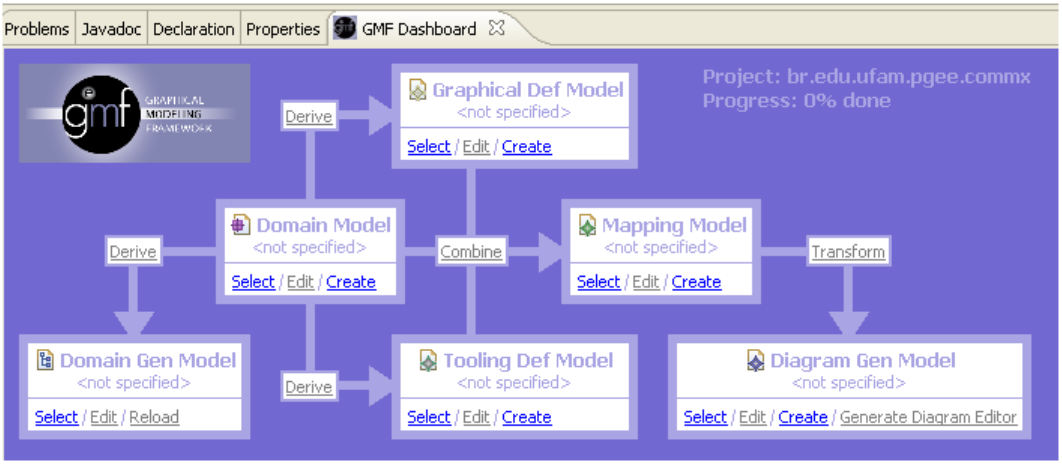


Figura 5.15: GMF Dashboard do *toolkit Ufam Sensor CommX*.

- *Domain* e *Domain Gen Model* - são produzidas pelo EMF (ecore e genmodel), referem-se ao modelo conceitual concebido, entidade e suas relações.

- *Graphical Definition Model* - é usado para definir o visual do editor, seus elementos gráficos (figura de nós, conexões, legendas e outros).

- *Tooling Definion Model* - é o modelo que define os componentes que estarão presentes na paleta do editor.

- *Mapping Model* - é o modelo responsável pelo mapeamento entre os elementos gráficos e os de domínio.

- *Digram Editor Generation Model* - é o responsável pela junção dos *plug-ins* e produção de um novo na plataforma Eclipse. Ele permite editar preferências nas propriedades para criação do novo *plug-in*.

A Figura 5.16 apresenta o *toolkit Ufam Sensor CommX* e seus respectivos componentes identificados por letras, sendo que serão especificados a seguir de acordo com seu respectivo papel.

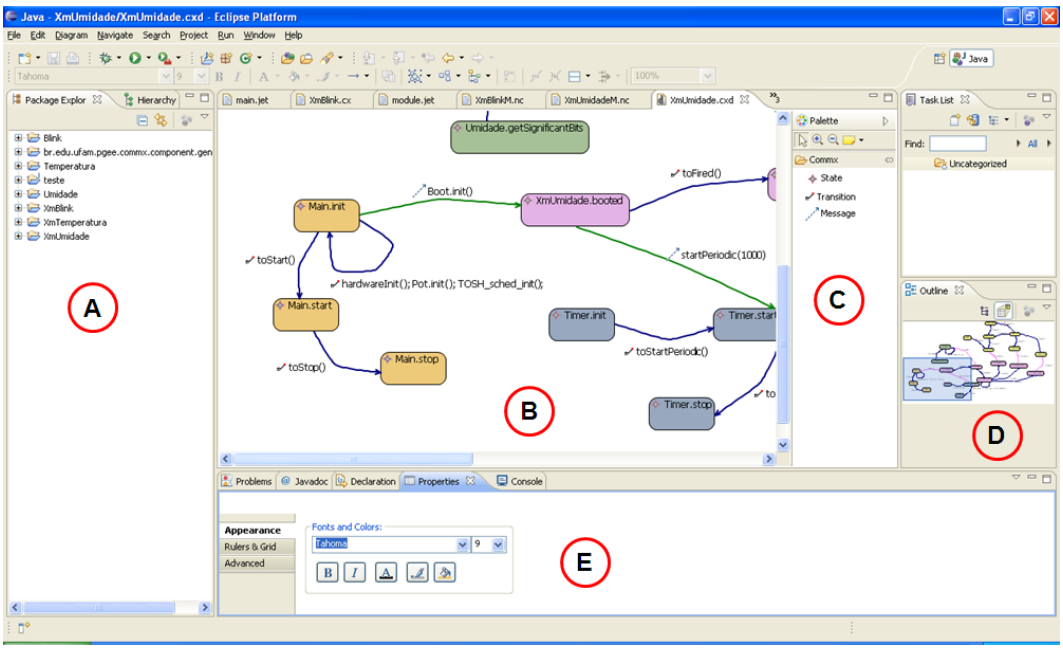


Figura 5.16: Interface *Ufam Sensor CommX*.

- (A) *Explorer* - tem por objetivo a organização das aplicações através de arquivos correspondentes às mesmas com intuito de alcançar um melhor gerenciamento dos arquivos pertinentes à aplicação.

- (B) *Modeling View* - área especificada para criação das aplicações de modo interativo, permitindo ao desenvolvedor uma melhor compreensão do modelo.

- (C) *Pallete* - refere-se à área onde estão os componentes utilizados na criação de aplicações, é composto por três tipos, state, transition e message. State e transition fazem referência ao autômato (componente XM) e message expressa a comunicação entre as referidas máquinas de estado, promovendo um sistema que se comunica.

- (D) *Propertiew View* - tem como objetivo oferecer ao desenvolvedor alterar características dos modelos criados na ferramenta.

- (E) *Outline View* - oferece ao desenvolvedor uma visão geral da distribuição de componentes no modelo, favorecendo a navegação na aplicação modelada.

5.4 Geração de Código

Para construção de geradores de código, a utilização de *templates* é a solução mais adequada. Templates são arquivos de texto que possuem regras de construção utilizando-se de seleções e expansões de código (Lucrédio, 2009 apud Czarnecki e Eisernecker, 2000). O processo é controlado pelo *plug-in* Code Generator (br.edu.ufam.pgee.commx) citado anteriormente. O *plug-in* é composto dos seguintes *templates* para geração do código *nesC*, Main, Configuration, Module e Dump, sendo que todos possuem o sufixo ".jet" para identificá-los que são parte do *plug-in* JET2. A Figura 5.17 mostra o referido *plug-in* e seus respectivos *templates* de geração de código. A seguir serão descritos cada um destes mencionados anteriormente.

- *Main.jet* - trata-se do arquivo principal, controla o processo de geração de código através da chamada aos demais arquivos componentes.

- *Module.jet* - refere-se ao *template* gerador do *Module* para aplicações em *TinyOS*, extrai as informações referentes à lógica da aplicação.

- *Configuration.jet* - *template* que extrai e gera os componentes presentes na aplicação, bem como as interfaces usadas e providas à aplicação.

- *Dump.jet* - é utilizado na conferência do código gerado, expressa quantidade de estados utilizados no modelo, número de transições (*transitions*) e mensagens (*messages*).

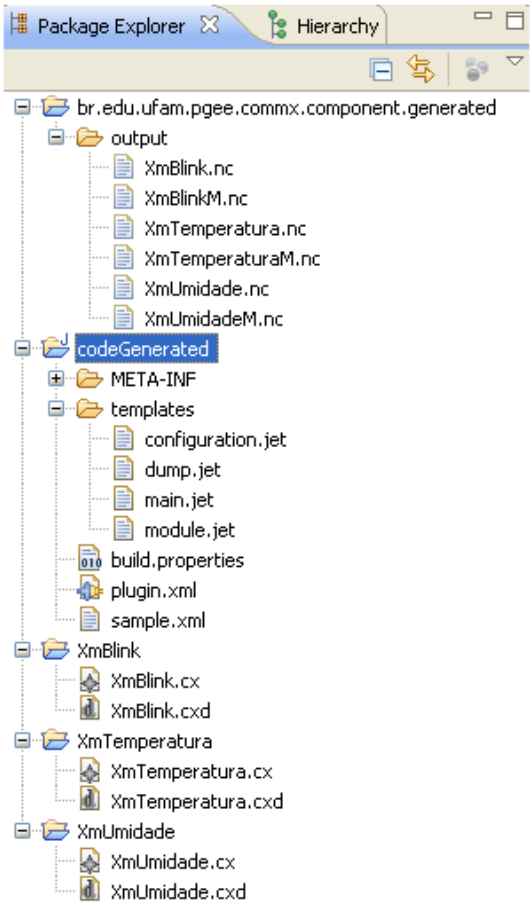


Figura 5.17: *Plug-ins* de geração de código.

5.5 Resumo

Neste capítulo foi apresentado o *toolkit Ufam Sensor CommX*, desde a definição do modelo em XSD, até a geração do referido editor gráfico. Para isto foi feito um levantamento da tecnologia empregada, *plug-ins* envolvidos na criação do *toolkit* , além das etapas de criação da ferramenta.

Inicialmente foi apresentada a arquitetura do *toolkit*, onde foi abordada a plataforma Eclipse e seus componentes, o módulo core e o módulo interface com o usuário. Também foram descritos os *plug-ins* utilizados para a construção do *toolkit* e que fazem parte do projeto Modeling (GMF, EMF e GEF), além do JET que corresponde à geração de código utilizando templates.

A definição do modelo CommX foi feita em XSD é o ponto de partida para a construção do *toolkit*, é a partir dele que se origina o ecore e o genmodel, modelos bases na construção dos *plug-ins* envolvidos. O *Ufam Sensor CommX Toolkit* foi desenvolvido utilizando o *GMF Dashboard*, como ferramenta de orientação para a construção da referida

ferramenta. O *toolkit* é composto por cinco *plug-ins*, *Model CommX*, *Edit CommX*, *Tree Editor CommX*, *Graphical Editor CommX* e *codeGenerator CommX*.

A geração de código foi feita utilizando *templates* do *plug-in* JET2, compostos por *Main*, *Configuration*, *Module* e *Dump* para geração do código em *nesC*. No próximo capítulo serão apresentados o funcionamento da referida ferramenta utilizando-se dos casos de uso proposto no capítulo anterior.

Capítulo 6

Testes e Resultados

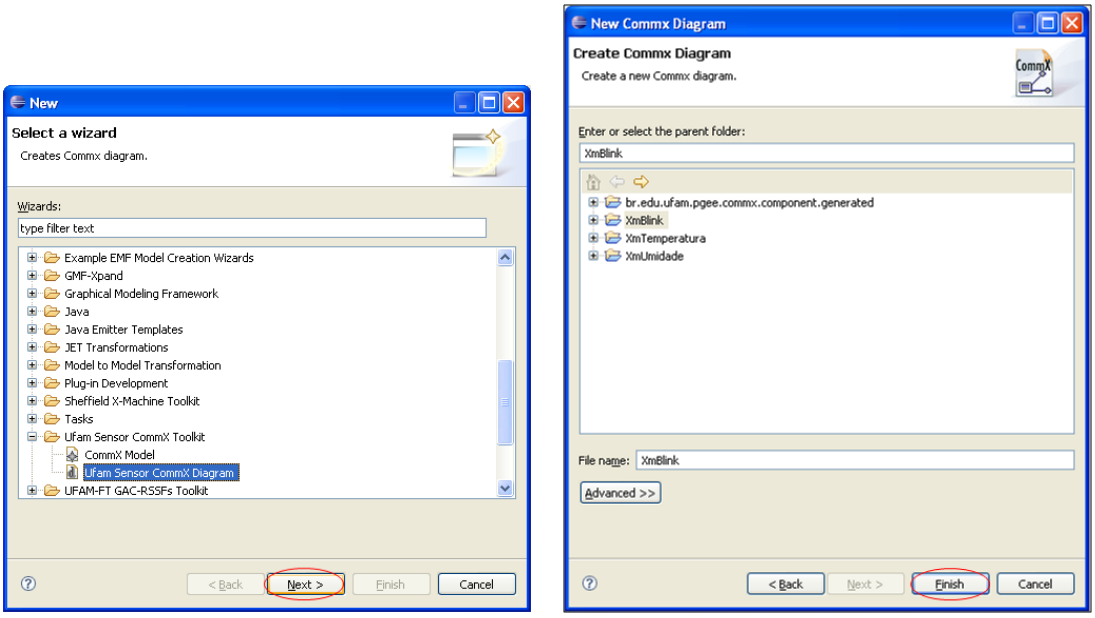
Este capítulo apresenta a implementação dos casos de uso propostos no Capítulo 4, utilizando o *toolkit Ufam Sensor CommX*, de modo a testar desde a modelagem até a geração automática de código. O capítulo apresenta ainda os artefatos gerados (código em XML e em forma de árvore de componentes) com a utilização da ferramenta. Testes e resultados das provas de conceito também serão apresentados ao final do capítulo.

6.1 Automatizando os Cenários

Nesta seção são apresentados os casos de uso com a utilização do *toolkit Ufam Sensor CommX*. Inicialmente é apresentado o caso de uso XmBlink, mostrando desde a criação do projeto até a geração de código em *nesC* . Em seguida, o caso de uso explorado é o de monitoramento de pacientes em hospital, onde os sensores de temperatura e umidade são estudados. Da mesma forma que o primeiro caso de uso, estes também apresentam as etapas de criação do projeto até a geração de código.

6.1.1 XmBlink

A aplicação XmBlink, conforme descrito anteriormente, controla um *led* vermelho, fazendo-o piscar por um intervalo de tempo pré-determinado, neste caso, 1000 milisegundos. Para iniciar a construção de aplicações no *toolkit*, é necessário criar uma pasta no Explorer com o nome da aplicação proposta e em seguida selecionar o *wizzard CommX* e nomeá-la, conforme Figura 6.1. A aplicação criada gera dois arquivos, o ".cx" e o ".cxd", sendo que o primeiro apresenta duas visões, uma em árvore de componentes e outra em XML que será descrito na seção de resultados. O segundo arquivo se refere ao modelo gráfico da aplicação, onde são utilizados os componentes da Paleta do *toolkit* (*state*, *transition* e *message*). A Figura 6.2 expressa a aplicação modelada em forma de árvore de componentes.



(a) (b)

Figura 6.1: Processo de criação de aplicação usando *toolkit*.

Figura 6.2: Visão da aplicação XmBlink em árvore de componentes.

A Figura 6.3 apresenta a aplicação modelada na ferramenta, sendo formada por quatro componentes, *Main*, *SglTimer*, *LedsC*, *XmBlink*, que estão representados por cores diferentes. Pode-se observar que os componentes, estão conectados (se comunicando) através de uma ligação na cor azul, esta ligação é o componente message. Também pode-se notar que dentro dos componentes existe uma ligação entre estados de coloração verde, representada pelo componente transition. Estas características visuais foram criadas para deixar mais claro o papel que cada uma possui no diagrama proposto. Outra informação extraída do modelo gráfico, é que a cada transição (*transition*) entre estados ou a cada message entre componentes pode ser representada por um comando ou uma função, oferecendo a flexibilidade que o método possui.

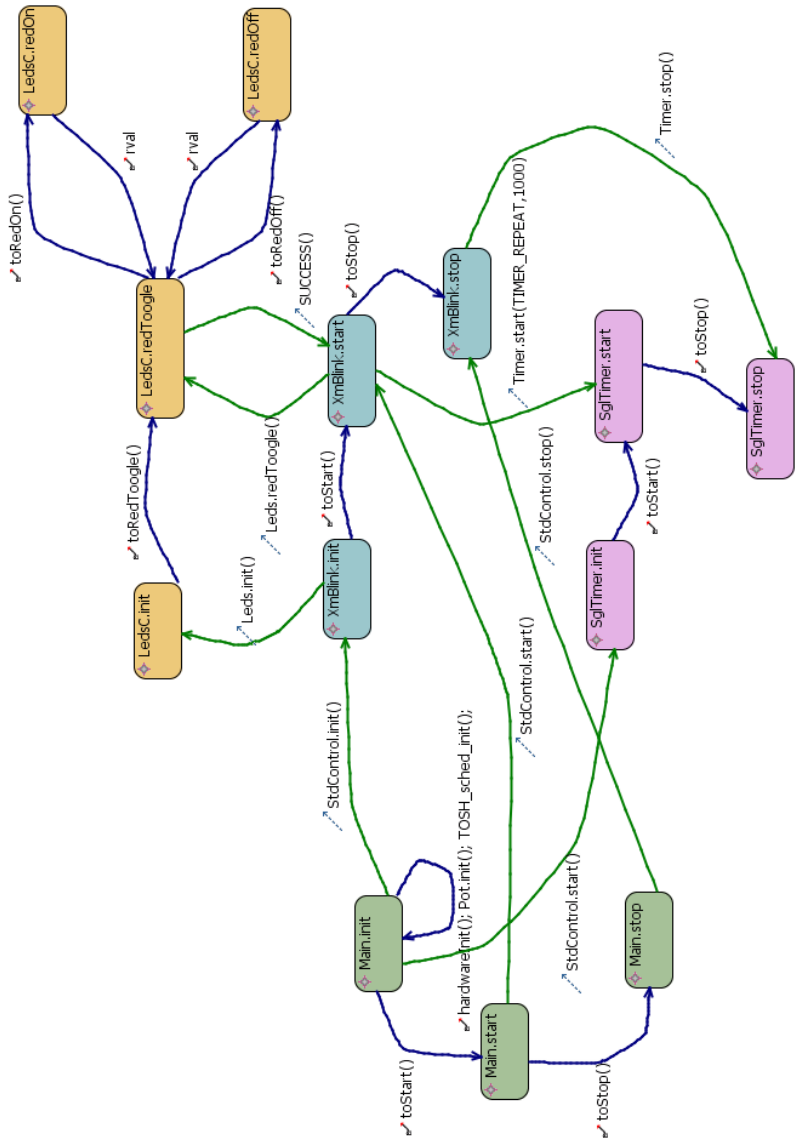


Figura 6.3: Visão da aplicação XmBlink em árvore de componentes.

6.1.2 XmUmidade

A partir desta seção será abordada a utilização da ferramenta no caso de uso de monitoramento de pacientes. Para criação da aplicação XmUmidade serão seguidos os mesmos procedimentos da criação da aplicação XmBlink descritos anteriormente.

A Figura 6.4 apresenta o código expresso em árvore de componentes e a Figura 6.5 mostra a modelagem feita com o *toolkit*. A aplicação é composta por cinco componentes, *Main*, *LedsC*, Umidade (correspondente ao sensoriamento), *Timer* e XmUmidade que controla a aplicação.

Figura 6.4: Aplicação XmUmidade expressa em árvore.

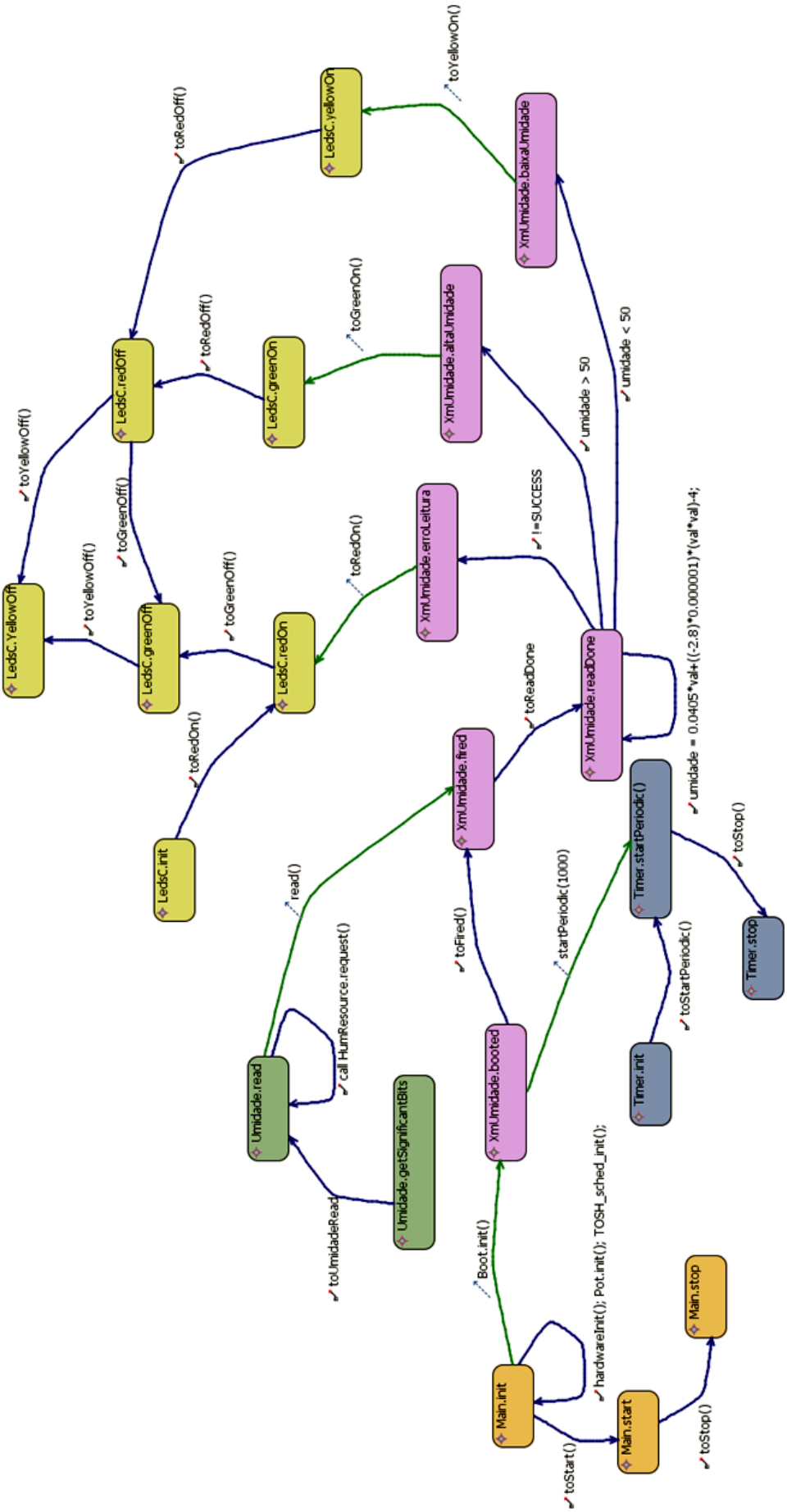


Figura 6.5: Aplicação XmUmidade expressa em árvore.

6.1.3 XmTemperatura

A aplicação XmTemperatura é expressa em forma de árvore de componentes, conforme pode ser conferido na Figura 6.6. O diagrama gerado no *toolkit* é mostrado na Figura 6.7.

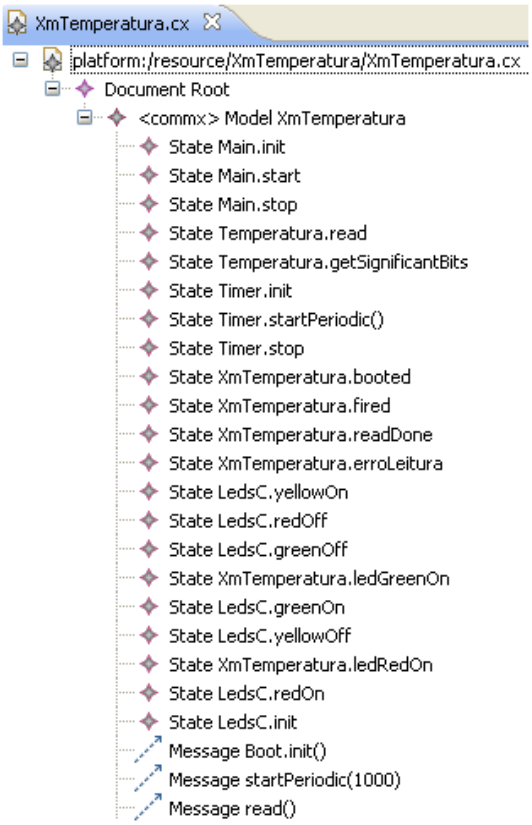


Figura 6.6: Aplicação XmTemperatura expressa em árvore.

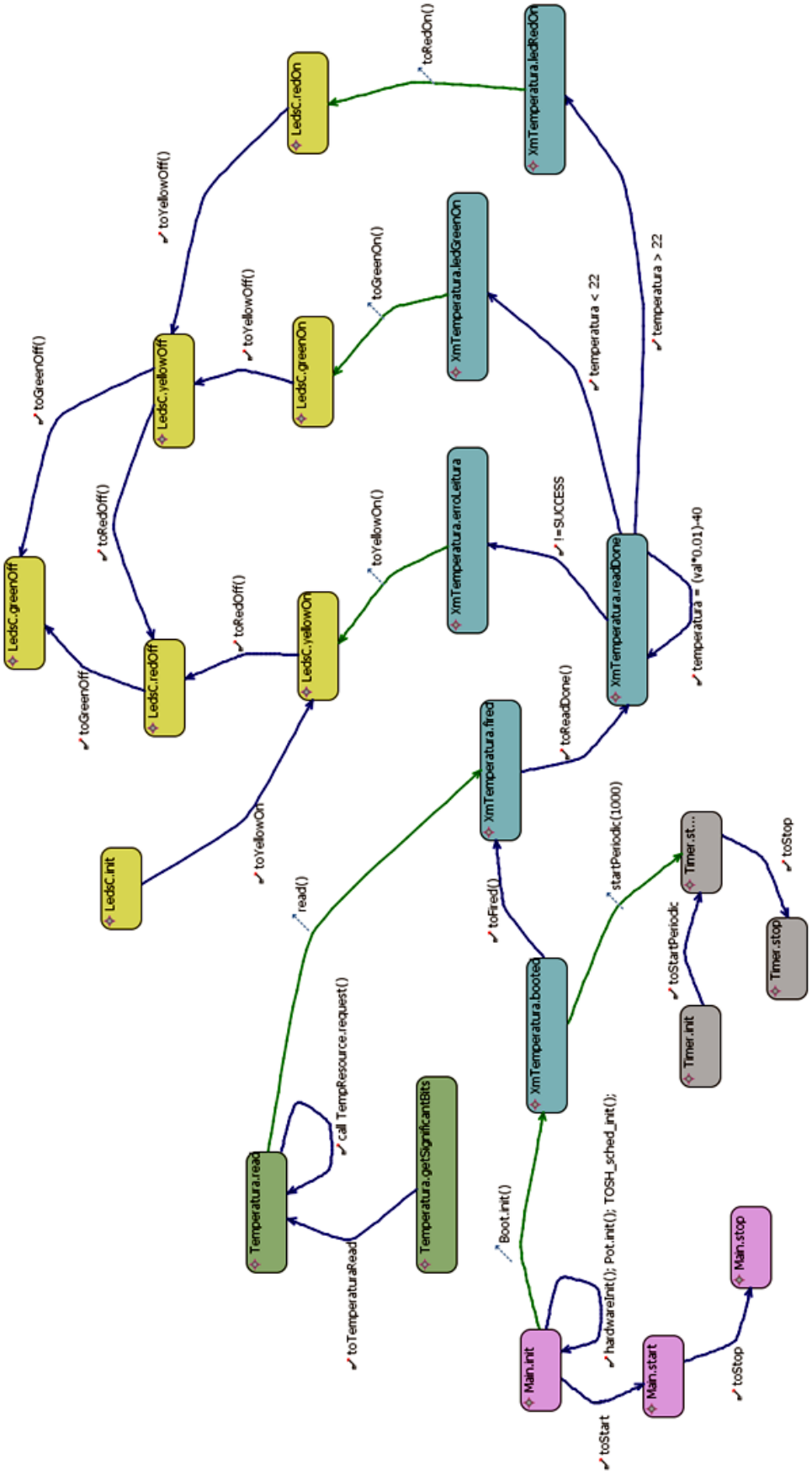


Figura 6.7: Diagrama XmTemperatura gerado no *toolkit Ufam Sensor CommX*.

6.2 Resultados

Nesta seção serão descritos os resultados obtidos através do emprego do *toolkit Ufam Sensor CommX*. Além disso, apresenta-se os códigos gerados em XML e suas respectivas traduções para *nesC* , conseguidas por meio da programação de templates. São apresentados também os testes dos códigos gerado na plataforma *TinyOS* 2.x.

6.2.1 XmBlink

O processo de tradução do código gerado é dado através do uso de templates que se encarregam de fazer a transformação mediante a linguagem de programação escolhida, neste caso *nesC* . A seguir serão apresentados os resultados dos códigos gerados em XML (Código 6.1) e em *nesC* , com seus respectivos arquivos, configuração (configuration) e módulo (module) (Figuras 6.8 e 6.9).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Commx initial-state="XmBlink.init" model-name="XmBlink">
3   <state name="Main.init"/>
4   <state name="Main.start"/>
5   <state name="Main.stop"/>
6   <state name="SglTimer.init"/>
7   <state name="SglTimer.start"/>
8   <state name="SglTimer.stop"/>
9   <state name="XmBlink.init"/>
10  <state name="XmBlink.start"/>
11  <state name="XmBlink.stop"/>
12  <state name="LedsC.init"/>
13  <state name="LedsC.redToogle"/>
14  <state name="LedsC.redOff"/>
15  <state name="LedsC.redOn"/>
16  <message functionName="StdControl.init()" readsFrom="Main.init"
        writesTo="XmBlink.init"/>
17  <message functionName="StdControl.start()" readsFrom="Main.start"
        writesTo="XmBlink.start"/>
18  <message functionName="StdControl.stop()" readsFrom="Main.stop"
        writesTo="XmBlink.stop"/>
19  <message functionName="StdControl.start()" readsFrom="Main.init"
        writesTo="SglTimer.init"/>
20  <message functionName="Timer.start(TIMER_REPEAT,1000)" readsFrom="
        XmBlink.start" writesTo="SglTimer.start"/>
21  <message functionName="Timer.stop()" readsFrom="XmBlink.stop" writesTo
```

```
    ="SglTimer.stop"/>
22  <message functionName="Leds.init()" readsFrom="XmBlink.init" writesTo=
        "LedsC.init"/>
23  <message functionName="Leds.redToogle()" readsFrom="XmBlink.start"
        writesTo="LedsC.redToogle"/>
24  <message functionName="SUCCESS()" readsFrom="LedsC.redToogle" writesTo
        ="XmBlink.start"/>
25  <transition fromState="Main.init" functionName="hardwareInit(); Pot.
        init(); TOSH_sched_init();" toState="Main.init"/>
26  <transition fromState="Main.init" functionName="toStart()" toState="
        Main.start"/>
27  <transition fromState="Main.start" functionName="toStop()" toState="
        Main.stop"/>
28  <transition fromState="SglTimer.init" functionName="toStart()" toState
        ="SglTimer.start"/>
29  <transition fromState="SglTimer.start" functionName="toStop()" toState
        ="SglTimer.stop"/>
30  <transition fromState="XmBlink.init" functionName="toStart()" toState=
        "XmBlink.start"/>
31  <transition fromState="XmBlink.start" functionName="toStop()" toState=
        "XmBlink.stop"/>
32  <transition fromState="LedsC.init" functionName="toRedToogle()"
        toState="LedsC.redToogle"/>
33  <transition fromState="LedsC.redToogle" functionName="toRedOff()"
        toState="LedsC.redOff"/>
34  <transition fromState="LedsC.redOff" functionName="rval" toState="
        LedsC.redToogle"/>
35  <transition fromState="LedsC.redToogle" functionName="toRedOn()"
        toState="LedsC.redOn"/>
36  <transition fromState="LedsC.redOn" functionName="rval" toState="LedsC
        .redToogle"/>
37 </Commx>
```

Código 6.1: XmBlink expresso em XML.

XmBlink.nc XmBlinkM.nc

```
/**
 * Este código foi gerado automaticamente por Ufam Sensor CommX Toolkit em Ter 08/11/2011 11:01:09
 * Pós-Graduação em Engenharia Elétrica - Universidade Federal do Amazonas
 * Geração Automática de Código para Redes de Sensores Sem Fio Usando Communicating X-Machine
 *
 * @Marcus Braga
 **/


configuration XmBlink{
}

implementation{

    components Main, SingleTimer, XmBlinkM, LedsC;

    Main.StdControl -> XmBlinkM.StdControl;
    Main.StdControl -> SingleTimer.StdControl;
    BlinkM.Timer -> SingleTimer.Timer;
    BlinkM.Leds -> LedsC;
}
```

Figura 6.8: Código de configuração XmBlink em *nesC* .

XmBlink.nc XmBlinkM.nc

```
/**
 * Este código foi gerado automaticamente por Ufam Sensor CommX Toolkit em Ter 08/11/2011 11:01:09
 * Pós-Graduação em Engenharia Elétrica - Universidade Federal do Amazonas
 * Geração Automática de Código para Redes de Sensores Sem Fio Usando Communicating X-Machine
 *
 * @Marcus Braga
 **/


module XmBlinkM{
   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();
```

Figura 6.9: Código do módulo XmBlinkM em *nesC* .

A partir dos códigos gerados na ferramenta, inicia-se outra etapa dos testes, que se refere a comprovação que os mesmos serão compilados no *TinyOS*. A aplicação foi compilada para sensor mica2. A Figura 6.10 mostra o processo de compilação relatado. Também foi criada uma pasta Commx na árvore da plataforma *tinyos-2.x*, conforme pode ser visto na Figura 6.11.

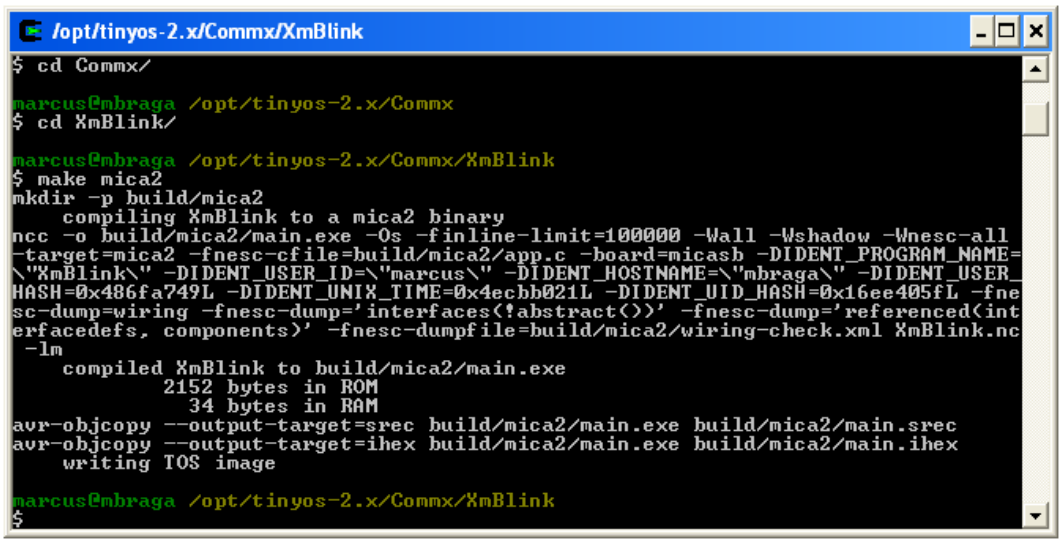


Figura 6.10: Aplicação *XmBlink* compilada em *TinyOS*.

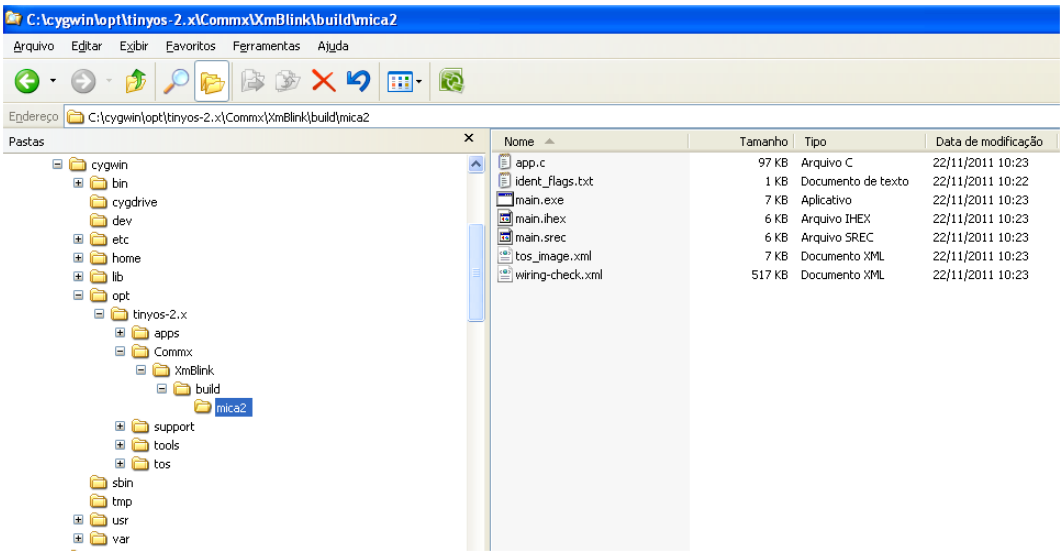


Figura 6.11: Código *XmBlink* gerado em *nesC* para a plataforma de sensores mica2.

6.2.2 XmUmidade

Os resultados obtidos após a geração de código em *nesC* serão apresentados a seguir, no Código 6.2 da aplicação XmUmidade expressa em XML e os códigos de configuração e do módulo (Figura 6.12 e 6.13).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Commx initial-state="XmUmidade.booted" model-name="XmUmidade">
3   <state name="Main.init"/>
4   <state name="Main.start"/>
5   <state name="Main.stop"/>
6   <state name="Timer.init"/>
7   <state name="Timer.startPeriodic()"/>
8   <state name="Timer.stop"/>
9   <state name="Umidade.getSignificantBits"/>
10  <state name="Umidade.read"/>
11  <state name="XmUmidade.booted"/>
12  <state name="XmUmidade.fired"/>
13  <state name="XmUmidade.readDone"/>
14  <state name="XmUmidade.erroLeitura"/>
15  <state name="LedsC.redOn"/>
16  <state name="LedsC.greenOff"/>
17  <state name="LedsC.YellowOff"/>
18  <state name="XmUmidade.altaUmidade"/>
19  <state name="LedsC.greenOn"/>
20  <state name="LedsC.redOff"/>
21  <state name="XmUmidade.baixaUmidade"/>
22  <state name="LedsC.yellowOn"/>
23  <state name="LedsC.init"/>
24  <message functionName="Boot.init()" readsFrom="Main.init" writesTo="
        XmUmidade.booted"/>
25  <message functionName="startPeriodic(1000)" readsFrom="XmUmidade.
        booted" writesTo="Timer.startPeriodic()"/>
26  <message functionName="read()" readsFrom="Umidade.read" writesTo="
        XmUmidade.fired"/>
27  <message functionName="toRedOn()" readsFrom="XmUmidade.erroLeitura"
        writesTo="LedsC.redOn"/>
28  <message functionName="toGreenOn()" readsFrom="XmUmidade.altaUmidade"
        writesTo="LedsC.greenOn"/>
29  <message functionName="toYellowOn()" readsFrom="XmUmidade.baixaUmidade
        " writesTo="LedsC.yellowOn"/>
30  <transition fromState="Main.init" functionName="hardwareInit(); Pot.
        init(); TOSH_sched_init();" toState="Main.init"/>
31  <transition fromState="Main.init" functionName="toStart()" toState="
        Main.start"/>
32  <transition fromState="Main.start" functionName="toStop()" toState="
        Main.stop"/>
33  <transition fromState="Timer.init" functionName="toStartPeriodic()"
```

```
        toState="Timer.startPeriodic()"/>
34  <transition fromState="Timer.startPeriodic()" functionName="toStop()"
        toState="Timer.stop"/>
35  <transition fromState="Umidade.read" functionName="call HumResource.
        request()" toState="Umidade.read"/>
36  <transition fromState="Umidade.getSignificantBits" functionName="
        toUmidadeRead" toState="Umidade.read"/>
37  <transition fromState="XmUmidade.booted" functionName="toFired()"
        toState="XmUmidade.fired"/>
38  <transition fromState="XmUmidade.fired" functionName="toReadDone"
        toState="XmUmidade.readDone"/>
39  <transition fromState="XmUmidade.readDone" functionName="umidade =
        0.0405*val+((-2.8)*0.000001)*(val*val)-4;" toState="XmUmidade.
        readDone"/>
40  <transition fromState="XmUmidade.readDone" functionName="!=SUCCESS"
        toState="XmUmidade.erroLeitura"/>
41  <transition fromState="LedsC.redOn" functionName="toGreenOff()"
        toState="LedsC.greenOff"/>
42  <transition fromState="LedsC.greenOff" functionName="toYellowOff()"
        toState="LedsC.YellowOff"/>
43  <transition fromState="XmUmidade.readDone" functionName="umidade > 50"
         toState="XmUmidade.altaUmidade"/>
44  <transition fromState="LedsC.greenOn" functionName="toRedOff()"
        toState="LedsC.redOff"/>
45  <transition fromState="XmUmidade.readDone" functionName="umidade &lt;
        50" toState="XmUmidade.baixaUmidade"/>
46  <transition fromState="LedsC.redOff" functionName="toGreenOff()"
        toState="LedsC.greenOff"/>
47  <transition fromState="LedsC.yellowOn" functionName="toRedOff()"
        toState="LedsC.redOff"/>
48  <transition fromState="LedsC.redOff" functionName="toYellowOff()"
        toState="LedsC.YellowOff"/>
49  <transition fromState="LedsC.init" functionName="toRedOn()" toState="
        LedsC.redOn"/>
50 </Commx>
```

Código 6.2: XmUmidade expresso em XML.

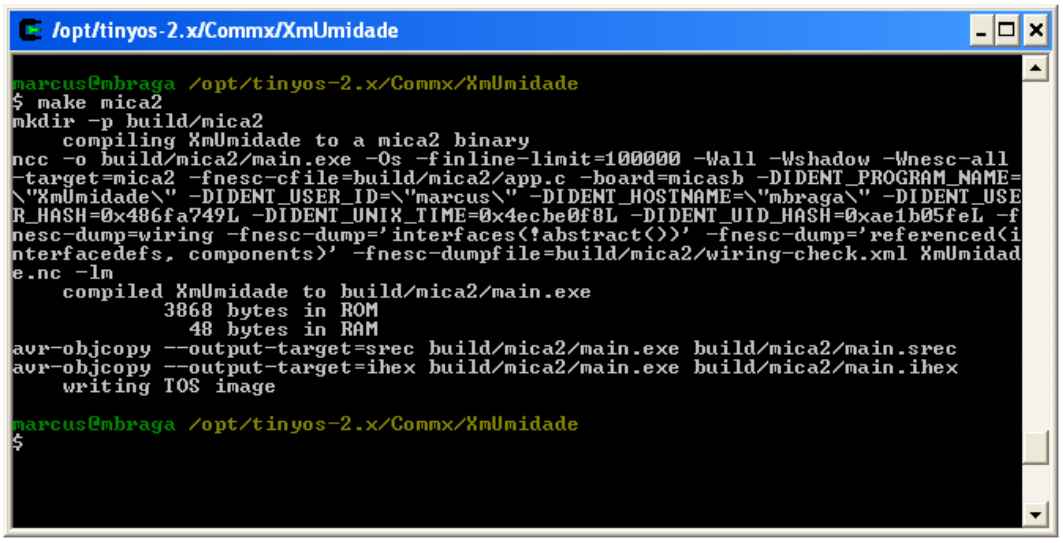


Figura 6.12: Código do módulo XmUmidade em *nesC* .

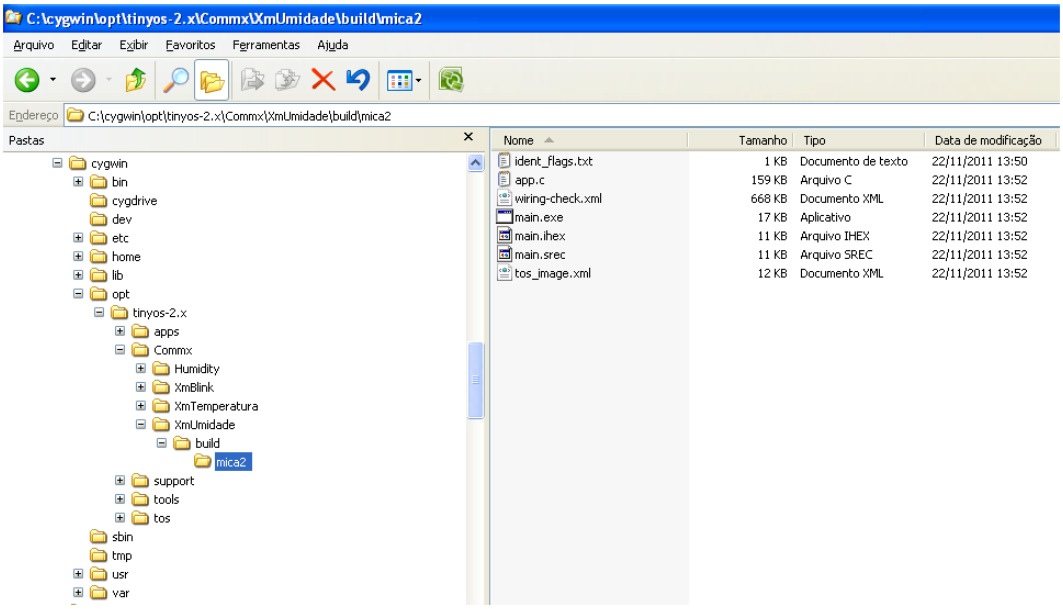


Figura 6.13: Código do módulo XmUmidadeM em *nesC* .

As Figuras 6.14 e 6.15 mostram os testes dos códigos gerados pela ferramenta para serem compilados pelo *TinyOS* e sua respectiva pasta contendo o código para ser inserido no sensor.

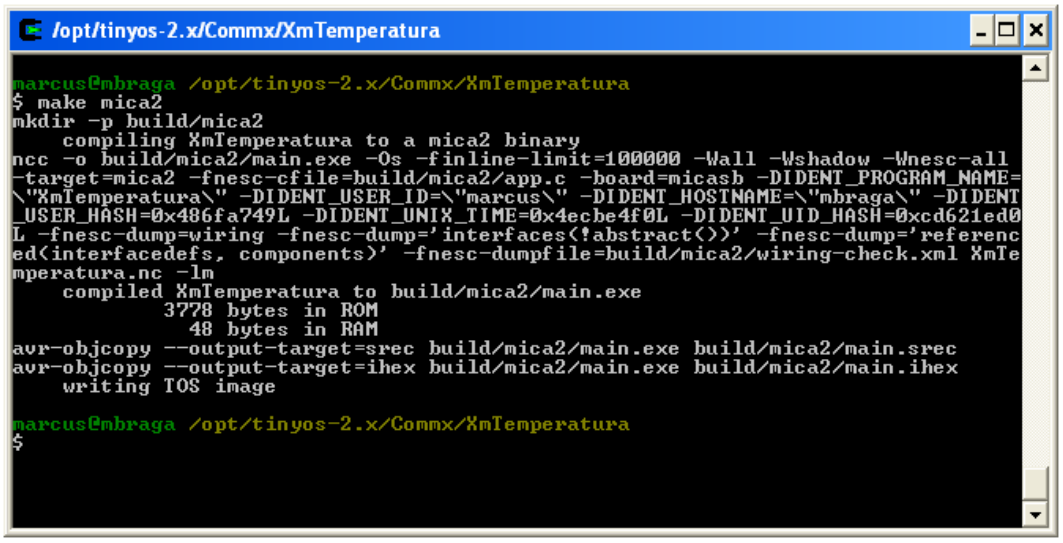


Figura 6.14: Aplicação XmUmidade compilada em *TinyOS*.

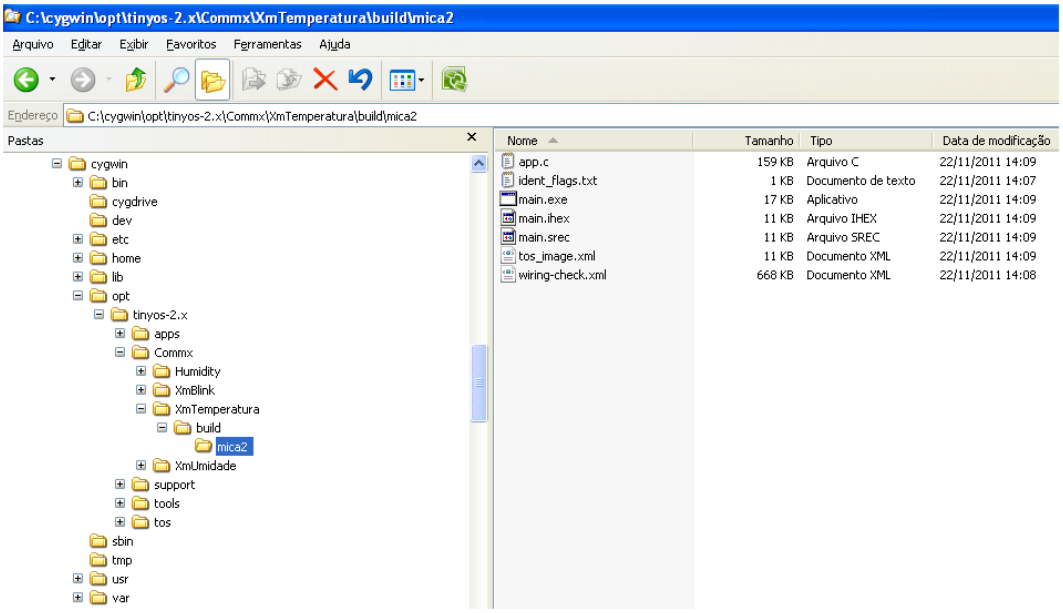


Figura 6.15: Código XmUmidade gerado em *nesC* para a plataforma de sensores mica2.

6.2.3 XmTemperatura

A seguir serão apresentados o Código 6.3 gerado em XML e os códigos de configuração e do módulo em *nesC* (Figuras 6.16 e 6.17).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Commx initial-state="XmTemperatura.booted" model-name="XmTemperatura">
3   <state name="Main.init"/>
4   <state name="Main.start"/>
5   <state name="Main.stop"/>
```

```
6   <state name="Temperatura.read"/>
7   <state name="Temperatura.getSignificantBits"/>
8   <state name="Timer.init"/>
9   <state name="Timer.startPeriodic()"/>
10  <state name="Timer.stop"/>
11  <state name="XmTemperatura.booted"/>
12  <state name="XmTemperatura.fired"/>
13  <state name="XmTemperatura.readDone"/>
14  <state name="XmTemperatura.erroLeitura"/>
15  <state name="LedsC.yellowOn"/>
16  <state name="LedsC.redOff"/>
17  <state name="LedsC.greenOff"/>
18  <state name="XmTemperatura.ledGreenOn"/>
19  <state name="LedsC.greenOn"/>
20  <state name="LedsC.yellowOff"/>
21  <state name="XmTemperatura.ledRedOn"/>
22  <state name="LedsC.redOn"/>
23  <state name="LedsC.init"/>
24  <message functionName="Boot.init()" readsFrom="Main.init" writesTo="
        XmTemperatura.booted"/>
25  <message functionName="startPeriodic(1000)" readsFrom="XmTemperatura.
        booted" writesTo="Timer.startPeriodic()"/>
26  <message functionName="read()" readsFrom="Temperatura.read" writesTo="
        XmTemperatura.fired"/>
27  <message functionName="toYellowOn()" readsFrom="XmTemperatura.
        erroLeitura" writesTo="LedsC.yellowOn"/>
28  <message functionName="toGreenOn()" readsFrom="XmTemperatura.
        ledGreenOn" writesTo="LedsC.greenOn"/>
29  <message functionName="toRedOn()" readsFrom="XmTemperatura.ledRedOn"
        writesTo="LedsC.redOn"/>
30  <transition fromState="Main.init" functionName="hardwareInit(); Pot.
        init(); TOSH_sched_init();" toState="Main.init"/>
31  <transition fromState="Main.init" functionName="toStart" toState="Main
        .start"/>
32  <transition fromState="Main.start" functionName="toStop" toState="Main
        .stop"/>
33  <transition fromState="Temperatura.read" functionName="call
        TempResource.request()" toState="Temperatura.read"/>
34  <transition fromState="Temperatura.getSignificantBits" functionName="
        toTemperaturaRead" toState="Temperatura.read"/>
35  <transition fromState="Timer.init" functionName="toStartPeriodic"
        toState="Timer.startPeriodic()"/>
36  <transition fromState="Timer.startPeriodic()" functionName="toStop"
```

```
        toState="Timer.stop"/>
37  <transition fromState="XmTemperatura.booted" functionName="toFired()"
        toState="XmTemperatura.fired"/>
38  <transition fromState="XmTemperatura.fired" functionName="toReadDone()
        " toState="XmTemperatura.readDone"/>
39  <transition fromState="XmTemperatura.readDone" functionName="
        temperatura = (val*0.01)-40" toState="XmTemperatura.readDone"/>
40  <transition fromState="XmTemperatura.readDone" functionName="!=SUCCESS
        " toState="XmTemperatura.erroLeitura"/>
41  <transition fromState="LedsC.yellowOn" functionName="toRedOff()"
        toState="LedsC.redOff"/>
42  <transition fromState="LedsC.redOff" functionName="toGreenOff" toState
        ="LedsC.greenOff"/>
43  <transition fromState="XmTemperatura.readDone" functionName="
        temperatura &lt; 22" toState="XmTemperatura.ledGreenOn"/>
44  <transition fromState="LedsC.greenOn" functionName="toYellowOff()"
        toState="LedsC.yellowOff"/>
45  <transition fromState="LedsC.yellowOff" functionName="toRedOff()"
        toState="LedsC.redOff"/>
46  <transition fromState="XmTemperatura.readDone" functionName="
        temperatura > 22" toState="XmTemperatura.ledRedOn"/>
47  <transition fromState="LedsC.redOn" functionName="toYellowOff()"
        toState="LedsC.yellowOff"/>
48  <transition fromState="LedsC.yellowOff" functionName="toGreenOff()"
        toState="LedsC.greenOff"/>
49  <transition fromState="LedsC.init" functionName="toYellowOn" toState="
        LedsC.yellowOn"/>
50 </Commx>
```

Código 6.3: XmTemperatura expresso em XML.

XmTemperatura.nc | XmTemperaturaM.nc

```
/**
 * Este código foi gerado automaticamente por Ufam Sensor CommX Toolkit em Seg 07/11/2011 11:54:38
 * Pós-Graduação em Engenharia Elétrica - Universidade Federal do Amazonas
 * Geração Automática de Código para Redes de Sensores Sem Fio Usando Communicating X-Machine
 *
 * @Marcus Braga
 **/


configuration XmTemperatura{
}

implementation{

    components Main, Temperatura, new TimerMillic() as Timer, XmTemperaturaM, LedsC;

    XmTemperatura.SenseTimer -> Timer0;
    XmTemperatura -> Main.boot;
}
```

Figura 6.16: Código de configuração XmTemperatura em *nesC* .

XmTemperatura.nc | XmTemperaturaM.nc

```
/**
 * Este código foi gerado automaticamente por Ufam Sensor CommX Toolkit em Seg 07/11/2011 11:54:39
 * Pós-Graduação em Engenharia Elétrica - Universidade Federal do Amazonas
 * Geração Automática de Código para Redes de Sensores Sem Fio Usando Communicating X-Machine
 *
 * @Marcus Braga
 **/

#include "Timer.h"

module XmTemperaturaM{
   uses{
      interface Boot;
      interface Timer<TMilli> as SenseTimer;
      interface Read<uint16_t> as ReadUmidade1;
   }
}
implementation{

 event void Boot.booted() {
      SenseTimer.startPeriodic(1000);{
 }

 event void SenseTimer.fired() {
      call ReadTemperatura.read();{
 }
 event void ReadTemperatura.readDone(error_t result, uint16_t val) {
      float temperatura = (val*0.01)-40;
      if (result != SUCCESS) {
         call Leds.yellowOn();call Leds.redOff();call Leds.greenOff();
      } else if (temperatura < 22)  {
         call Leds.greenOn();call Leds.redOff();call Leds.yellowOff();
      } else {
         call Leds.redOn();call Leds.greenOff();call Leds.yellowOff();
```

Figura 6.17: Código do módulo XmTemperaturaM em *nesC* .

A seguir serão apresentados os resultados dos testes com o compilador do *TinyOS*. As Figuras 6.18 e 6.19 mostram o resultado dos testes com o compilador e com o código

compilado respectivamente.

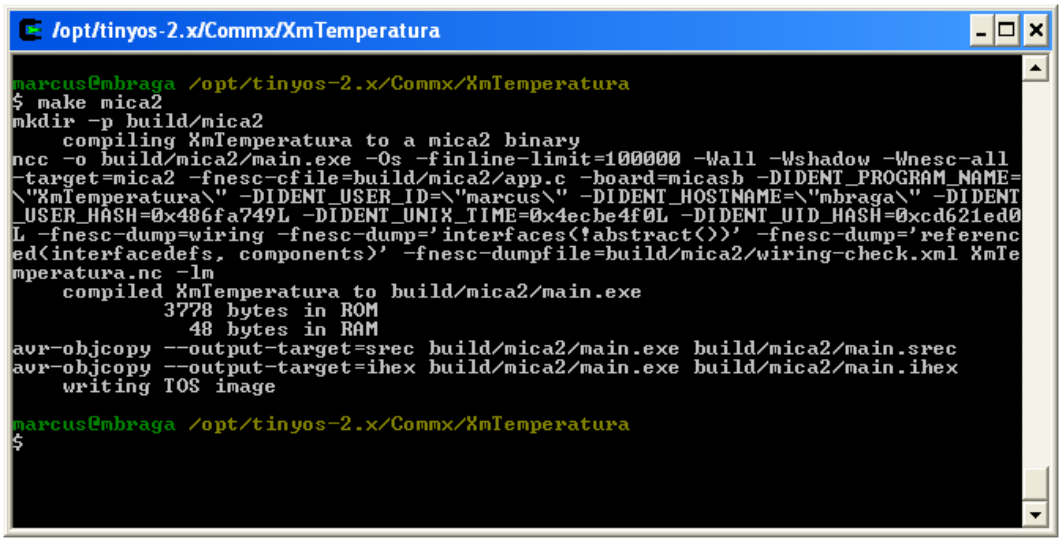


Figura 6.18: Aplicação XmTemperatura compilada em *TinyOS*.

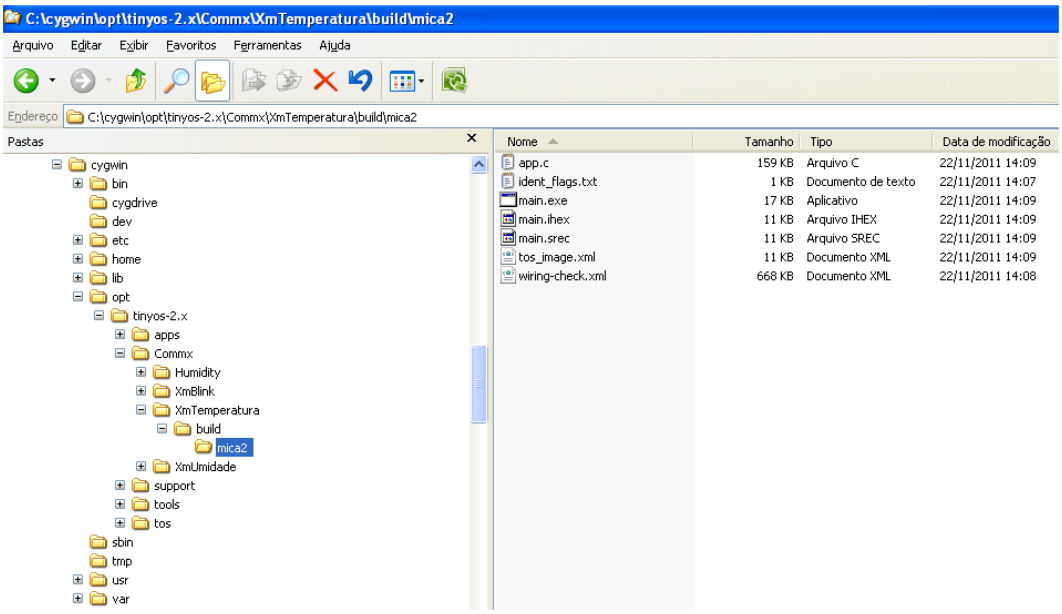


Figura 6.19: Código XmTemperatura gerado em *nesC* para a plataforma de sensores mica2.

6.3 Resultados alcançados

Com base nos resultados obtidos com a implementação dos casos de uso propostos, pode-se notar que a metodologia empregada destaca-se de modo positivo das abordagens de outras ferramentas apresentadas no Capítulo 3 por ser intuitiva e favorecer o aumento no nível de abstração, permitindo que o desenvolvedor se ocupe com a lógica da aplicação.

Alguns pontos interessantes podem ser destacados na utilização do *toolkit Ufam Sensor CommX*, são eles:

- O uso da ferramenta facilitou o processo desde a modelagem até a geração de código.
- A ferramenta possibilita a construção de aplicações por partes, utilizando composição por componentes. Sendo que ao fim do processo, bastando construir comunicação entre os componentes. Isto favorece um melhor entendimento do problema e conseqüente decomposição de sua complexidade.
- As visões de código proporcionadas pela ferramenta, XML e árvore de componentes, é outro ponto a ser destacado. Permite que desenvolvedor tenha outros pontos-de-vista sobre a aplicação modelada.
- A utilização do código XML nos templates favorece a geração de código para outras linguagens de programação, permitindo que a ferramenta tenha melhor aceitação no desenvolvimento de aplicações RSSF.

6.4 Resumo

O principal objetivo deste capítulo foi avaliar a ferramenta *Ufam Sensor CommX* descrita e implementada nos capítulos anteriores. Para isto, foram implementados os casos de uso propostos no Capítulo 4: (a) XmBlink que controla a frequência que um *led* pisca; (b) XmUmidade; e (c) XmTemperatura que fazem parte do cenário de monitoramento de pacientes em um hospital.

Nestes cenários, foi utilizado o *toolkit* na criação e conseqüente geração de código para RSSF, foram observadas visões de código, em XML e árvore de componentes. Por fim, realizou-se, teste utilizando o compilador do *TinyOS*, de modo a testar os códigos gerados na ferramenta.

No capítulo a seguir serão feitas as considerações finais sobre esta pesquisa, destacando suas contribuições e sua importância, além de identificar melhorias e trabalhos futuros.

Capítulo 7
Considerações Finais

O principal objetivo proposto por esta pesquisa foi a construção de uma solução para auxiliar na criação de aplicações para RSSF, utilizando o método formal CXM desde a modelagem até a geração de código. Para isto foi criada inicialmente uma metodologia de desenvolvimento e em seguida foi feita a automação da mesma usando o *toolkit Ufam Sensor CommX*, proporcionando ao desenvolvedor um suporte desde a modelagem até a geração de código.

Para o desenvolvimento do *Ufam Sensor CommX* foi feito um estudo em RSSF, Métodos Formais e MDD. Foram abordados conceitos, modelos de programação, sistemas operacionais e linguagem de programação *nesC*. Com relação a Métodos Formais, foram descritos conceitos sobre máquina de computação geral, métodos formais e definições de XM e sua extensão proposta neste trabalho CXM. Em MDD foram apresentados conceitos, relação da plataforma Eclipse com MDD e arquitetura clássica de modelagem. Os conceitos apresentados foram de grande valia no desenvolvimento deste trabalho, além de fornecer conhecimento sobre o estado da arte das áreas em questão.

Em seguida, este trabalho analisou as abordagens de alguns trabalhos correlatos existentes na literatura. Os trabalhos foram avaliados de acordo com os aspectos (1) interface gráfica; (2) geração de código; (3) plataforma de atuação; (4) tipo de abordagem; e (5) verificação de código. A partir destas observações foi montado um comparativo entre os trabalhos relatados e as características desejadas.

A integração da RSSF com o método formal CXM foi realizada através da criação de um modelo que permite a construção de aplicações RSSF até a geração de código. A geração do código foi conseguida através de um processo composto por três etapas: (1) modelagem em componentes; (2) comunicação entre os componentes; e (3) geração de código através das traduções expressas em XMDL para XML e em seguida XML para *nesC*. Para testar a integração foram propostos três estudos de caso, (1) XmBlink que controla o funcionamento de um led durante o intervalo de tempo; (2) XmUmidade, sensor

de umidade; e (3) XmTemperatura, sensor de temperatura.

Em seguida, foi construído na plataforma Eclipse o *toolkit Ufam Sensor CommX*, onde se automatiza o processo de criação de aplicações RSSF descritos anteriormete. Os casos de uso foram implementados utilizando a ferramenta automática gerada. O *toolkit* trouxe a figura do *template* como gerador de código através das especificações da aplicação em XML. Esta peculiaridade permite a geração de código para diversas linguagens de programação. No entanto, para esta pesquisa, *nesC* foi a linguagem escolhida por se tratar da linguagem utilizada na plataforma RSSF mais difundida, o *TinyOS*.

Testes e provas de conceito e resultados fazem parte da avaliação da ferramenta *Ufam Sensor CommX*. Para isso foram avaliados os três casos de uso propostos anteriormente e foram avaliados os códigos resultantes: (1) XML; (2) árvore de componentes; (3) *nesC* (*module* e *configuration*). O primeiro código serviu de base para o gerador de código (JET2), o segundo proporcionou uma nova abstração ao desenvolvedor a respeito da aplicação modelada. O código em *nesC* proporcionou dois arquivos com extensão "*.nc", onde ambos foram essenciais para o funcionamento do nó sensor. Todos os códigos gerados pela ferramenta foram testados no compilador TinyOS, compilados para o sensor mica2 e gerando os respectivos executáveis para serem baixados para o nó em questão.

A conclusão é que o *toolkit Ufam Sensor CommX* obteve sucesso em relação aos objetivos definidos no início deste trabalho: os principais *frameworks* e geradores de código para RSSF foram avaliados, de modo a extrair as melhores características para compor uma nova ferramenta. Foi elaborado um método de tradução das especificações modeladas através do método CXM, convertendo inicialmente XMDL para XML e, por conseguinte para *nesC*. O *framework* de geração de editores GMF serviu de base para construção do *toolkit Ufam Sensor CommX*, proporcionando uma interface mais intuitiva e amigável ao desenvolvedor, favorecendo a utilização da ferramenta e tendo como resultado o código em *nesC* do modelo. O *toolkit* trouxe três tipos de visões da aplicação para o programador: (1) representação gráfica da aplicação utilizando autômatos que se comunicam (componentes); (2) representação da aplicação em formato de árvore de componentes; e (3) código XML, que proporciona tradução para outras linguagens de programação.

Espera-se que a presente pesquisa venha a contribuir para a literatura e o desenvolvimento de novas aplicações para RSSF e outros estudos de maior complexidade, como sistemas autonômicos e para possíveis trabalhos acadêmicos sejam de artigos, monografias e mesmo dissertações.

7.1 Problemas encontrados

Ao longo do desenvolvimento desta pesquisa alguns empencilhos se apresentaram e serão relatados a seguir. Um dos primeiros problemas enfrentados foi encontrar subsídios concretos na literatura que norteassem a construção da ferramenta. A maioria dos trabalhos estudados não trazia informações mais precisas sobre os passos a serem executados ou quais direções a seguir na construção da ferramenta. Essa etapa foi superada fazendo uso de listas de discussões na web, onde desenvolvedores de vários níveis trocam vários tipos de informação, o que torna um poderoso aliado para o desenvolvimento de pesquisas.

Para o desenvolvimento da ferramenta na plataforma Eclipse foi utilizado do projeto Modeling que contempla os plug-ins GMF, GEF, EMF e JET. Porém um dos problemas encontrados na sua utilização foi em relação às versões dos plug-ins utilizados, implicando em incompatibilidade no seu funcionamento. Tais observações só foram percebidas quando de seu funcionamento, pois não foi gerada nenhuma advertência nas compilações.

O referido *toolkit* foi desenvolvido para o *TinyOS* 2.x, sistema operacional que apresenta instabilidade e não possui suporte de ferramentas de simulação que sua versão anterior trazia. Mais uma vez foi utilizado o recurso das listas de discussão para sobrepor mais esta barreira.

7.2 Melhorias e sugestões para trabalhos futuros

Esta pesquisa resultou na elaboração de um *toolkit* de modelagem e geração de código para RSSF utilizando método formal CXM e MDD. A seguir serão descritas possíveis contribuições para o desenvolvimento do *toolkit Ufam Sensor CommX*. Uma possível extensão deste trabalho refere-se à melhoria do editor gráfico, aumentando o nível dos componentes de modo a permitir o encapsulamento dos mesmos como um único componente, conforme apresentado nos casos de uso do

Capítulo 4. Desta forma, é possível representar melhor a XM como componente. Detalhes como memória (M), portas e canais de input e output, entre outras características, estariam mais explícitas e disponíveis para o usuário. Outra sugestão de trabalho refere-se à adequação automática do código mediante escolha da plataforma de sensores, bem com suas respectivas linguagens de programação, pois o desenvolvimento rápido da microeletrônica contribui para o surgimento de novas plataformas, modelos de sensores e

conseqüentes linguagens de programação.

Um possível trabalho futuro consiste na implementação de um simulador agregado ao editor gráfico, permitindo com que as aplicações construídas sejam simuladas/testadas de modo a permitir uma aplicação mais livre de falhas e com código mais enxuto.

Por fim, uma possível extensão desta pesquisa consiste na verificação do código gerado em *nesC* ou outra linguagem, com intuito de garantir a corretude no código desta linguagem alvo.

Referências

ABDUNABI, T. *X-Machine Toolkit within Eclipse Platform*. Dissertação (Mestrado) — Department of Computer Science, University of Sheffield, Reino Unido, 2007.

ABRACH, H.; BHATTI, S.; CARLSON, J.; DAI, H.; ROSE, J.; SHETH, A.; SHUCKER, B.; DENG, J.; HAN, R. Mantis: System support for multimodal networks of in-situ sensors. In: *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*. New York, NY, USA: ACM, 2003. (WSNA '03, 941358), p. 50–59. Disponível em: <http://doi.acm.org/10.1145/941350.941358>.

AGUADO, J. *Conformance Testing of Distributed Systems: an X-Machine based Approach*. Tese (Doutorado) — Department of Computer Science, University of Sheffield, 2004.

AKYILDIZ, I.; SU, W.; SANKARASUBRAMANIAM, Y.; CAYIRCI, E. A survey on sensor networks. *Communications Magazine, IEEE*, v. 40, n. 8, p. 102 – 114, aug 2002.

AKYILDIZ, I. F.; KASIMOGLU, I. H. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks*, v. 2, n. 4, p. 351 – 367, 2004.

ANISZCZYK, C.; GALLARDO, D. CT316, *Get Started With the Eclipse Platform*. jul. 2007. Disponível em: <http://www.ibm.com/developerworks/opensource/library/os-eclipse-platform/?S_TACT=105AGX59&S_CMP=GR&ca=dgr-eclipse-1>. Acesso em: 31 de agosto de 2011.

ANISZCZYK, C.; MARZ, N. Create more – better – code in eclipse with jet. 2006. Disponível em: <http://www-128.ibm.com/developerworks/opensource/library/os-ecl-jet/>. Acesso em: 05 de setembro de 2011.

BAKSHI, A. B.; PRASANNA, V. K. *Architecture-Independent Programming for Wireless Sensor Networks*. [S.l.]: Wiley, 2008. I-XV, 1-187 p. ISBN 978-0-471-77889-9.

BALANESCU, T.; GHEORGHE, M.; HOLCOMBE, M. Where mathematics, computer science, linguistics and biology meet. In: . Norwell, MA, USA: Kluwer Academic Publishers, 2001. cap. Deterministic Stream X-machines Based on Grammar Systems, p. 13–23. ISBN 0-7923-6693-X. Disponível em: <http://dl.acm.org/citation.cfm?id=374286.374288>.

BARBOSA, T. M. G. A. *Arquitetura de Redes de Sensores do Corpo Humano*. Tese (Doutorado) — Departamento de Engenharia Elétrica, Universidade de Brasília, Brasil, 2008.

BARNARD, J. Comx: A design methodology using communicating x-machines. *Information and Software Technology*, v. 40, n. 56, p. 271 – 280, 1998. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584998000536>.

BARNARD, J.; WHITWORTH, J.; WOODWARD, M. Communicating x-machines. *Information and Software Technology*, v. 38, n. 6, p. 401 – 407, 1996. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/0950584995010661>.

BRAGA, M. L.; SANTOS, A. J.; JUNIOR, V. F. L. Modelagem e geração de código para rede de sensores sem fio. In: *9th International Information and Telecommunication Technologies Symposium*. [S.l.]: I2TS, 2010.

_____. Usando communicating x-machine na construção de aplicações para redes de sensores sem fio. *Anais do IV Simpósio de Modelagem Computacional do Sul*, v. 1, p. 233–238, 2010.

BRANCO, A.; RODRIGUEZ, N. *Macroprogramação em Redes de Sensores Sem Fio*. [S.l.], 2010.

BUDINSKY, F. *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley, 2004. (The Eclipse Series). ISBN 9780131425422. Disponível em: <http://books.google.com.br/books?id=ff-9ZYhvPwwC>.

CHECHIK, M.; GANNON, J. Automatic analysis of consistency between requirements and designs. *Software Engineering, IEEE Transactions on*, v. 27, n. 7, p. 651 –672, jul 2001. ISSN 0098-5589.

CHEONG, E.; LEE, E. A.; ZHAO, Y. Viptos: A graphical development and simulation environment for tinyos-based wireless sensor networks. In: *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*. New York, NY, USA: ACM, 2005. (SenSys '05), p. 302–302. ISBN 1-59593-054-X. Disponível em: <http://doi.acm.org/10.1145/1098918.1098967>.

CLARKE, E.; WING, J. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 28, n. 4, p. 626–643, dez. 1996. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/242223.242257>.

DIAS, A. C. F. *Uma Linguagem Específica do Domínio para Uma Abordagem Orientada aos Objectivos Baseada em KAOS*. Dissertação (Mestrado) — Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal, 2009.

DUNKELS, A.; GRONVALL, B.; VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In: *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. Washington, DC, USA: IEEE Computer Society, 2004. (LCN '04), p. 455–462. ISBN 0-7695-2260-2. Disponível em: <http://dx.doi.org/10.1109/LCN.2004.38>.

ECLIPSE. *Eclipse IDE Project*. 2008. Disponível em: <http://eclipse.org>. Acesso em: 02 de setembro 2011.

EILENBERG, S. *Automata, Languages and Machines*. [S.l.]: Academic Press, 1974. (Pure and Applied Mathematics, V. 4).

ELDER, P. *JET Enhancement Proposal (JET2)*. 2005. Disponível em: <http://www.eclipse.org/modeling/emf/docs/architecture/jet2/jet2.html>. Acesso em: 05 de setembro de 2011.

ELEFTHERAKIS, G. *Formal Verification of X-machine models: Towards Formal Development of Computer-Based Systems*. Tese (Doutorado) — Department of Computer Science, University of Sheffield, Reino Unido, 2003.

EVERS, L. *Concise and Flexible Programming of Wireless Sensor Networks*. University of Twente [Host], 2010. ISBN 9789036530286. Disponível em: <http://books.google.com.br/books?id=gb95kgEACAAJ>.

FAIRTLOUGH, M.; HOLCOMBE, M.; IPATE, F.; JORDAN, C.; LAYCOCK, G.; ZHENHUA, D. *Using an X-Machine to Model a Video Cassette Recorder*. [S.l.: s.n.], 2005.

FERREIRA, M. A. R. *Desenvolvimento de um Plug-in Eclipse para Modelagens de Aplicações Geoespaciais*. Dissertação (Mestrado) — Universidade do Vale do Itajaí, 2009.

FIGUEIREDO, C.; NEVES, J.; MAGALHãES, L.; PINTO, V. *Especificação Formal de Software*. Portugal, 2002.

FONSECA, A. d. A. *Desenvolvimento de um Plugin para Composição de Serviços Web utilizando a Plataforma Eclipse*. Salvador, Brasil, 2008.

FRöEHLICH, A. A.; WANNER, L. F. Operating system support for wireless sensor networks. In: JOURNAL OF COMPUTER SCIENCES. *Journal of Computer Sciences*. [S.l.]: Journal of Computer Sciences, 2008. v. 4, p. 272–281.

GALLARDO, D. *Developing eclipse plug-ins: How to create, debug, and install your plug-in*. [S.l.], Dezembro 2002.

GAY, D.; LEVIS, P.; BEHREN, R. von; WELSH, M.; BREWER, E.; CULLER, D. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 38, n. 5, p. 1–11, maio 2003. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/780822.781133>.

GEF. *Graphical Editing Framework, 2011*. 2011. Disponível em: <http://www.eclipse.org/gef/>. Acesso em: 02 de setembro de 2011.

GMF. *Graphical Modeling Framework*. 2011. Disponível em: <http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1>. Acesso em: 02 de setembro de 2011.

GONçALVES, P. R. P. *Monitoramento Remota de Pacientes em Ambulatório*. Dissertação (Mestrado) — Universidade Fernando Pessoa, 2008. Disponível em: <https://bdigital.ufp.pt/dspace/bitstream/10284/1226/3/DM-PauloGoncalves.pdf>. Acesso em: 02 de setembro de 2011.

GUCLU, K. *A First Look at Eclipse Plug-In Programming*. 2008. Disponível em: <http://www.developer.com/open/article.php/3316241/A-First-Look-at-Eclipse-Plug-In-Programming.htm>. Acesso em: 20 de agosto. 2011.

HAILPERN, B.; TARR, P. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, v. 45, n. 3, p. 451 –461, 2006. ISSN 0018-8670.

HAN, C.-C.; KUMAR, R.; SHEA, R.; KOHLER, E.; SRIVASTAVA, M. A dynamic operating system for sensor nodes. In: *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2005. (MobiSys '05), p. 163–176. ISBN 1-931971-31-5. Disponível em: <http://doi.acm.org/10.1145/1067170.1067188>.

HILL, J.; HORTON, M.; KLING, R.; KRISHNAMURTHY, L. The platforms enabling wireless sensor networks. *Commun. ACM*, ACM, New York, NY, USA, v. 47, n. 6, p. 41–46, jun. 2004. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/990680.990705>. Acesso em: 20 jun. 2011.

HOLCOMBE, M.; HOLCOMBE, W.; IPATE, F. *Correct systems: building a business process solution*. Springer, 1998. (Applied computing). ISBN 9783540762461. Disponível em: <http://books.google.com.br/books?id=ndFTAAAAMAAJ>.

IPATE, F. E. *Theory of X-machines and Applications in Specification and Testing*. Tese (Doutorado) — Department of Computer Science, University of Sheffield, 1995.

JET. *Java Emiter Template*. 2011. Disponível em: <http://www.eclipse.org/modeling/m2t/?project=jet>. Acesso em: 05 e setembro de 2011.

KALLOE, A. *Programming Heterogeneous Wireless Sensor Networks*. Dissertação (Mestrado) — Departamento de Matemática e Ciência da Computação, Universidade de Tecnologia de Eindhoven, 2008.

KASTEN, O. *A State-Based Programming Model for Wireless Sensor Networks*. Tese (Doutorado) — Swiss Federal Institute of Technology Zurich (ETH Zurich), 2007.

KEFALAS, P.; ELEFTHERAKIS, G.; KEHRIS, E. *XMDL User Manual: version 1.6*. Reino Unido, 2001.

_____. Communicating x-machines: a practical approach for formal and modular specification of large systems. *Information and Software Technology*, v. 45, n. 5, p. 269 – 280, 2003. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584903000181>.

_____. Communicating x-machines: from theory to practice. In: *Proceedings of the 8th Panhellenic conference on Informatics*. Berlin, Heidelberg: Springer-Verlag, 2003. (PCI'01), p. 316–335. ISBN 3-540-3-540-07544-5. Disponível em: <http://dl.acm.org/citation.cfm?id=1756269.1756290>.

LAMSWEERDE, A. v. Formal specification: a roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000. (ICSE '00), p. 147–159. ISBN 1-58113-253-0. Disponível em: <http://doi.acm.org/10.1145/336512.336546>.

LAYCOCH, G. T. *The Teory and Practice of Specification Based Software Testing*. Tese (Doutorado) — Department of Computer Science, University of Sheffield, Reino Unido, 1993.

LIM, J. B. *RaPTEX: Rapid Prototyping Tool for Embedded Communication Systems.* Dissertação (Mestrado) — North Carolina State University, Department Science Computer, Estados Unidos, 2007.

LUCRéDIO, D. *Uma Abordagem Orientada a Modelos para Reutilização de Software.* Tese (Doutorado) — Instituto de Ciências Matemáticas e de Computação – ICMC, Universidade de São Paulo – USP, Brasil, 2009.

LYMBERIS, A.; DITTMAR, A. Advanced wearable health systems and applications - research and development efforts in the european union. *Engineering in Medicine and Biology Magazine, IEEE*, v. 26, n. 3, p. 29 –33, may-june 2007. ISSN 0739-5175.

MAN, K.; VALLEE, T.; LEUNG, H.; MERCALDI, M.; WULP, J. van der; DONNO, M.; PASTRNAK, M. Tepawsn - a tool environment for wireless sensor networks. In: *Industrial Electronics and Applications, 2009. ICIEA 2009. 4th IEEE Conference on.* [S.l.: s.n.], 2009. p. 730 –733.

MANTIS. *MANTIS Project.* set 2006. Disponível em: <http://mantis.cs.colorado.edu/ rhan /sensornets.html>. Acesso em: 05 e setembro de 2011.

MARTINS, F.; LOPES, L.; SILVA, M. S.; BARROS, J. *Robust Programming for Sensor Networks.* [S.l.], 2008.

MIYAZAWA, A. H. *Geração Parcial de Código Java a partir de Especificações Formais Z.* Dissertação (Mestrado) — Instituto de Matemática e Estatística – IME, Universidade de Sã Paulo – USP, Brasil, 2008.

MOORE, B.; DEAN, D.; GERBER, A.; WAGENKNECHT, G.; VANDERHEYDEN, P. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.* [S.l.]: IBM, 2004. Acesso em: 02 de setembro de 2011.

MOTTOLA, L. *Programming Wireless Sensor Networks: From Physical to Logical Neighborhoods.* Tese (Doutorado) — Politécnica de Milão, Departamento de Engenharia e Informação, Itália, 2008.

MOTTOLA, L.; PICCO, G. P. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 43, n. 3, p. 19:1–19:51, abr. 2011. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1922649.1922656>.

NAKAMURA, E. F. *Fusão de Dados em Redes de Sensores Sem Fio.* Tese (Doutorado) — Departamento da Ciencia da Computação, Universidade Federal de Minas Gerais - UFMG, Brasil, 2007.

OGUNSHILE, E. K. A. *Automatic Generation of Java Code From Communicating X-Machine Specifications.* Dissertação (Mestrado) — Department of Computer Science, University of Sheffield, Reino Unido, 2005.

OLIVEIRA, H. *Localização no Tempo e no Espaço em Redes de Sensores Sem Fio.* Tese (Doutorado) — Departamento da Ciência da Computação, Universidade Federal de Minas Gerais - UFMG, Brasil, 2008.

RöMER, K.; MATTERN, F. The design space of wireless sensor networks. *Wireless Communications, IEEE*, v. 11, n. 6, p. 54 – 61, dec. 2004. ISSN 1536-1284.

ROSSETO, S. *Integrando Comunicação Assíncrona e Gerência Cooperativa de Tarefas em Ambientes de Computação Distribuída.* Tese (Doutorado) — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro PUC-RIO, Brasil, 2006.

RUIZ, L. B.; CORREIA, L. H.; VIEIRA, L. F.; VIEIRA, M. A.; SERóDIO, C. M.; NAKAMURA, E. F.; MACEDO, D. *Arquiteturas para Redes de Sensores Sem Fio.* [S.l.]: Sociedade Brasileira de Computação, 2004. 167-218 p.

SALLAI, J.; BALOGH, G.; DORA, S. *TinyDT: TinyOS Plug-in for the Eclipse Platform.* 2005. Disponível em: <http://www.tinydt.net (package date)>. Acesso em: 5 de julho de 2011.

SCHMIDT, D. Guest editor's introduction: Model-driven engineering. *Computer*, v. 39, n. 2, p. 25 – 31, feb. 2006. ISSN 0018-9162.

SILVA, R. C. *Teoria da Computação.* 2007. Disponível em: <http://www.dainf.ct.utfpr.edu.br/ fabro/LFA/TeoriaComputacaoNovo.pdf>. Acesso em: 5 de julho de 2011.

STTANETT, M. *The Theory of X-Machines – Part 1.* Reino Unido, 2006.

SUGIHARA, R.; GUPTA, R. K. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, ACM, New York, NY, USA, v. 4, n. 2, p. 8:1–8:29, abr. 2008. ISSN 1550-4859. Disponível em: <http://doi.acm.org/10.1145/1340771.1340774>.

TERFLOTH, K. *Rule-Based Programming Model for Wireless Sensor Networks.* Tese (Doutorado) — Universidade de Berlim, Alemanha, 2009.

VIEIRA, N. J. *Introdução aos Fundamentos da Computação.* [S.l.]: Pioneira Thomson Learning, 2006.

VOELTER, M. *MD* Best Practices.* 2008. Disponível em: <http://www.voelter.de>. Acesso em: 20 de junho 2011.

VOELTER, M.; GROHER, I. Product line implementation using aspect-oriented and model-driven software development. In: *Software Product Line Conference, 2007. SPLC 2007. 11th International.* [S.l.: s.n.], 2007. p. 233 –242.

VOLGYESI, P.; LEDECZI, A. Component-based development of networked embedded applications. In: *Euromicro Conference, 2002. Proceedings. 28th.* [S.l.: s.n.], 2002. p. 68 – 73. ISSN 1089-6503.

WALKINSHAW, N. *Visual X-Machine Description Language (VXMDL).* Dissertação (Mestrado) — Department of Computer Science, University of Sheffield, Reino Unido, 2002.

WANNER, L. F. *Um Ambiente de Suporte à Execução de Aplicações em Rede de Sensores Sem Fio.* Dissertação (Mestrado) — Ciência da Computação, Universidade Federal de Santa Catarina, Brasil, 2006.

APÊNDICE A – Publicações

A.1 Publicação 1

BRAGA, M. L.; SANTOS, A. J. e LUCENA JUNIOR, V. F. (2010). Usando Communicating X-Machine na Construção de Aplicações para Redes de Sensores Sem Fio. In: *Southern Conference on Computational Modeling - MCSUL*, Rio Grande do Sul, 2010.

A.2 Publicação 2

BRAGA, M. L.; SANTOS, A. J. e LUCENA JUNIOR, V. F. (2010b). Modelagem e Geração de Código para Rede de Sensores Sem Fio. In: *Proceedings of the 9th International Information and Telecommunication Technologies Symposium - I2TS 2010*, Rio de Janeiro, 2010.

A.3 Publicação 3

SANTOS, A. J.; BRAGA, M. L. e LUCENA JUNIOR, V. F. Geração Automática de Código para Redes de Sensores Sem Fio Baseado em Componentes de Software. In: *V Congresso Norte-Nordeste de Pesquisa e Inovação - Connepi 2010.*