ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

CML: Uma Linguagem de Especificação de Requisitos Não Funcionais para C

Trabalho de Conclusão de Curso

Engenharia da Computação

Frederico Guilherme Álvares de Oliveira Júnior
Orientador: Prof. Dr. Ricardo Massa Ferreira Lima

Recife, novembro de 2006

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

CML: Uma Linguagem de Especificação de Requisitos Não Funcionais para C

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Frederico Guilherme Álvares de Oliveira Júnior
Orientador: Prof. Dr. Ricardo Massa Ferreira Lima

Recife, novembro de 2006

UNIVERSIDADE
DE PERNAMBUCO

Frederico Guilherme Álvares de Oliveira Júnior

CML: Uma Linguagem de Especificação de Requisitos Não Funcionais para C

Resumo

Especificação formal de programas tem sido uma prática muito comum na engenharia de software, uma vez que um software bem especificado resulta em um produto com menos falhas e ambigüidades. Existem linguagens para especificação de programas que permitem que desenvolvedores especifiquem o comportamento de seus programas. Muitas dessas linguagens fazem uso do paradigma Projeto por Contrato (*Design by Contract*), em que um componente de software garante que o resultado das operações seja correto, se as restrições impostas por ele forem obedecidas. São exemplos desse tipo de linguagem, JML (*Java Modeling Language*), para aplicativos Java, e *DBC for C* (*Design by Contract* para C), para aplicativos C.

Contudo, assegurar a corretude de programas escritos na linguagem de programação C é uma tarefa muito difícil, pois C permite referências a endereços de memória, deixando para o programador, quase sempre, a responsabilidade pela corretude do software. Em contrapartida, os requisitos não funcionais, como tempo de processamento, consumo de energia, tamanho do código fonte, etc., são preocupações típicas de programadores C e, em muitas situações, as principais justificativas para o uso de C como linguagem de desenvolvimento.

Em detrimento disso, este trabalho apresenta a linguagem CML (*C Modeling Language*), que tem como propósito prover um meio de especificar requisitos não funcionais em programas desenvolvidos na linguagem de programação C, utilizando ferramentas externas para verificação desses requisitos.

Abstract

Formal specification has been a common practice in software engineering, therefore a well-specified software results in a product with less failures and ambiguities. There are some specification languages that provide developers a way to specify the behavior of their softwares. Many of these languages use the Design by Contract paradigm, where a software component ensures the right results of operations, if the constraints imposed by it can be met. Examples of such a kind of languages are JML (Java Modeling Language) for Java applications, and DBC for C (Design by Contract for C) for C applications.

However, to ensure correctness of applications developed in the C programming language is a very hard task because C allows memory address references, leaving to programmers, almost always, the responsibility for the software correctness. On the other hand, non-functional requirements, like processing time, power consumption, code size, are typical concerns of C programmers and, in many cases, the main reason for choosing C as development language.

For all these reasons, this work presents the language CML (C Modeling Language), which its purpose is to provide a way to specify non-functional requirements in applications developed in C programming language, using external tools to verify these requirements.

Sumário

Índice de Figuras v

Índice de Tabelas vii

Tabela de Símbolos e Siglas viii

1 Introdução 10

1.1 Motivação 10
1.2 Objetivos 11
1.3 Contribuições 11
1.4 Roteiro 12

2 Trabalhos Relacionados e Fundamentação Teórica 13

2.1 Trabalhos Relacionados 14
2.1.1 Projeto por Contrato 14
2.1.2 JML 16
2.1.3 DBC for C 19
2.2 Ferramentas externas de análise 19
2.2.1 Ferramenta para análise de consumo de energia 20
2.2.2 Ferramenta para análise de restrições temporais 20
2.3 Resumo 27

3 Implementação 29

3.1 Modelagem 29
3.2 Compilador da linguagem CML 32
3.2.1 Processo de Compilação 32
3.2.2 JavaCC 33
3.2.3 Gramática da linguagem CML 34
3.3 Integração com ferramentas de análise 38
3.3.1 Módulos da ferramenta de análise de restrições temporais 39
3.3.2 Ambiente CML 39
3.4 Resumo 41

4 Validação da Proposta 42

4.1 Semáforo 43
4.2 Sistema de monitoramento de veículos 45
4.3 Resumo 50

5 Conclusões e Trabalhos Futuros 51

5.1 Contribuições 51
5.2 Dificuldades e Limitações 52
5.3 Trabalhos Futuros 52

Apêndice A 54

Apêndice B **60**

Bibliografia **63**

Índice de Figuras

Figura 1. Esquema de funcionamento da linguagem CML 14
Figura 2. Utilização de invariantes na linguagem Eiffel. 15
Figura 3. Utilização de pré-condição e pós-condição na linguagem Eiffel. 15
Figura 4. Exemplo de especificação na linguagem JML. 17
Figura 5. Esquema de funcionamento do PCAF. 20
Figura 6. Arquitetura da abordagem proposta por Barreto. 23
Figura 7. Arquitetura do *framework* de geração de código. 24
Figura 8. Versão simplificada do despachante. 25
Figura 9. Exemplo de tabela de escala. 25
Figura 10. Arquitetura proposta para vários processadores. 26
Figura 11. Código do despachante do processador CTC. 26
Figura 12. Tabela de escala para o processador CTC. 27
Figura 13. Código do despachante dos processadores nodos. 27
Figura 14. Tabela de escala de um processador nodo. 27
Figura 15. Exemplo de atribuição em CML. 30
Figura 16. Exemplo de especificação de uma tarefa em CML. 30
Figura 17. Especificação de uma mensagem em CML. 31
Figura 18. Especificação de uma tarefa com relação de precedência e exclusão. 31
Figura 19. Notação EBNF. 32
Figura 20. Diagrama de classes entidades da linguagem CML. 33
Figura 21. Estrutura de um arquivo de especificação JavaCC. 34
Figura 22. Definição do *token* <*START_COMMENT*>. 35
Figura 23. Definição do *token* <*END_COMMENT*>. 35
Figura 24. Expressão regular que ignora qualquer outra coisa. 35
Figura 25. Definição dos *tokens* do estado <*IN_COMMENT_BLOCK*>. 36
Figura 26. Elemento mais abstrato da gramática da linguagem CML. 36
Figura 27. Definição do elemento *statement()*. 37
Figura 28. Definição do elemento *taskType()*. 37
Figura 29. Definição do elemento *taskName()*. 37
Figura 30. Conjunto de elementos que formam o elemento *preceds()*. 38
Figura 31. Integração do ambiente CML com a ferramenta externa de análise. 40
Figura 32. Exemplo de especificação no formato XML. 41
Figura 33. Especificação das tarefas *Verde*, *Amarelo* e *Vermelho*. 43

Figura 34. Especificação do semáforo no formato XML. 44
Figura 35. Resultado positivo das análises das especificações do Semáforo. 44
Figura 36. Diagrama de tempo da escala gerada para o Semáforo. 45
Figura 37. Resultado negativo das análises das especificações do Semáforo. 45
Figura 38. Especificação das tarefas TV_0 e TV_1. 46
Figura 39. Especificação das tarefas *TB0* e *TB0*. 46
Figura 40. Especificação das tarefas *TR0* e *TR1*. 47
Figura 41. Especificação das tarefas *TW0* e *TW1*. 47
Figura 42. Especificação das tarefas *TT0* e *TT1*. 47
Figura 43. Especificação das tarefas TG_0 e TG_1. 48
Figura 44. Especificação das tarefas *TRA* e *M1*. 48
Figura 45. Especificação do Sistema de Monitoramento de Veículo no formato XML. 49
Figura 46. Diagrama de tempo da escala gerada para o Sistema de Monitoramento de Veículos. 49

Índice de Tabelas

Tabela 1. Conjunto de anotações da linguagem CML. 30
Tabela 2. Especificação das Tarefas do Semáforo. 43
Tabela 3. Especificação do Sistema de Controle de Veículos. 46

Tabela de Símbolos e Siglas

JML – Java Modeling Language.
DBC – Design by Contract.
CML – C Modeling Language.
XML – Extensible Markup Language.
UML – Unified Modeling Language.
ESC/Java 2 – Extended Static Checker for Java Version 2.
HTML – Hypertext Markup Language
PCAF – Power Consumption Analysis Framework.
CPN – Coloured Petri Net (Rede de Petri Colorida).
TPN – Timed Petri Net (Rede de Petri Temporizada).
TTU – Task Time Unit.
WCET – Worst Case Execution Time (Tempo de execução do pior caso).
PNML – Petri Net Markup Language.
DOM – Document Object Model.
JDOM – Java Document Object Model.

Agradecimentos

Gostaria de agradecer a minha família, minhas irmãs Ana e Luciana, e especialmente aos meus pais, Evanilde e Frederico (*in memoriam*), pelo apoio na minha educação. Agradeço muito a minha namorada Sandra, por estar sempre ao meu lado nos momentos difíceis.

Gostaria de agradecer a todos os professores do Departamento de Sistemas Computacionais, pela boa formação que me foi dada, em especial o meu orientador, Professor Ricardo Massa, por ter me ajudado muito no desenvolvimento deste trabalho. Agradeço também aos amigos do Centro de Informática, Eduardo Tavares e professor Raimundo Barreto, pela atenção dada durante todo o semestre.

Agradeço as pessoas que contribuíram de alguma forma, na minha formação profissional. Aos amigos Rui Marques, da 7graus.com, Marcelo Pinheiro, da Unidata. Aos amigos da RedeWireless, principalmente Raquel Gondim e Ricardo Albuquerque, pela paciência e compreensão com minhas obrigações acadêmicas.

Por fim, gostaria de agradecer aos meus amigos do curso de Engenharia da Computação da Universidade de Pernambuco. Obrigado pelo companheirismo durante todo o curso.

Muito obrigado a todos.

Capítulo 1

Introdução

A especificação de sistemas de software tem como objetivo descrever com concisão, exatidão e clareza a estrutura e as funcionalidades que o software deverá apresentar [1]. O resultado de um software bem especificado é um produto mais próximo do ideal, além de uma redução substancial de falhas e ambigüidades.

Atualmente, existem linguagens de especificação que utilizam anotações no código fonte para descrever o comportamento de aplicativos e assegurar a sua corretude. Essas linguagens fazem uso da técnica de Projeto por Contrato (*Design by Contract*) [2], em que um componente de software garante que o resultado das operações seja correto, se as condições impostas por ele forem obedecidas. O objetivo dessas linguagens é garantir que o comportamento de um aplicativo (ou um componente de software) esteja correto.

JML (*Java Modeling Language*) [3][4] é uma linguagem de especificação comportamental de aplicativos escritos na linguagem de programação Java. Esta linguagem possui uma gama de ferramentas que auxiliam o desenvolvedor/projetista de software a especificar sua aplicação. Dentre as ferramentas disponíveis estão: um gerador automático de documentação, um gerador de classes Java para testes unitários, além de verificadores de código muito poderosos.

DBC for C (*Design by Contract* para C) [5] é uma ferramenta que utiliza o mesmo princípio da linguagem JML, porém voltada para a linguagem de programação C. Entretanto, diferentemente de JML, *DBC for C*, é pouco documentada e não há manutenção por parte da comunidade e de seus criadores.

1.1 Motivação

Dispositivos com baixa capacidade de recursos requerem que seus aplicativos atendam a restrições que, muitas vezes, não são exigidas por computadores com maior poder de processamento. Nesses casos, a linguagem C é muito utilizada, pois possui compiladores que geram códigos com bom desempenho. Além disso, C provê ao desenvolvedor maior flexibilidade na manipulação dos dados e na maneira como os mesmos são armazenados na memória.

Assegurar a corretude de programas escritos na linguagem de programação C é uma tarefa muito difícil, uma vez que C permite referências a endereços de memória, que são fontes de efeitos colaterais e dificultam a realização de análises estáticas sobre a corretude do programa.

De fato, a própria filosofia utilizada no projeto da linguagem C coloca nas mãos do programador a responsabilidade sobre o desenvolvimento de programa corretos. É incomum utilizar mecanismos de prova de corretude de programas escritos nesta linguagem. Em contrapartida, os requisitos não funcionais, como tempo de processamento, consumo de energia, tamanho de código e etc., são preocupações típicas de programadores C e, em muitas situações, as principais justificativas para o uso de C como linguagem de desenvolvimento.

Neste contexto, assim como JML é empregado para especificar requisitos funcionais de programas Java, seria apropriado disponibilizar uma linguagem de especificação de requisitos não funcionais para a linguagem C.

1.2 Objetivos

Este trabalho tem como principal objetivo propor uma linguagem para especificação de requisitos não funcionais de aplicativos escritos na linguagem C.

Os objetivos específicos do trabalho são:

- propor uma linguagem para especificação de requisitos não funcionais com enfoque nas restrições temporais em sistemas de tempo real;

- construir um ambiente para integração da linguagem a ferramentas que automatizam a análise dos requisitos não funcionais especificados.

1.3 Contribuições

Este trabalho apresenta uma linguagem para especificação de requisitos não funcionais de aplicações escritas na linguagem C. Pela limitação do escopo, a linguagem proposta neste trabalho suporta apenas especificação requisitos relativos a restrições temporais de sistemas de tempo real crítico. As contribuições são descritas abaixo:

- **Definição e modelagem da linguagem:** a linguagem CML (*C Modeling Language*) consiste em anotações referentes ao aplicativo que está sendo especificado. Essas anotações são colocadas dentro de blocos de comentários, de forma que não haja interferência na compilação do aplicativo;

- **Construção do compilador:** foi desenvolvido um compilador que traduz as especificações para uma representação intermediária. Essa representação é importante para que ferramentas externas possam utilizar a especificação com mais facilidade. Na versão atual, essa representação foi definida com uma estrutura de objetos Java. Tal representação é então traduzida para o formato de entrada da ferramenta desejada;

- **Interação com ferramentas externas:** para que CML seja utilizada de forma prática, procurou-se integrá-la com ferramentas externas de análise das especificações. Barreto [6] propôs uma ferramenta que verifica restrições temporais em programas escritos na linguagem de programação C. Essa ferramenta recebe como entrada um arquivo de especificação no formato XML, que é gerado a partir da representação intermediária (estrutura de objetos Java) criada pelo compilador da linguagem CML.

1.4 Roteiro

O Capítulo 2 aborda alguns conceitos importantes para o entendimento deste trabalho como, linguagens de especificação, Projeto por Contrato, ferramentas de análises de requisitos não funcionais, além de alguns trabalhos relacionados como a linguagens JML e *DBC for C*.

O Capítulo 3 detalha a definição e implementação da linguagem. A implementação do compilador e a integração com uma ferramenta externa de análise de requisitos são descritas.

O Capítulo 4 consiste na validação da proposta deste trabalho, por meio da especificação de dois sistemas, em que o leitor pode ter uma noção prática do funcionamento da linguagem CML. O primeiro sistema é de um Semáforo de Trânsito e o segundo é de um Sistema de Monitoramento de Veículos. Esses estudos de caso validam a forma como as especificações são traduzidas para um formato compreendido pelas ferramentas externa de análise.

Por fim, o Capítulo 5 é reservado às considerações finais e aos trabalhos futuros.

Capítulo 2

Trabalhos Relacionados e Fundamentação Teórica

Atualmente, existem linguagens de especificação que utilizam anotações no código fonte para especificar o comportamento de aplicativos e assegurar a sua corretude. Essas linguagens geralmente utilizam a técnica de Projeto por Contrato, em que um componente de software garante que o resultado das operações seja correto se as condições impostas por ele forem obedecidas. O objetivo dessas linguagens é garantir que o comportamento de um aplicativo (ou um componente de software) esteja correto. Um exemplo disso é JML [3], que é uma linguagem de especificação comportamental de aplicativos escritos com a linguagem Java. A exemplo de JML, a ferramenta *DBC for C* [5] também provê ao usuário uma forma de especificação comportamental de programas desenvolvidos na linguagem C. Contudo, a falta de documentação e manutenção dessa ferramenta impossibilita o seu estudo e a sua evolução.

A linguagem C é bastante utilizada para desenvolvimento de software em dispositivos de baixa capacidade de recursos, pois possui compiladores que geram códigos com bom desempenho. Além disso, C provê ao desenvolvedor maior flexibilidade na manipulação dos dados e na maneira como os mesmos são armazenados na memória. Portanto, boa parte dos desenvolvedores que utilizam C tem grande preocupação, não apenas em garantir que o programa se comporte de maneira correta, mas também em garantir que aspectos não funcionais, como tempo de processamento e consumo de energia, sejam tratados como requisitos do software, neste caso, requisitos não funcionais.

Um dos grandes desafios da engenharia de software é o levantamento de requisitos não funcionais, pois na maioria das vezes, eles são relacionados a propriedades do hardware no qual o software será executado. Essa dificuldade pode ocasionar em requisitos mal especificados, ou na impossibilidade de assegurar o cumprimento deles. Em detrimento disso, este trabalho tem como objetivo propor uma linguagem de especificação de requisitos não funcionais. A exemplo das linguagens de especificação Eiffel [2], JML e *DBC for C*, a linguagem CML (*C Modeling Language*) se utiliza de anotações em código fonte para especificar requisitos de software.

Este trabalho se concentra na definição da linguagem CML, assim como no desenvolvimento de um compilador para interpretá-la, ou seja, identificar as anotações inseridas em códigos fonte escritos na linguagem C, e transformá-as em parâmetros de entrada para as ferramentas externas de análise. Por exemplo, a ferramenta proposta por Barreto [6], que

preocupa-se com o requisito de restrições temporais em sistemas de tempo real. A linguagem proposta neste trabalho está limitada a este escopo. Entretanto, é possível que este trabalho seja estendido posteriormente para que a linguagem CML passe a considerar uma gama maior de requisitos não funcionais, como tamanho de código, consumo de energia, dentre outros. A Figura 1 ilustra um esquema de funcionamento da linguagem CML e como os módulos envolvidos devem interagir.

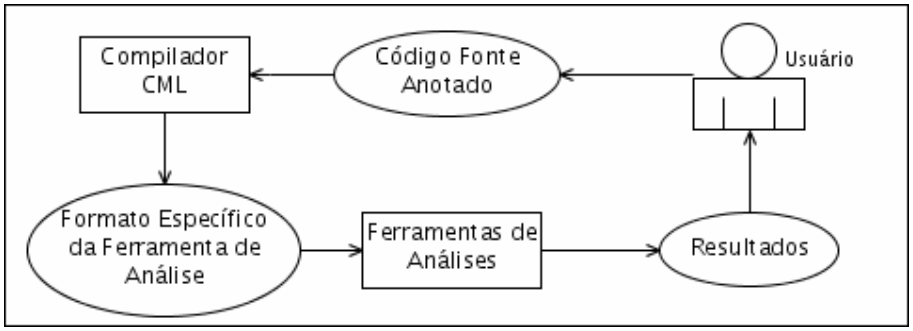

Figura 1. Esquema de funcionamento da linguagem CML.

Resumidamente, a linguagem CML funciona da seguinte forma. Um código fonte na linguagem C com especificações anotadas é dado como entrada para o compilador da linguagem CML, este, por sua vez, identifica as anotações no código e as envia para as ferramentas que dão suporte às análises. Os resultados obtidos por essas ferramentas são exibidos, explicitando se o programa está dentro ou não do que foi especificado.

Para cada requisito, é necessário que a linguagem CML saiba se comunicar com a ferramenta de análise correspondente. Como já foi dito, a linguagem suporta atualmente apenas um tipo de requisito, e portanto, utiliza apenas a ferramenta proposta por Barreto [6]. Essa ferramenta recebe como entrada uma especificação composta por características de uma ou várias tarefas. Essa especificação é inserida no próprio código fonte, utilizando as anotações. A ferramenta de análise, por sua vez, realiza algumas análises utilizando redes de Petri [7], que é um formalismo matemático utilizado para especificação e verificação de sistemas. Como resultado, é retornada uma escala possível para as restrições impostas pelo desenvolvedor. Caso não haja nenhuma escala possível, fica caracterizado que os requisitos especificados não foram atendidos.

Neste capítulo serão abordados com mais detalhes alguns trabalhos relacionados, como a técnica de Projeto por Contrato, as linguagem de especificação JML e a ferramenta *DBC for C*. Em seguida, serão detalhadas as ferramentas que dão suporte a linguagem CML, ou seja, as ferramentas que realizam análises de aspectos não funcionais de programas desenvolvidos na linguagem C.

2.1 Trabalhos Relacionados

2.1.1 Projeto por Contrato

Projeto por Contrato (*Design by Contract*) [2] é uma técnica para desenvolvimento de software e sua principal característica é tratar a relação entre a classe e seus clientes como um contrato formal, explicitando os direitos e obrigações de cada parte. Um contrato é dado quando um

cliente tem a obrigação de cumprir alguns requisitos antes de usufruir da classe, e a classe, por sua vez, assegurar que o resultado está de acordo com o que foi especificado no contrato, garantindo a corretude do código especificado. Isso pode ser aplicado a uma chamada de um método de uma classe, por exemplo.

Uma das pioneiras na utilização da técnica de Projeto por Contrato foi a linguagem orientada a objetos Eiffel [8]. Para especificar um componente de software, a linguagem Eiffel utiliza algumas assertivas[1] que controlam as condições de uso desse componente. As assertivas mais comuns da linguagem Eiffel são invariantes, pré-condições e pós-condições. Invariantes são restrições que são válidas para toda a classe, e não apenas para um método. A Figura 2 ilustra um exemplo de utilização de invariantes na linguagem Eiffel. Nesse exemplo, a classe representa um contador que é sempre não negativo. A palavra-chave *invariant* especifica que os valores possíveis para o atributo *item* têm que ser um número não-negativo.

```
class CONTADOR
  ...
feature
  ...
invariant
       item >=0
end
```

Figura 2. Utilização de invariantes na linguagem Eiffel.

Uma pré-condição indica o que deve ser feito para o funcionamento correto do componente que está sendo especificado. A Figura 3 ilustra um exemplo de especificação de um método na linguagem Eiffel, utilizando uma pré-condição e uma pós-condição. A palavra-chave *require* indica o início de uma pré-condição, nesse caso, ela define que o argumento passado para o método *decrementar* tem que ser, obrigatoriamente, maior que zero.

```
decrementar is
     require
            item > 0
     do
            item := item -1
     ensure
            item= old item -1
end
```

Figura 3. Utilização de pré-condição e pós-condição na linguagem Eiffel.

Uma pós-condição, por sua vez, é a garantia dada pela outra parte do contrato, ou seja, os direitos que são assegurados para quem utiliza o componente de software que está sendo especificado. Na Figura 3, a palavra-chave *ensure* indica o início de uma pós-condição na linguagem Eiffel. Essa pós-condição especifica que, ao final do método, a variável *item* será decrementada de um.

Aqui foi mostrada apenas uma parte do grande número de asserções disponíveis da linguagem Eiffel. O intuito foi exemplificar a utilização da técnica de Projeto por Contrato. É possível também, especificar casos excepcionais, porém, esses casos estão detalhados na próxima seção.

---

[1] Recurso da linguagem para testar as suposições feitas pelo programador dentro do código.

2.1.2 JML

Um diagrama de classes UML (*Unified Modeling Language*)[9] pode ser anotado para incluir restrições adicionais ao modelo. Para isso, existe a linguagem OCL (*Object Constraint Language*) [9], que utiliza elementos de lógica de predicados e teoria dos conjuntos de forma textual. Entretanto, a linguagem OCL não possui muitas ferramentas para suporte para verificações das restrições estabelecidas.

Assim como OCL, JML (*Java Modeling Language*) [3] é uma linguagem de especificação formal baseada na técnica de Projeto por Contrato, pode ser utilizada para especificar comportamentos de módulos e interfaces de aplicativos desenvolvidos na linguagem Java. Além de possuir uma sintaxe muito semelhante à linguagem Java, existem várias ferramentas que auxiliam a utilização da linguagem para tarefas como verificação estática, compilação, geração de casos de testes para a ferramenta JUnit [10], geração de documentação Javadoc [11], entre outros.

As especificações JML são inseridas no próprio código em forma de anotações (*annotations*), ou seja, não há a necessidade de armazenar as especificações em arquivos de metadados, geralmente apresentados no formato XML. Essa abordagem já é muito utilizada dentro de aplicativos orientados a objetos, em que as classes de objetos são anotadas com intuito de serem mapeadas em objetos (tabelas e atributos) de banco de dados relacional. Isso permite uma proximidade maior entre a especificação e o código em que se está trabalhando, proporcionando a escrita de um código de melhor qualidade.

A linguagem JML possui um grande número de anotações, quase todas baseadas no Projeto por Contrato da linguagem Eiffel, para especificação de métodos e classes da linguagem Java. Como dito anteriormente, as especificações JML são inseridas em forma de anotações, ou seja, dentro de comentários, como //@ ou /*@*/. O símbolo @ indica o início de linha e, portanto, são ignorados.

Exemplos de Asserções

Assim como na linguagem Eiffel, JML provê uma grande variedade de asserções para serem utilizadas em aplicativos Java. A Figura 4 ilustra um exemplo de uma classe especificada com auxílio das asserções invariante, pré-condição, pós-condição e exceções, na linguagem JML.

```
public class Pessoa {

        private String nome;
        private int peso;

        /*@ public invariant !nome.equals("")
          @ && peso >= 0;
          @*/
        public int getPeso() {
             return peso;
        }

        /*@ also
          @ requires kgs >= 0;
          @ requires peso + kgs >= 0;
          @ ensures peso == \old(peso + kgs);
          @ signals (Exception) peso != \old(peso+kgs);
          @*/
        public void addKgs(int kgs) {
          peso += kgs;
        }

        public Pessoa(String n) {
             nome = n; peso = 0;
        }
}
```

Figura 4. Exemplo de especificação na linguagem JML.

- **Invariantes:** a asserção *invariant* representa um invariante da classe, ou seja, uma restrição comum da classe, nesse caso o invariante é formado sob a condição de o atributo *nome* ser diferente de vazio e o atributo *peso* ser maior ou igual a zero.

```
/*@ public invariant !nome.equals("")
  @ && peso >= 0;
  @*/
```

- **Pós-condições e Pré-condições:** a assertiva *requires* representa a pré-condição do contrato, onde o argumento *kgs* obrigatoriamente tem que ser maior ou igual à zero. Já a assertiva *ensures* representa a pós-condição do contrato, onde é assegurado que o atributo peso será igual ao seu valor antigo (representado pela palavra-chave *old*) somado ao argumento *kgs*. A palavra-chave *also*, indica que o método herda as especificações da sua classe pai.

```
/*@ also
  @ requires kgs >= 0;
  @ requires peso + kgs >= 0;
  @ ensures peso == \old(peso + kgs);
  @*/
```

- **Exceção:** a assertiva *signals* representa um caso excepcional do contrato, ou seja, uma exceção que será lançará caso o valor do atributo *peso*, após a execuçao do método, seja diferente da soma do seu valor antigo com o valor do argumento *kgs*.

```
/*@signals (Exception) kgs < 0
```

Ferramentas

Uma das grandes vantagens da utilização de JML é a existência de várias ferramentas de suporte. Abaixo serão abordadas as principais delas:

- **Compilador JML (jmlc):** para que as anotações sejam reconhecidas como instruções, é necessário um compilador especial, que reconheça essas anotações, inseridas em blocos de comentários, pois tais anotações são ignoradas pelo compilador Java, que as consideram simples comentários.

- **Interpretador JML/Java (jmlrac): é** o responsável por verificar se o código fonte que se deseja especificar atende às restrições impostas pelas especificações. Essa ferramenta é capaz de fornecer informações dinâmicas e estáticas. Um exemplo de informação estática seria o posicionamento de um possível erro encontrado. E de informação dinâmica seria valores de variáveis. O *jmlrac* não é capaz de verificar todos os tipos de violação de regras, uma vez que a verificação é dinâmica e baseada nas contraprovas, é possível que ela não realize todas as contraprovas necessárias, deixando de verificar algumas violações.

- **ESC/Java 2 [12]:** é uma evolução do ESC/Java, desenvolvida pela equipe do Compaq Research Center [13]. Essa ferramenta promove uma verificação estática de códigos escritos na linguagem Java. A verificação é dita estática porque é realizada diretamente no código-fonte e não em tempo de execução, como é no caso do *jmlrac*. A verificação também é dita estendida porque tenta encontrar erros que, geralmente, não são verificados pelos compiladores usuais (compilador Java). Por exemplo, referências a variáveis nulas, acesso a índices de *arrays* fora dos limites, dentre outros. É notável a importância de um verificador estático, uma vez que não há a necessidade da execução do código fonte para realizar as verificações.

- **Ferramenta de teste unitário:** JUnit[10] é uma poderosa ferramenta para realização de testes unitários de programas desenvolvidos na linguagem Java. Existem duas ferramentas para suporte de JML para testes unitários. Uma delas é a *jmlunit*, que gera automaticamente classes de teste da ferramenta JUnit, a partir das anotações JML. Outra ferramenta é *jml-junit*, que auxilia na comunicação com a ferramenta JUnit, executando a classe recém-gerada pela ferramenta *jmlunit*. É possível observar que, com a quantidade de ferramentas existentes para suporte a JML, é possível realizar verificações não apenas utilizando ferramentas de verificação convencionais do JML, como ***jmlrar*** e ESC/Java2, mas também, com ferramentas de testes.

- **Gerador de documentação HTML (jmldoc):** documentos Javadoc são bastante conhecidos da comunidade de desenvolvedores Java. Com auxílio de uma ferramenta, é possível criar documentos, em formato HTML, que servem como documentação de código fonte, como dependência entre classes e todas as informações sobre classes, atributos e métodos. Essa documentação é bastante útil quando se tem um grupo de pessoas trabalhando em um mesmo projeto e compartilhando um mesmo código, além de facilitar o entendimento do código, inclusive para novos integrantes de uma equipe. A linguagem JML também tem suporte a esse tipo de documentação através da ferramenta

jmldoc. Essa ferramenta é responsável por gerar, a partir das especificações JML, arquivos HTML contendo a documentação das especificações.

2.1.3 DBC for C

Assim como JML, *DBC for C* (*Design by Contract* para C) [5] também é uma linguagem para especificações comportamentais, baseadas na técnica de Projeto por Contrato, em aplicativos desenvolvidos na linguagem C. Tanto o compilador, quando os verificadores são implementados na linguagem de programação Ruby [14]. As assertivas utilizadas por DBC for C seguem o mesmo padrão, como sugere o trecho de código abaixo. Nesse caso, cada variável que utiliza a estrutura de dados *DArray_T* tem que atender as especificações colocadas como invariantes do sistema.

```
/**
  context DArray_T
  inv: self != NULL
  inv: self->length >= 0
  inv: self->capacity >= self->length
  inv: (self->capacity == 0 and self->ptr == NULL)
    or (self->capacity > 0 and self->ptr != NULL)
 */
struct DArray_T {
  void  **ptr;
  long    length;
  long    capacity;
};
```

No caso abaixo, são utilizadas as assertivas de pré e pos condições, especificando restrições para evitar que um array seja acessado de forma errada.

```
/**
  pre: index >= 0
  pre: index < ary->length
  post: return == ary->ptr[index]
 */
void *DArray_get(DArray_T ary, long index)
{
  return ary->ptr[index];
}
```

Diferentemente de JML, não existem muitas ferramentas que auxiliem na utilização do *DBC for C*, além de possuir uma documentação bastante escassa, o que dificulta o estudo da ferramenta, além de impossibilitar a sua evolução com ajuda da comunidade.

2.2 Ferramentas externas de análise

Para que a linguagem proposta neste trabalho tenha utilidade prática, é necessária a utilização de ferramentas para realizar as análises das restrições impostas. Essas ferramentas desempenham um papel similar ao dos verificadores existentes na linguagem JML. A verificação nesse caso consiste em enviar parâmetros obtidos nas anotações no código fonte para ferramentas externas que realizam análises de aspectos não funcionais, tais como consumo de energia e restrições temporais.

2.2.1 Ferramenta para análise de consumo de energia

Uma das grandes preocupações no desenvolvimento de software para sistemas embarcados, ou de pouco poder de processamento, é o consumo de energia. Um dos fatores mais importantes dessa preocupação é o interesse de empresas fabricantes de dispositivos móveis, pois menor consumo de energia implica uma maior autonomia de seus dispositivos.

O PCAF (*Power Consumption Analysis Framework*) [15] é uma ferramenta muito poderosa para análise de consumo de energia em programas desenvolvidos em C. O processo de análise se dá a partir do código de máquina, obtido através da compilação do código fonte escrito na linguagem C. Esse código é traduzido em um modelo de rede de Petri Colorida[7], e enviado para a ferramenta de análise CPN-Tools [16].

A Figura 5 ilustra o esquema de funcionamento do PCAF. Primeiramente, tem-se o código fonte na linguagem C. Esse código é compilado em um código de máquina, que por sua vez, servirá como entrada da ferramenta PCAF. Um dos módulos que compõem o PCAF é um compilador que transforma o binário (código fonte compilado) no modelo de redes de Petri da ferramenta CPN-Tools. Por último, a ferramenta CPN-Tools é utilizada para realizar análises que determinam o consumo de energia de cada instrução do código.

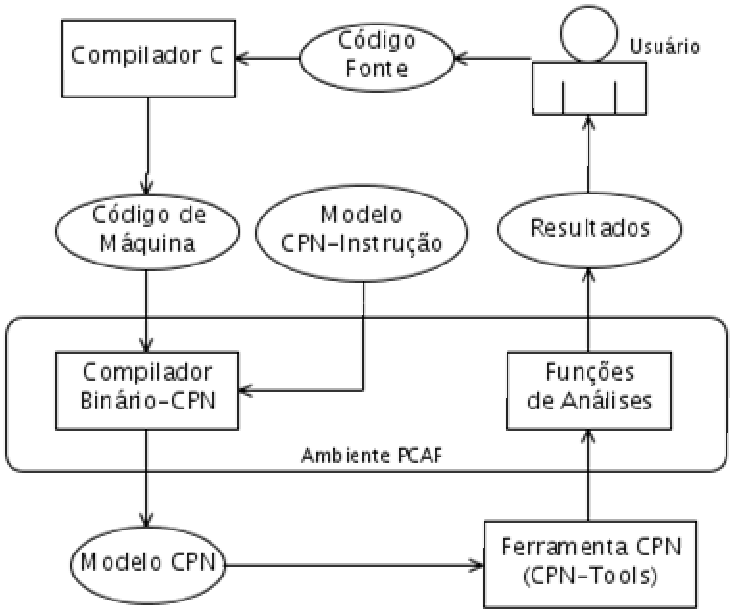

Figura 5. Esquema de funcionamento do PCAF.

2.2.2 Ferramenta para análise de restrições temporais

Sistemas de tempo real é uma classificação de sistemas que tem como característica não apenas a correta execução lógica das tarefas, mas também o limite de tempo em que elas forem executas. Portanto, em um sistema de tempo real, uma tarefa é bem sucedida se ela for concluída com sucesso e dentro de um tempo pré-determinado [17].

Os sistemas de tempo real podem ser subdivididos em *Soft Real Time* (apenas tempo real) e *Hard Real Time* (tempo real crítico). No primeiro, o não cumprimento das restrições temporais não acarreta maiores danos ou desastres, já nos sistemas de tempo real críticos, se as restrições temporais não forem atendidas, as conseqüências podem ser desastrosas, podendo causar um

prejuízo muito grande ou até mesmo perda de vidas humanas. Para que essas restrições temporais sejam atendidas, às vezes é necessário mais de um processador para dividir a carga de processamento das tarefas.

Nos sistema de tempo real, o escalonamento é muito importante, tendo em vista que as tarefas têm que ser bem escalonadas de forma que as mesmas possam atender às restrições temporais, proporcionando o funcionamento correto do sistema. De uma forma geral, existem dois tipos de escalonamento de tarefas: *runtime* (dinâmico) e o *pre-runtime* (estático). No escalonamento *runtime*, a decisão sobre qual tarefa a ser executada é feita em tempo de execução. Já no escalonamento *pré-runtime* essa decisão é tomada em tempo de compilação. O método de escalonamento também pode ser classificado com o preemptivo e não-preemptivo. No primeiro caso, uma tarefa com maior prioridade pode interromper a execução de uma outra de menor prioridade, ao contrário do que acontece no escalonamento não-preemptivo.

Em seu trabalho, Barreto [6] propôs uma ferramenta para geração automática de código para sistemas embarcados de tempo real críticos, a partir de um conjunto de especificações. Esse trabalho foi estendido por Tavares [18], permitindo a utilização de vários processadores, o que é ideal para sistemas embarcados de tempo real críticos.

Nessa abordagem, uma especificação é composta por um conjunto de tarefas, e a cada tarefa são atribuídos: as restrições temporais, as relações entre processadores, o método de escalonamento, assim como o processador no qual a tarefa será executada. Além disso, existem tarefas especiais para realizar a comunicação entre os processadores, elas são chamadas de tarefas de comunicação. Essas tarefas são modeladas utilizando redes de Petri temporizadas, que permitem modelagem de sistemas levando em consideração o tempo que determinada ação necessita para ser executada, utilizando notações de tempo [7]. O tipo de escalonamento adotado é o *pré-runtime*, já que essa política é mais previsível que a *runtime*, e conseqüentemente mais aconselhável para sistemas de tempo real. Uma possível escala (uma em que todas as restrições forem atendidas) é encontrada, e a partir dela, um código é gerado.

A Figura 6 ilustra um diagrama com mais detalhes sobre as fases que compõem o método proposto:

- **Especificação:** o modelo de especificação consiste em um conjunto de tarefas (*tasks*), cada uma com uma série de propriedades: restrições de tempo (fase, tempo de início, pior tempo de execução, *deadline* e período), método de escalonamento (preemptivo ou não-preemptivo), relações entre tarefas (precedência e exclusão), processador onde a tarefa está alocada e código fonte. No caso de se utilizar mais de um processador, a comunicação entre eles é tratada como uma tarefa especial, chamada de tarefa de comunicação, que é composta pelo tempo de comunicação do pior, canal de comunicação, transmissor e receptor.

- **Modelagem:** esta fase consiste em traduzir as especificações em um modelo de rede de Petri Temporizada. Essa estrutura é montada através de composição de blocos, em que cada bloco consiste em uma parte da estrutura da especificação. O resultado dessa modelagem é utilizado como entrada para a fase de geração de escala. Mais detalhes sobre a modelagem em rede de Petri podem ser obtidos em [6] [18].

- **Geração da escala:** o método utilizado para o escalonamento é o *pré-runtime*, uma vez que o trabalho se concentra em sistemas de tempo real. Portanto, a escala é gerada toda estaticamente, ou seja, antes da execução. Mais detalhes sobre o algoritmo para geração de escala podem ser obtidos em [6] [18].

- **Geração de Código:** nesta fase, o código da escala é gerado, baseado na escala que foi computada previamente. Nesse trabalho, Barreto [6] propôs um *framework* para geração de código considerando os itens de especificação descritos anteriormente.

Em seguida serão detalhadas as etapas de especificação e de geração de código. A especificação é de extrema importância para linguagem CML, pois trata diretamente dos atributos utilizados por ela para especificar um sistema de tempo real. A geração de código também é muito importante para se ter uma noção melhor da arquitetura de codificação, que é a mesma adotada nos sistemas utilizados no Capítulo 4 para validar a proposta.

Especificação de um sistema de tempo real

Um sistema de tempo real, segundo Barreto [6], pode ser especificado de acordo com os seguintes itens:

- **Restrições temporais:** a especificação de uma tarefa é composta por um conjunto de atributos, como fase (*phase*), período (*period*), tempo de início (*release*), tempo de execução do pior caso (WCET), tempo máximo de execução de uma tarefa (*deadline*) e o processador onde cada tarefa está alocada. Pode-se entender como fase o intervalo de tempo da primeira requisição da tarefa depois do sistema ter iniciado. Compreende-se como período a periodicidade com que a tarefa é executada. O *release*, WCET e *deadline* são instantes de tempo relativos ao início de um período. Devido à utilização de uma arquitetura com múltiplos processadores, torna-se necessário uma alocação prévia das tarefas nos processadores. No escalonamento *pre-runtime*, as tarefas são escalonadas levando em consideração o período da escala (P_S) que corresponde ao MMC (mínimo múltiplo comum) entre os períodos de todas as tarefas. Com esse valor, é possível saber o número de instâncias da mesma tarefa τ_i, com período p_i, por $N(\tau_i) = P_S/p_i$.

- **Relações entre tarefas:** as relações são classificadas em relações de precedência e exclusão. Quando uma tarefa *T1* precede uma tarefa *T2* significa que *T2* só pode iniciar a sua execução quando *T1* for finalizada. Esse tipo de relação é bastante útil quando uma tarefa necessita de alguma informação produzida por outra. Quando uma tarefa *T1* tem uma relação de exclusão com uma tarefa *T2* significa que *T2* não pode iniciar sua execução enquanto *T1* estiver executando, ou seja, a *T2* não pode interromper a execução de *T1*.

- **Método de escalonamento:** os métodos de escalonamento podem ser preemptivo ou não-preemptivo. No escalonamento preemptivo a execução de uma tarefa *T1* pode ser interrompida por uma outra tarefa *T2*. No escalonamento não-preemptivo, a tarefa é executada sem interferência de nenhuma outra tarefa, até que a mesma seja concluída.

- **Comunicação entre processadores:** a comunicação entre processadores é realizada através de uma tarefa especial chamada tarefa de comunicação. Uma tarefa de comunicação é composta por uma tarefa de envio, uma tarefa de recebimento, o tempo de comunicação do pior caso e o canal de comunicação (*bus*) entre as tarefas alocadas em diferentes processadores.

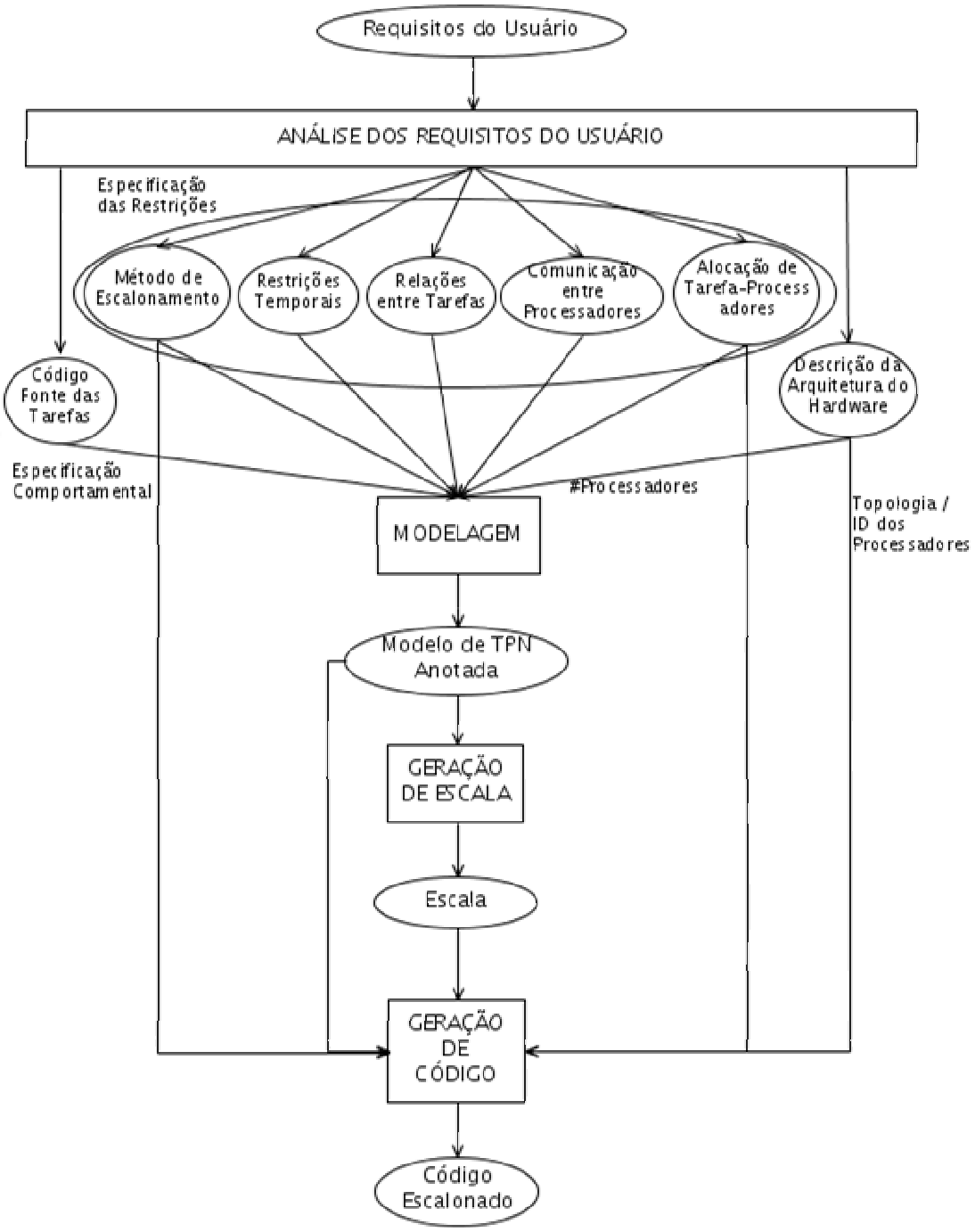

Figura 6. Arquitetura da abordagem proposta por Barreto [6] .

Geração de código

O *framework* utilizado pela geração de código não contempla apenas a geração de código das tarefas, mas também inclui um manipulador de interrupções e um simples despachante. O

despachante é utilizado para desempenhar tarefas como programação do *clock* (relógio), armazenamento de contexto, recuperação de contexto, e chamada de tarefas para execução. O manipulador de interrupções sempre sede o controle ao despachante, que por sua vez avalia se há necessidade de armazenamento ou recuperação de contexto, antes de executar a tarefa.

De acordo com a Figura 7, o funcionamento do *framework* de geração de código pode ser resumido como segue:

1. Considerando que o sistema seja iniciado com o valor do *clock* igual a zero. O manipulador de interrupções é obrigatoriamente chamado e cede o controle para o núcleo do despachante, que por sua vez utiliza o *clock* atual e verifica se existe alguma tarefa para ser executada (linha 4, Figura 8).
2. Para saber quando e qual tarefa deve ser executada, o despachante utiliza da tabela de escala, representada por um vetor de *item_escala*. Esse vetor é acessado como uma lista circular (linha 13, Figura 8).
3. O núcleo do despachante armazena o contexto da tarefa atual, se ela estiver sendo preemptada por outra tarefa (linha 6, Figura 8).
4. O núcleo do despachante acessa a memória externa para armazenar esse contexto.
5. O núcleo do despachante restaura o contexto de uma nova tarefa (linha 9, Figura 8).

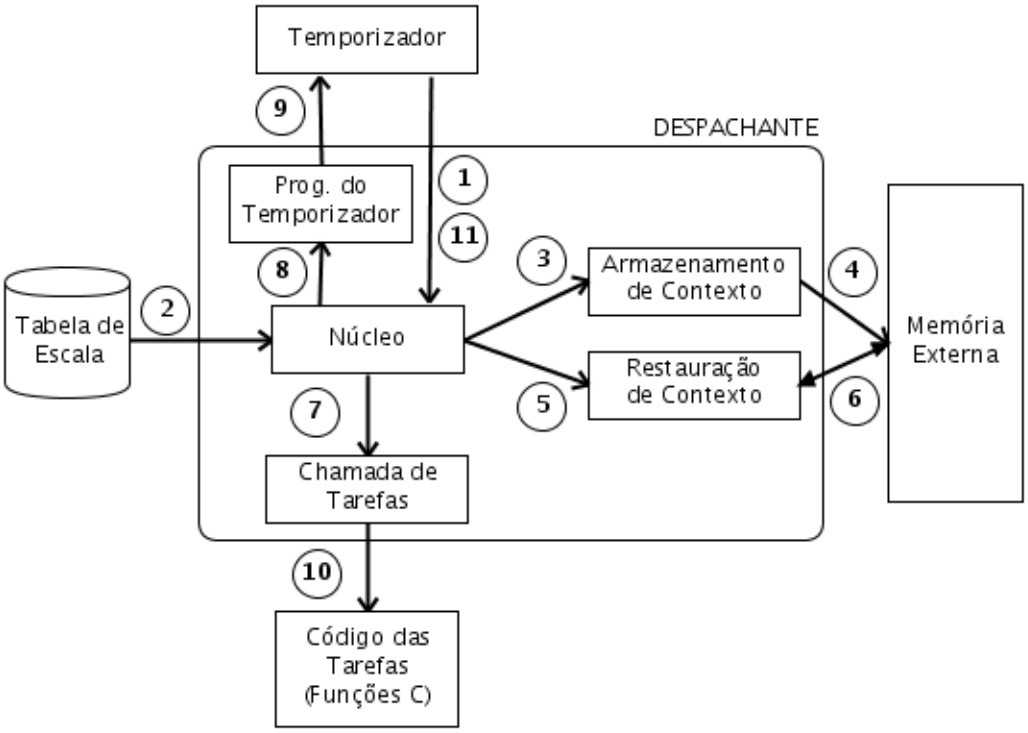

Figura 7. Arquitetura do *framework* de geração de código [6].

6. O núcleo do despachante acessa a memória externa para obter esse contexto.
7. Baseado na tabela de escala, o núcleo do despachante atribui a uma variável, a função que representa a próxima tarefa a ser executada (linha 12, Figura 8).
8. O núcleo do despachante programa o temporizador para interromper no início da execução da próxima tarefa (linha 14, Figura 8).
9. O temporizador é ativado (linha 15, Figura 8).

10. A função que representa o código das tarefas.
11. Quando há uma interrupção, o controle é cedido ao despachante novamente.

A Figura 8 ilustra uma versão simplificada do despachante. Existem algumas variáveis globais como *indice_tabela*, que identifica o índice da tabela de escala atual, *nova_tarefa*, que representa a tarefa que está sendo executada, *clock_atual*, que representa o valor do *clock*, e *TAMANHO_TABELA*, que indica o número de entradas da tabela de escalas, dentre outras.

```
1   void despachante()
2   {
3     struct item_escala nova_tarefa = tabela_escala[indice_tabela];
4     clock_atual = nova_tarefa.clock;
5     if(existeTarefaEmExecucao) {
6        // armazernamento de contexto
7     }
8     if(nova_tarefa.ehRetornoPreempcao) {
9        // recuparação de contexto
10    }
11
12    funcao_tarefa = nova_tarefa.ponteiro;
13    indice_tabela = ((++indice_tabela) % TAMANHO_TABELA);
14    programarTemporizador(tabela_escala[indice_tabela].clock);
15    ativarTemporizador();
16  }
```

Figura 8. Versão simplificada do despachante.

A tabela de escala é representada como um vetor de *item_escala*. Cada entrada do vetor representa uma parte a ser executada da instância de determinada tarefa. Essa estrutura representa: (i) o tempo de início da execução; (ii) um sinalizador de retorno preempção; (iii) o identificador da tarefa; (iv) ponteiro para função que representa o código da tarefa. A Figura 9 ilustra um exemplo de tabela de escala com 7 instâncias de tarefas.

```
#define TAMANHO_TABELA 7

struct item_escala
tabela_escala[TAMANHO_TABELA] =
{
   { 1, false, 1, (int *)tarefaA},
   { 4, false, 2, (int *)tarefaB},
   { 6, false, 3, (int *)tarefaC},
   { 8, true, 2, (int *)tarefaB},
   {10, false, 4, (int *)tarefaD},
   {11, true, 2, (int *)tarefaB},
   {13, true, 1, (int *)tarefaA},
   {18, false, 1, (int *)tarefaA},
   {20, false, 3, (int *)tarefaC},
   {22, false, 2, (int *)tarefaB},
   {28, true, 1, (int *)tarefaA}
};
```

Figura 9. Exemplo de tabela de escala.

A arquitetura proposta por Tavares [18], como extensão ao trabalho de Barreto [6], adota o protocolo Sincronização Mestre Central (*Central Master Synchronization*)[19], em que um processador mestre desempenha a contagem de tempo e envia mensagens periódicas de sincronização aos processadores escravos. Nesse contexto, o processador mestre não executa nenhuma tarefa, mas sim um despachante especial. Em contrapartida, os processadores escravos não possuem seus próprios *clocks* de tempo real. O único *clock* de tempo real fica situado no processador mestre. Esse processador é também chamado processador CTC (*Central Time Counting*) e os escravos de processadores nodos. A Figura 10 ilustra a arquitetura proposta para vários processadores.

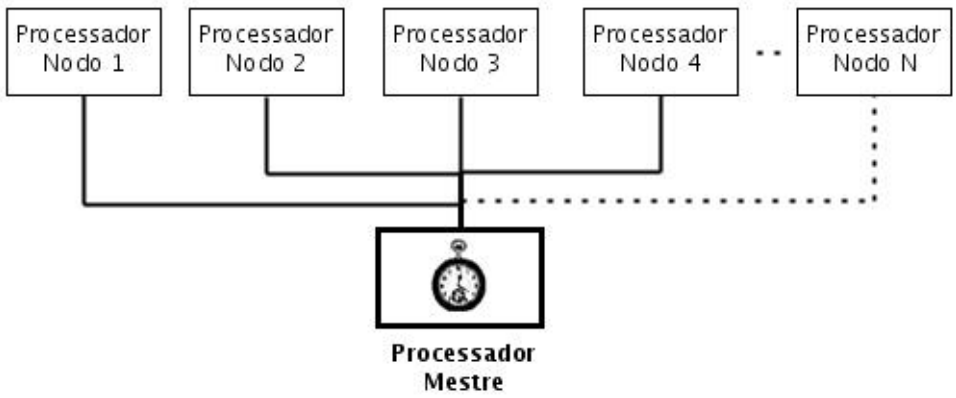

Figura 10. Arquitetura proposta para vários processadores [18].

No processador CTC, o despachante (Figura 11) é o responsável pela interrupção dos processadores nodos e pela programação do temporizador, de acordo a tabela de escala (Figura 12). A tabela, por sua vez, contém apenas os instantes de tempo e os processadores nodos a sofrerem as interrupções. Por exemplo, a primeira entrada do vetor possui como instante de tempo o valor 0 e como processadores a serem interrompidos o valor 252, que convertendo para binário será 11111100. Isso significa que os processadores conectados no pinos 0 e 1 da porta de entrada/saída sofrerão uma interrupção no instante de tempo 0.

```
void despachante_ctc ()
{
 struct item_escala_ctc info_nodos = escala_ctc[scheduleIndexCTC];

 porta_paralela = info_nodos.nodos;

 indice_tabela_ctc = ((++indice_tabela_ctc)%TAMANHO_TABELA_CTC);
 progamarTemporizador(escala_ctc[indice_tabela_ctc].clock);
 ativarTemporizador();
}
```

Figura 11. Código do despachante do processador CTC.

```
#define TAMANHO_TABELA_CTC 6
struct item_escala_ctc escala[SCHEDULE_SIZE_CTC] =
{
{0, 252},  //11111100 - tarefas T3, T0
{10, 248}, //11111000 - tarefas enviarM2, receberM2, T1
{14, 251}, //11111011 - tarefa T4
{50, 252}, //11111100 - tarefas enviarM1 and receberM1
{52, 253}, //11111101 - tarefa T2
{250, 254} //11111110 - tarefa T0
};
```

Figura 12. Tabela de escala para o processador CTC.

O despachante dos processadores escravos (Figura 13) é muito semelhante ao despachante utilizado na arquitetura para apenas um processador. A diferença é que ele não programa mais o temporizador, deixando isso a cargo do processador mestre. Portanto, a estrutura *item_escala_nodo* possui: (i) um sinalizador de retorno preempção; (ii) o identificador da tarefa; (iii) ponteiro para função que representa o código da tarefa. A Figura 14 ilustra um exemplo de tabela de escala de um processador nodo.

```
void despachante_nodo()
{
  struct item_escala_nodo nova_tarefa = tabela_escala[indice_tabela];
  if(existeTarefaEmExecucao) {
     // armazernamento de contexto
  }
  if(nova_tarefa.ehRetornoPreempcao) {
     // recuparação de contexto
  }
  else
  {
     funcao_tarefa = nova_tarefa.ponteiro;
  }
  indice_tabela = ((++indice_tabela) % TAMANHO_TABELA);
}
```

Figura 13. Código do despachante dos processadores nodos.

```
#define TAMANHO_TABELA_NODO 4
struct item_escala_nodo escala[TAMANHO_TABELA_NODO] =
{
{false, 0, (int *)tarefaT0},
{false, 1, (int *)tarefaT1},
{false, 4, (int *)enviarM1},
{false, 0, (int *)tarefaT0}
};
```

Figura 14. Tabela de escala de um processador nodo.

2.3 Resumo

Este capítulo descreveu alguns trabalhos relacionados e fundamentação teórica necessária para o entendimento deste trabalho. Primeiramente, foi descrita a arquitetura da linguagem CML, que é composta por um compilador integrado com ferramentas que realizam análises de aspectos não

funcionais de aplicativos desenvolvidos na linguagem C. Em seguida, foram abordadas a linguagem de especificação JML, e a ferramenta *DBC for C*, que são úteis para especificação comportamental de programas baseadas na técnica de Projeto por Contrato, utilizada pioneiramente pela linguagem Eiffel.

Foram abordados também alguns trabalhos que propõem análises de aspectos não funcionais, como a ferramenta PCAF e o trabalhos propostos por Barreto [6] e Tavares [18]. O primeiro propõe um *framework* para análise de consumo de energia, e o segundo uma ferramenta de análises de restrições temporais em sistemas de tempo real. Esta última é a ferramenta utilizada neste trabalho. Ambas as ferramentas funcionam para aplicativos desenvolvidos na linguagem C.

No próximo capítulo, serão detalhados os atributos que compõem a linguagem CML, o processo de desenvolvimento do seu compilador, além da forma como ele interage com a ferramenta de análise de restrições temporais.

Capítulo 3

Implementação

A linguagem CML tem como objetivo criar um meio, mais próximo do código fonte, para especificação de requisitos não funcionais em aplicativos desenvolvidas na linguagem C. Apesar de ser uma linguagem projetada para dar suporte a qualquer tipo de requisito não funcional, a linguagem CML, neste trabalho, é focada em apenas um tipo de requisito, restrições temporais de sistemas de tempo real.

Assim como a linguagem JML, CML é baseada em anotações embutidas no código fonte do programa que está sendo especificado. Para que não haja interferência na compilação do código fonte do programa, as especificações são inseridas entre blocos de comentários. No Capítulo 2, foi apresentada uma ferramenta de análise de restrições temporais de sistemas de tempo real. As análises são realizadas a partir de um modelo baseado em redes de Petri. O processo como esse modelo é gerado e a maneira como as análises são realizadas, estão fora do escopo deste trabalho. Essa ferramenta é utilizada pela linguagem CML para analisar as restrições impostas pela especificação.

Neste capítulo, serão abordadas as etapas do processo de implementação da linguagem e a maneira como a mesma interage com a ferramenta externa de análise. A primeira seção é dedicada à definição da linguagem, ou seja, o levantamento de todos os atributos necessários para que se possa especificar um determinado programa, tendo em vista as restrições temporais para sistemas de tempo real. Na seção seguinte, é detalhada a metodologia utilizada para construção do compilador, assim como a definição de sua gramática. A última seção é dedicada à definição do ambiente CML e a forma como ele interage com os módulos da ferramenta de análises de restrições temporais.

3.1 Modelagem

O primeiro passo para a construção da linguagem CML foi definir os parâmetros que compõem a especificação do programa. Tendo como base a ferramenta apresentada no Capítulo 2, que possibilita analisar restrições temporais de sistemas de tempo real, é possível levantar os atributos e compor o conjunto de anotações para esse tipo de requisito não funcional. No caso da linguagem CML, as especificações são colocadas dentro de blocos de comentário, com intuito de não interferir na compilação do programa. A Figura 15 ilustra um caso genérico de atribuição de um valor qualquer a um atributo.

```
/**
 * @atributo valor
 */
```

Figura 15. Exemplo de atribuição em CML.

A linguagem CML é uma linguagem de domínio específico (DSL – *Domain Specifc Language*). Portanto, para especificar as restrições temporais a linguagem CML utiliza treze tipos de anotações, que são específicas para requisitos não funcionais, mais especificamente para restrições temporais em sistemas de tempo real. Na medida em que a linguagem seja estendida para suportar mais requisitos, o número de anotações aumenta. A Tabela 1 define esses atributos.

Tabela 1. Conjunto de anotações da linguagem CML.

Anotação	Descrição
@type	indica o tipo da tarefa - pode ser T (tarefa comum) ou M (mensagem)
@name	nome que identifica a tarefa.
@scheduling	método de escalonamento - pode ser P (preemptivo) ou NP (não-preemptivo)
@processor / @bus	nome do processador/canal de comunicação, onde a tarefa será alocada.
@release	tempo de *release*.
@period	tempo de período.
@phase	tempo de fase.
@deadline	*deadline* da tarefa.
@computing / @communication	tempo de computação da tarefa, ou de comunicação da mensagem.
@preceds	relação de precedência com uma lista de tarefas.
@excludes	relação de exclusão com uma lista de tarefas.

Todas as restrições temporais são expressas em TTU (*task time unit*) que é a menor parte indivisível de uma tarefa, em que outra tarefa não pode interromper a sua execução. A Figura 16 ilustra um exemplo completo de especificação de uma tarefa em CML. Nesse caso, o parâmetro *@type* indica o tipo de tarefa que se está especificando, o valor *T* significa que é uma tarefa. A tarefa *T1* trabalha com o método de escalonamento não-preemptivo, representado pelo valor *NP*, está alocada ao processador *P1*, tem um tempo de *release* de 1 TTU, período de 9 TTU, fase de 1 TTU, um *deadline* de 9 TTU e um WCET (*computing*) de 8 TTU.

```
/**
 * @type T
 * @name T1
 * @scheduling NP
 * @processor P1
 * @release 1
 * @period 9
 * @phase 1
 * @deadline 9
 * @computing 1
 */
void T1(){ //implementação }
```

Figura 16. Exemplo de especificação de uma tarefa em CML.

A Figura 17 ilustra um exemplo de tarefa de comunicação. Nesse exemplo a mensagem *M1* segue da tarefa *T1* para a tarefa *T2* através do canal de comunicação (*bus*) *B1*. Isso acontece pelo fato de que a tarefa *T1* (Figura 16) precede a mensagem *M1*, e esta, por sua vez, precede a tarefa *T2* (Figura 18). O tempo de comunicação é dado pelo atributo *@communication*. É importante ressaltar que mensagem é enviada pela tarefa *sendM1*, no processador de origem, e recebida pela tarefa *receiveM1*, no processador de destino. Entretanto, é necessário especificar apenas a mensagem, em qualquer uma das duas tarefas.

```
/**
 * @type M
 * @name M1
 * @bus B1
 * @communication 2
 * @preceds {T1}
 */
void sendM1(){ //implementação }
      .
      .
      .
void receiveM1(){ //implementação }
```

Figura 17. Especificação de uma mensagem em CML.

A relação de exclusão representada pela palavra-chave *@excludes* demonstra um caso onde a *T2* não pode iniciar sua execução antes da tarefa *T4* ter sido finalizada, ou seja, *T2* não pode interferir na execução de *T4*. A relação de precedência pode acontecer com mais de uma tarefa, como ilustrado na Figura 18. Neste caso, as tarefas que a tarefa especificada precede são colocadas no formato de uma lista, e os elementos dela são separados por vírgula.

```
/**
 * @type T
 * @name T2
 * @scheduling NP
 * @processor P1
 * @release 1
 * @period 9
 * @phase 1
 * @deadline 9
 * @computing 1
 * @preceds {M1,T3}
 * @excludes {T4}
 */
void T2(){ //implementação }
```

Figura 18. Especificação de uma tarefa com relação de precedência e exclusão.

3.2 Compilador da linguagem CML

3.2.1 Processo de Compilação

O processo de compilação de uma determinada linguagem é geralmente dividido em duas etapas: análise e síntese. A etapa de análise pode ser subdividida nas etapas de análise léxica e sintática. Na etapa de síntese, a geração de código é tratada. A seguir essas três etapas são detalhadas.

1. **Análise Léxica:** tem a preocupação de dividir o código fonte em *tokens* [20]. São exemplos de *tokens*: palavras-chave, números, cadeias de caracteres e caracteres especiais de pontuação. Os padrões ignorados não são considerados *tokens*, como espaços em branco e identificadores de quebra de linha.

2. **Análise Sintática:** tem a preocupação de extrair um significado do código fonte [20]. Os *tokens* identificados no processo de análise léxica servem como entrada para o processo de análise sintática. Portanto, o analisador sintático tenta encontrar um significado determinado pela seqüência em que os *tokens* são consumidos. Essas seqüências são definidas por uma gramática, ou seja, uma gramática define as regras para a construção sintaticamente correta dos programas. Se um programa não obedecer às regras definidas na gramática, durante o processo de análise sintática, o compilador exibe uma mensagem de erro. EBNF (*Extended Backus-Naur-Form*) [20] é uma notação que propicia a especificação de uma gramática sem ambigüidades. Uma especificação utilizando essa notação é feita em termos de regras de produção, em que uma regra de produção significa que um elemento da gramática pode ser composto por outros elementos da gramática. A Figura 19 ilustra a especificação de um elemento de gramática utilizando a notação EBNF. A saída do analisador sintático é a AST (*Abstract Syntax Tree*).

```
GRAMMAR_ELEMENT := Lista de Elementos
                 | Lista Alternativa de elementos
```

Figura 19. Notação EBNF.

3. **Geração de código:** após o código passar pelos processos de análise léxica e análise sintática, o compilador utiliza a AST para gerar o código objeto.

Pela facilidade no desenvolvimento e por ter uma grande gama de aplicativos auxiliares, a linguagem Java foi escolhida para a implementação do compilador da linguagem CML. Ainda por questões de agilidade no processo de desenvolvimento e de facilidade de implementação, também será utilizada a ferramenta geradora de *parsers*[2] JavaCC (Compiler Compiler) [21]. Um gerador de *parsers* transforma uma especificação de uma gramática em um programa que consegue reconhecê-la. Como a linguagem CML funciona apenas dentro de blocos de comentários, isso faz com que ela seja encarada como uma linguagem secundária em relação à linguagem de programação principal utilizada para compilar o programa. Portanto, as anotações

[2] Programa de computador (ou apenas um componente de um programa) que serve para analisar a estrutura gramatical de uma entrada, manipulando os segmentos de texto ou símbolos que podem ser manipulados.

CML inseridas no código fonte não podem interferir no processo de compilação do código fonte original.

O compilador CML tem o objetivo de reconhecer os atributos das especificações de cada tarefa e gerar um código a partir delas. Nesse caso, o compilador gera uma representação intermediária (estrutura de objetos Java) da especificação. A Figura 20 ilustra o diagrama de classes que representa essa estrutura intermediária gerada pelo compilador CML.

A classe *AbstractTask* representa uma tarefa generalizada, em que seus atributos servem tanto para as tarefas normais, que são representadas pela classe *Task*, quanto para as tarefas de comunicação, que são representadas pela classe *CommunicationTask*. As classes que representam as tarefas possuem os mesmos atributos utilizados na especificação ilustrados nas Figuras 16, 17 e 18. Por fim, a classe *Specification* reúne uma lista de tarefas que representam a especificação do programa.

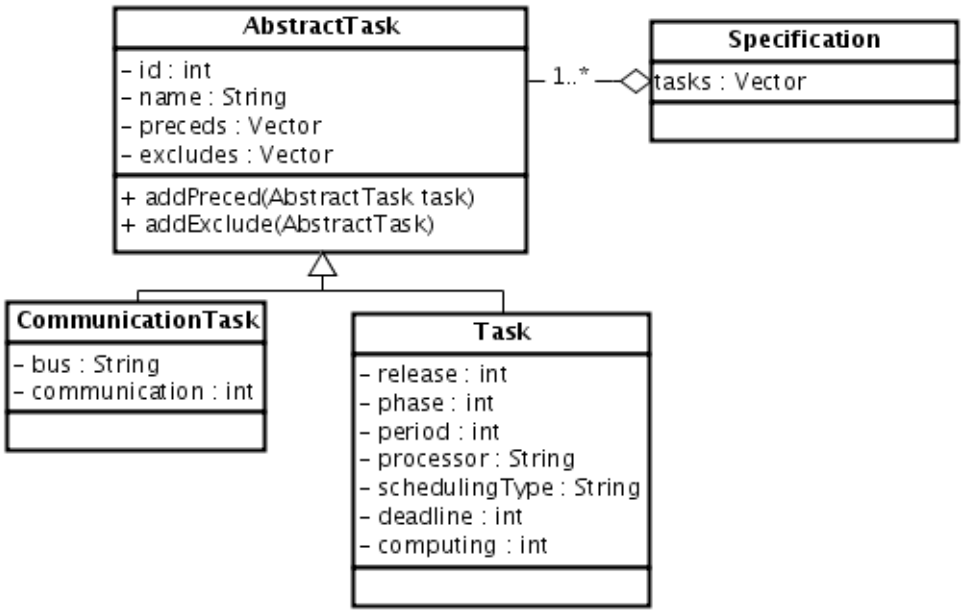

Figura 20. Diagrama de classes entidades da linguagem CML.

Uma vez as especificações do software estejam estruturadas em objetos Java, pode-se definir melhor a forma como essas especificações podem ser enviadas para as ferramentas externas de análises. Para a ferramenta proposta por Barreto [6], é necessário que as especificações estejam em um formato XML, que será discutido com mais detalhes no final deste capítulo.

3.2.2 JavaCC

A construção de analisadores léxicos e sintáticos é uma tarefa muito complexa e demanda muito tempo na construção de um compilador. É comum em projetos de compiladores a utilização de ferramentas que automatizem a etapa de análise. JavaCC é uma ferramenta desenvolvida pela *Sun Microsystems* [22], muito poderosa na geração automática de analisadores sintáticos para aplicações desenvolvidas em Java. A geração automática é dada através de um arquivo de especificação, em que são definidos os *tokens* e a gramática da linguagem, de maneira muito

similar à notação EBNF. A Figura 21 ilustra a estrutura de um arquivo de especificação JavaCC. A palavra reservada *options* indica a seção onde são configurados alguns parâmetros. Por exemplo, a versão da maquina virtual Java que o compilador suportará. A seção demarcada pelas palavras reservadas *PARSER_BEGIN* e *PARSER_END* indica a área em que é definida a classe principal do analisador. A palavra reservada *TOKEN* indica a seção onde são definidos os *tokens* da linguagem que se está projetando. Logo após a definição dos *tokens* é a seção reservada para a especificação da gramática da linguagem. Cada elemento não-terminal da gramática é definido no formato de um método da linguagem Java.

```
options { ... }

PARSER_BEGIN(Nome_do_Analisador)
...
PARSER_END(Nome_do_Analisador)

TOKEN : { ... }

void grammarElement(){ }
```

Figura 21. Estrutura de um arquivo de especificação JavaCC.

Uma vez a linguagem esteja definida no arquivo, o mesmo é compilado, e então o JavaCC gera automaticamente classes Java que representam os analisadores léxico e sintático da linguagem. Uma das grandes vantagens do JavaCC é que a especificação do analisador léxico e sintático é feita no mesmo arquivo. Geralmente a construção dos analisadores é feita por ferramentas separadas, como Lex [23] e Yacc [24] , que são geradores de analisadores léxicos e sintáticos, para a linguagem C, e JFlex [25] e CUP [26] para a linguagem Java.

3.2.3 Gramática da linguagem CML

A especificação da gramática de uma linguagem de programação é iniciada a partir dos *tokens* (palavras-chave) que compõem a linguagem em questão. Para isso, a definição da linguagem CML, descrita na primeira seção deste capítulo, é de extrema importância para se ter uma idéia mais clara de como a linguagem vai funcionar em termos de regras para reconhecimento de padrões léxicos e sintáticos no código fonte.

A ferramenta JavaCC se utiliza de um arquivo de especificação, onde o projetista do compilador define os *tokens*, expressões regulares e os elementos da gramática, nessa mesma ordem. Esse arquivo é então interpretado pelo JavaCC e um código Java do *parser* é gerado automaticamente. Em seguida serão descritos os elementos principais desse arquivo de especificação e a estrutura gerada pelo JavaCC.

Tokens e Expressões Regulares

Um dos *tokens* mais importantes para linguagem CML é o *token* responsável pelo início do bloco de comentário. Isso significa que, reconhecendo o *token* "/**" o *parser* mudará de estado e passará a reconhecer apenas *tokens* definidos para esse estado.

A Figura 22 ilustra o trecho onde é definido este *token*. Nesse caso, a palavra-chave <*DEFAULT*> significa que o *token* <*START_COMMENT*> é definido no estado padrão, e quando reconhecido muda para o estado <*IN_COMMENT_BLOCK*>.

```
<DEFAULT>
TOKEN : { <START_COMMENT : "/**" >  : IN_COMMENT_BLOCK }
```

Figura 22. Definição do *token* <*START_COMMENT*>.

A Figura 23 ilustra o caso inverso, ou seja, quando o *token* <END_COMMENT>, que sinaliza o fim do bloco de comentário, é reconhecido, o *parser* volta para o estado padrão.

```
<IN_COMMENT_BLOCK>
TOKEN : {  < END_COMMENT: "*/" > : DEFAULT }
```

Figura 23. Definição do *token* <*END_COMMENT*>.

Na Figura 24, a palavra-chave SKIP significa que os *tokens* reconhecidos pela expressão regular definida entre as chaves serão ignorados. Nesse caso, a expressão regular reconhece qualquer *token.* Como o *token* que representa o início do bloco de comentário já foi definido previamente, o *parser* ignora qualquer outro *token* no estado padrão e que esteja fora dos blocos de comentário.

```
SKIP : { <~[]> }
```

Figura 24. Expressão regular que ignora qualquer outra coisa.

Por fim, a Figura 25 ilustra a definição dos *tokens* que são reconhecidos no estado <*IN_COMMENT_BLOCK*>. Os *tokens* que representam os atributos de especificação já foram descritos na primeira seção deste capítulo e, portanto, não é necessário descrevê-los novamente. É importante ressaltar os *tokens* definidos com prefixo # são considerados macros, e podem ser reutilizados na definição de outros *tokens*. Por exemplo, as macros <*#LETTER*> representa qualquer letra do alfabeto além do caractere "_", e a macro <*#DIGIT*> representa qualquer dígito de 0 a 9. Essas macros são reutilizadas na construção de expressões regulares de *tokens* como <*STRING*> e <*INT*>. No primeiro caso, a expressão regular reconhece uma letra seguida por uma seqüência de letras ou dígitos, formando uma seqüência de caracteres. Já no segundo caso, a expressão regular reconhece uma seqüência de um ou mais dígitos, formando um número inteiro.

```
< IN_COMMENT_BLOCK >
TOKEN : { < TASK_TYPE : "@type" >
            | < TASK_NAME : "@name" >
            | < RELEASE : "@release" >
            | < PERIOD : "@period" >
            | < PHASE : "@phase" >
            | < PROCESSOR : "@processor" >
            | < BUS : "@bus" >
            | < SCHEDULING : "@scheduling" >
            | < COMPUTING : "@computing" >
            | < COMMUNICATION : "@communication" >
            | < DEADLINE : "@deadline" >
            | < PRECEDS : "@preceds" >
            | < EXCLUDES : "@excludes" >
            | < COMMUNICATION_VALUE: "M" >
            | < TASK_VALUE: "T" >
            | < SCHEDULING_VALUE: "NP" | "P" >
            | < STRING: <LETTER> (<LETTER> | <DIGIT>)* >
            | < INT: (<DIGIT>)+ >
            | < COMMA: "," >
            | < L_BRAC: "{" >
            | < R_BRAC: "}" >
            | <#LETTER: ["$","A"-"Z","_","a"-"z"]>
            | <#DIGIT: ["0"-"9"]>}
```

Figura 25. Definição dos *tokens* do estado <*IN_COMMENT_BLOCK*>.

Gramática

Uma vez os *tokens* estejam completamente definidos, é necessário fazer com que os mesmos tenham utilidade no processo de compilação, ou seja, até esta etapa do processo de compilação, não houve a preocupação se a ordem em que as palavras (*tokens*) são reconhecidas está correta.

Como foi dito anteriormente, a gramática de uma linguagem define a maneira como o compilador deve reconhecer a seqüência de *tokens*. É através de sua definição que se consegue analisar sintaticamente o conjunto de *tokens* reconhecido previamente.

Em JavaCC os elementos da gramática são definidos como métodos da linguagem Java e utiliza uma notação muito parecida com a notação EBNF, em que um elemento da gramática pode ser decomposto em outros elementos, até chegar em um elemento terminal. Podemos entender a estrutura da gramática como sendo uma árvore, onde a raiz é o elemento mais abstrato da gramática e as folhas são os elementos terminais.

No caso da linguagem CML o elemento mais abstrato, ou seja, a raiz da árvore, seria um bloco de comentário com atribuições dentro dele. A Figura 26 ilustra esse elemento. O método *init()* representa um elemento da gramática que pode ser decomposto no elemento *statement()* ou até chegar o terminal <*EOF*>, que é o *token* que indica final do arquivo.

```
void init() : { }
{
  (<START_COMMENT>(statement())+<END_COMMENT>)+
  | <EOF> { }
}
```

Figura 26. Elemento mais abstrato da gramática da linguagem CML.

A definição do elemento *statement()* está ilustrado na Figura 27. Esse elemento é composto por vários outros elementos que representam, cada um, um atributo de especificação da tarefa. Alguns deles foram omitidos, pois suas definições são iguais às dos outros elementos.

```
void statement() : { }
{
      taskType()
      | taskName()
      | release()
          .
          .
          .
      | excludes()

}
```

Figura 27. Definição do elemento *statement()*.

O definição do elemento *taskType()* está ilustrada na Figura 28. Esse elemento indica o início de uma tarefa, ou seja, quando o atributo indicado pelo *token* <*TASK_TYPE*> é reconhecido, uma nova instancia da classe *AbstractTask* é criada, baseada no tipo da tarefa (tarefa comum ou tarefa de comunicação).

```
void taskType(): { AbstractTask task; }
{
      <TASK_TYPE> (<TASK_VALUE> {task = new Task();}
      | <COMMUNICATION_VALUE> {task = new CommunicationTask();})
      {
         specification.setCurrentTask(task);
      }
}
```

Figura 28. Definição do elemento *taskType()*.

A Figura 29 ilustra a definição do elemento *taskName()* . À variável *value* é atribuído o valor do atributo *@task_name*. Esse valor é composto por uma cadeia de caracteres, representado pelo *token* <*STRING*>.

```
void taskName(): { Token value; }
{
    <TASK_NAME> value=<STRING>
    {
       specification.getCurrentTask().
       setName(value.toString());
    }
}
```

Figura 29. Definição do elemento *taskName()*.

O restante dos elementos é definido de forma similar aos ilustrados previamente. Entretanto, os atributos que tem como valor uma lista, como *@preceds* e *@excludes*, são tratados de maneira diferente dos demais. Esse tipo de caso está ilustrado na Figura 30. O elemento

preceds() é composto pelo *token* <*PRECEDS*>, que representa o atributo *@preceds*, e o elemento *precedsList()* entre os *tokens* <*L_BRAC*> *e* <*R_BRAC*>. O elemento *precedsList()*, por sua vez, é composto pelo elemento *precedsElement()* e uma seqüência, de uma ou mais vezes, do *token* <*COMMA*> seguido do elemento *precedsElement()*. Este é formado apenas por um *token* <*STRING*>, que representa o nome da tarefa, e então é adicionada à lista de tarefas às quais a tarefa especificada precede. A definição do elemento que trata do atributo *@excludes* é dada da mesma maneira.

```
void preceds(): {}
{
   <PRECEDS> <L_BRAC> precedsList() <R_BRAC>
}

void precedsList(): {}
{
   precedsElement() (<COMMA> precedsElement())*
}

void precedsElement(): { Token value; }
{
  value=<STRING>  {
      specification.getCurrentTask().getPreceds().add(
      specification.findTaskByName(value.toString()));
  }
}
```

Figura 30. Conjunto de elementos que formam o elemento *preceds()*.

A versão completa do arquivo de especificação do *parser* da linguagem CML para ferramenta JavaCC pode ser encontrada no Apêndice A deste trabalho.

3.3 Integração com ferramentas de análise

O resultado da compilação das anotações, inseridas dentro dos blocos de comentário do código fonte, é uma representação intermediária da especificação. Essa informação precisa ser traduzida em um formato que seja conhecido das ferramentas externas de análise. A ferramenta apresentada por Barreto [6] e estendida por Tavares [18], possui um gerador de modelo de redes de Petri baseado em um arquivo XML de especificação, ou seja, a partir do XML, a ferramenta gera uma representação em redes de Petri, das especificações. Partindo das especificações modeladas em uma rede de Petri, um gerador de escalas, que também integra a ferramenta, verifica se é possível gerar uma escala para as especificações dadas. Através desse gerador de escalas, é possível responder ao usuário da linguagem CML, se um sistema de tempo real atende às restrições impostas nas especificações. Em seguida serão detalhados os módulos que compõem a ferramenta de análise de restrições temporais, como também a implementação dos módulos que integram o ambiente CML.

3.3.1 Módulos da ferramenta de análise de restrições temporais

Gerador do modelo de redes de Petri

A especificação gerada no formato XML serve como entrada para a ferramenta que realiza as análises de restrições temporais em sistemas de tempo real. Essa ferramenta traduz as especificações em um modelo de redes de Petri, e foi desenvolvida na linguagem de programação Java.

O modelo gerado é armazenado na memória principal e dois tipos de arquivos são gerados: (i) um arquivo PNML (*Petri Net Markup Language*); (ii) e um arquivo de formato proprietário, utilizado apenas pela ferramenta de análise. PNML é um formato de arquivo XML muito utilizado por várias ferramentas utilitárias de redes de Petri. Nesse caso, o arquivo PNML é gerado para que o modelo gerado possa ser exportado e visualizado por outras ferramentas. Mais informações sobre o formato PNML podem ser encontradas em [27]. O outro arquivo representa a mesma rede de Petri contida no arquivo PNML, porém em um formato proprietário que serve como entrada para o gerador de escalas. A forma como o modelo de rede de Petri é criado, assim como uma abordagem mais profunda em relação às análises realizadas para determinar as escalas, estão fora do escopo deste trabalho.

Gerador de escalas

O gerador de escalas interpreta o arquivo que representa o modelo de redes de Petri e realiza uma série de análises para determinar se é possível ou não uma escala para as especificações dadas. Se sim, um outro arquivo é gerado, contendo uma escala que respeita as restrições temporais impostas na especificação de cada tarefa.

O gerador de escalas foi desenvolvido na linguagem de programação C e recebe como parâmetros de entrada o arquivo de formato proprietário, gerado pelo gerador de modelo descrito no item anterior, e o nome do arquivo em que será armazenada a escala, caso seja realmente possível gerá-la.

3.3.2 Ambiente CML

O ambiente CML é formado pelo compilador da linguagem CML e por uma classe controladora, que é responsável por controlar e solicitar a execução dos serviços de cada módulo, tanto de dentro do ambiente (compilador CML) como de fora (geradores de escala e de modelo de redes de Petri). É também formado por um módulo responsável por transformar a representação intermediária, retornada pelo compilador, em um arquivo XML. Esse arquivo XML é dado como entrada para o gerador de modelo de redes de Petri, que retorna o modelo em um formato proprietário (.sch). Então a classe controladora solicita uma escala para o gerador de escalas, fornecendo como argumento o modelo gerado. Por fim, a classe controladora interpreta a saída do gerador de escalas e informa ao usuário se é possível atender às especificações impostas.

Estrutura de pacotes

O ambiente CML é formado por quatro pacotes de classes Java:

- **br.upe.dsc.cml**: contém apenas a classe controladora, chamada de *CMLEngine*, que é a interface externa do ambiente. Essa classe também é responsável por controlar a

execução do compilador, que retorna uma representação intermediária (estrutura de objetos Java), do gerador de modelo de redes de Petri, que retorna um arquivo no formato compreendido pelo gerador de escalas. Este último, por se um aplicativo desenvolvido na linguagem C, obriga o ambiente a utilizar uma ferramenta que permita a comunicação com programas externos. Para isso, a ferramenta ExpectJ [28] foi utilizada.

- **br.upe.dsc.cml.compiler:** contém as classes do *parser* geradas automaticamente pela ferramenta JavaCC.

- **br.upe.dsc.cml.compiler.entities:** contém um conjunto de classes que formam a representação intermediária que é dada como retorno do compilador.

- **br.upe.dsc.cml.compiler.io**: contém a classe *RTTWriter*, que traduz da representação intermediária para o formato XML.

A Figura 31 ilustra um diagrama de blocos que representa a interação do ambiente CML e seu compilador com a ferramenta de análise de sistemas de tempo real.

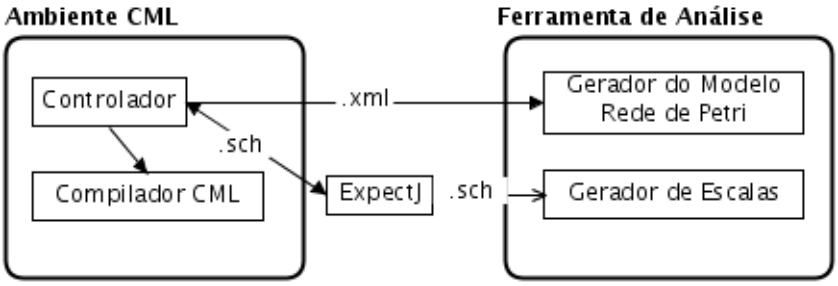

Figura 31. Integração do ambiente CML com a ferramenta externa de análise.

Geração do formato XML

O arquivo XML é estruturado de forma similar a representação intermediária gerada pelo compilador da linguagem CML. A Figura 32 ilustra um exemplo de especificação no formato XML com as tarefas, seus atributos, restrições temporais e relações com outras tarefas.

```
<realtime-table>
  <task release="1" period="9" phase="1" processor="P2" schedulingModel="NP"
oid="1088076" name="tarefa1">
  <time>
     <computing value="1"/>
     <deadline value="9"/>
  </time>
  </task>
  <task release="0" period="10" phase="2" processor="P1"
schedulingModel="NP" oid="131577" name="tarefa2">
     <time>
       <computing value="3"/>
       <deadline value="8"/>
     </time>
     <preceds>
       <message-ref name="M1"/>
     </preceds>
  </task>
  <message bus="T1" oid="30377347" name="M1">
     <time>
        <comunication value="1"/>
     </time>
     <preceds>
       <task-ref name="tarefa1"/>
     </preceds>
  </message>
</realtime-table>
```

Figura 32. Exemplo de especificação no formato XML.

Para transformar a representação intermediária, retornada pelo compilador, em um modelo XML, várias ferramentas podem ser utilizadas, como Xerces [29], DOM (*Document Object Model***)** [30], JDOM (*Java Document Object Model*) [31], entre outras. A representação é traduzida em nós hierárquicos, e com a utilização de uma dessas ferramentas, é possível escrever um arquivo XML que a represente. Neste trabalho, a ferramenta DOM foi utilizada.

3.4 Resumo

Este capítulo descreveu a implementação da linguagem CML. Primeiramente foi descrita a modelagem da linguagem baseada na ferramenta de análise de restrições temporais. Em seguida, foram descritas brevemente as etapas necessárias para a construção de um compilador, a ferramenta utilizada para geração automática do analisador sintático, e a definição da gramática da linguagem CML. Por fim, foram detalhados os módulos que compõem o ambiente CML, e a ferramenta de análise de restrições temporais, além da forma como eles interagem.

No próximo capítulo, serão abordados alguns sistemas de tempo real anotados com a linguagem CML, a fim de validar a proposta apresentada por este trabalho.

Capítulo 4

Validação da Proposta

Este capítulo tem como objetivo apresentar algumas aplicações para demonstrar a utilidade prática da linguagem CML, além de validar proposta apresentada neste trabalho. Primeiramente, são exibidas as tarefas com suas especificações, em seguida, são apresentadas as especificações no formato XML, que é o formato utilizado pela ferramenta de análise de sistemas de tempo real. Por fim, a escala gerada por essa ferramenta é exibida em um diagrama de tempo.

O processo aqui utilizado é o mesmo descrito no Capítulo 3, ou seja, a partir das tarefas especificadas, o ambiente CML traduz essas especificações em um formato entendido pela ferramenta de análise. Essa ferramenta, por sua vez, transforma as especificações em um modelo de rede de Petri, e as envia para um gerador de escalas, que determina se essas especificações podem ser atendidas ou não.

Duas aplicações de tempo real são abordadas neste capítulo:

- **Semáforo:** consiste em um semáforo de trânsito, que possui três luzes, obedecendo a uma ordem determinada e a um período de execução, ou seja, o tempo que cada uma delas fica acesa. Esse primeiro estudo de caso é muito simples e, portanto, tem o objetivo principal de introduzir o leitor aos conceitos básicos por meio de uma aplicação prática.

- **Sistema de Monitoramento de Veículos:** consiste em um conjunto de sensores que verificam se os componentes de um veículo estão funcionando corretamente. Esses sensores são capazes de verificar motor, freios, água, temperatura e câmbio, além de enviar os resultados para um sistema que os exibe em um painel conectado. Esse sistema utiliza dois processadores para atender os vários sensores.

Todas as aplicações consideram a utilização de plataforma 8051 [32], entretanto, o gerador de escalas utilizado pode ser estendido para suportar outras plataformas [6][18]. Da mesma forma, a linguagem CML suporta apenas especificação de requisitos para sistemas de tempo real, mas pode ser estendida a fim de suportar uma maior variedade de requisitos não funcionais, como consumo de energia, tamanho do arquivo, desempenho, entre outros.

4.1 Semáforo

Este estudo de caso tem como objetivo especificar um sistema que simula um semáforo de trânsito. É um estudo de caso simples com propósitos didáticos e, portanto de fácil compreensão por parte do leitor.

Um semáforo possui três luzes: uma verde, uma amarela e uma vermelha. Cada luz acende e depois de um tempo apaga, na mesma ordem, uma por vez. A luz verde fica acesa por cinco unidades de tempo, a luz amarela fica acesa por uma unidade de tempo e a luz vermelha por três unidades de tempo. Portanto, são nove unidades de tempo por período.

O semáforo pode ser modelado como três tarefas, cada uma representando uma luz. A especificação de cada tarefa é composta pelos atributos fase, *release*, WCET (computação), *deadline* e período. Todas as tarefas estão alocadas a um mesmo processador e o método de escalonamento de todas as tarefas é não-preemptivo. Existe uma relação de precedência entre as tarefas *Verde* e *Amarelo*, e entre as tarefas *Amarelo* e *Vermelho*. A Tabela 2 resume a especificação de todas as tarefas do semáforo, com suas restrições temporais, e as relações entre tarefas.

Tabela 2. Especificação das Tarefas do Semáforo.

Tarefa	Fase	*Release*	WCET	*Deadline*	Período
Verde	0	0	1	9	9
Amarelo	5	0	1	9	9
Vermelho	6	0	1	9	9
Relação entre Tarefas					
Verde PRECEDE Amarelo Amarelo PRECEDE Vermelho					

A Figura 33 ilustra as tarefas Verde, Amarelo e Vermelho com suas respectivas especificações, definidas na Tabela 2.

```
/**                    /**                    /**
 * @type T             * @type T             * @type T
 * @name vermelho      * @name amarelo        * @name verde
 * @processor P1       * @processor P1        * @processor P1
 * @scheduling NP      * @scheduling NP       * @scheduling NP
 * @phase 6           * @phase 5             * @phase 0
 * @release 0         * @release 0           * @release 0
 * @computing 1       * @computing 1         * @computing 1
 * @deadline 9        * @deadline 9          * @deadline 9
 * @period 9          * @period 9            * @period 9
 */                    * @preceds {vermelho}  * @preceds {amarelo}
void vermelho()        */                    */
{ ... }                void amarelo()         void verde()
                       { ... }               { ... }
```

Figura 33. Especificação das tarefas *Verde*, *Amarelo* e *Vermelho*.

As especificações no código fonte são compiladas e traduzidas em um formato XML, para em seguida, ser enviado para a ferramenta que realiza as análises. A Figura 34 ilustra esse arquivo gerado.

```
<realtime-table>
<task release="0" period="9" phase="6" processor="P1" s
      chedulingModel="NP" oid="1088076" name="vermelho">
        <time>
            <computing value="1"/>
            <deadline value="9"/>
        </time>
    </task>
    <task release="0" period="9" phase="5" processor="P1"
schedulingModel="NP" oid="131577" name="amarelo">
        <time>
            <computing value="1"/>
            <deadline value="9"/>
        </time>
        <preceds>
            <task-ref name="vermelho"/>
        </preceds>
    </task>
    <task release="0" period="9" phase="0" processor="P1"
schedulingModel="NP" oid="30377347" name="verde">
        <time>
            <computing value="1"/>
            <deadline value="9"/>
        </time>
        <preceds>
            <task-ref name="amarelo"/>
        </preceds>
    </task>
</realtime-table>
```

Figura 34. Especificação do semáforo no formato XML.

O arquivo de XML é levado para a ferramenta de análise, que traduz as especificações em um modelo de rede de Petri que serve como entrada para um gerador de escalas da própria ferramenta. O resultado final é dado baseado no resultado do gerador de escalas, ou seja, se existe ou não uma escala possível para as especificações impostas.

A Figura 35 ilustra o resultado gerado pelo ambiente CML. A classe principal (*CMLEngine*) é chamada, e o arquivo anotado é passado como argumento. A geração do XML é encapsulada neste processo, ou seja, depois que o compilador CML retornar uma estrutura de objetos Java, o ambiente traduz essa estrutura em um arquivo XML e em seguida faz uma chamada a ferramenta de análise, que retorna se a escala foi gerada com sucesso ou não.

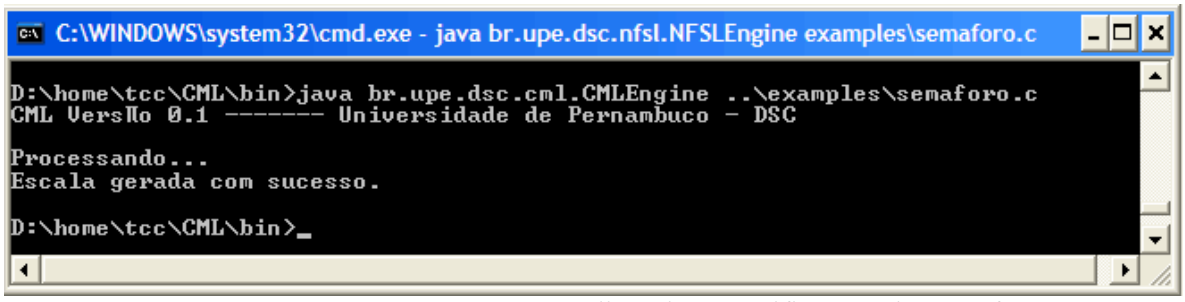

Figura 35. Resultado positivo das análises das especificações do Semáforo.

A Figura 36 ilustra o diagrama de tempo para a escala gerada. Nesse caso, as restrições foram atendidas, e conseqüentemente a escala foi gerada com sucesso.

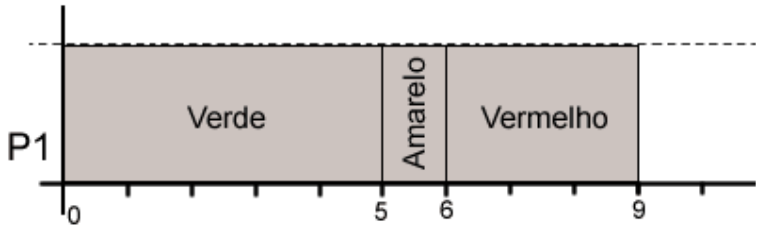

Figura 36. Diagrama de tempo da escala gerada para o Semáforo.

Pode-se admitir, de forma hipotética, que o valor do atributo período da tarefa *Vermelho* seja 5, dessa forma tem-se que o tempo levado para que essa tarefa volte a ser executada é menor do que o necessário para as outras tarefas terminarem de executar. Portanto, o gerador de escalas não consegue gerar nenhuma escala possível, o que representa que as especificações não foram atendidas. A Figura 37 ilustra esse caso.

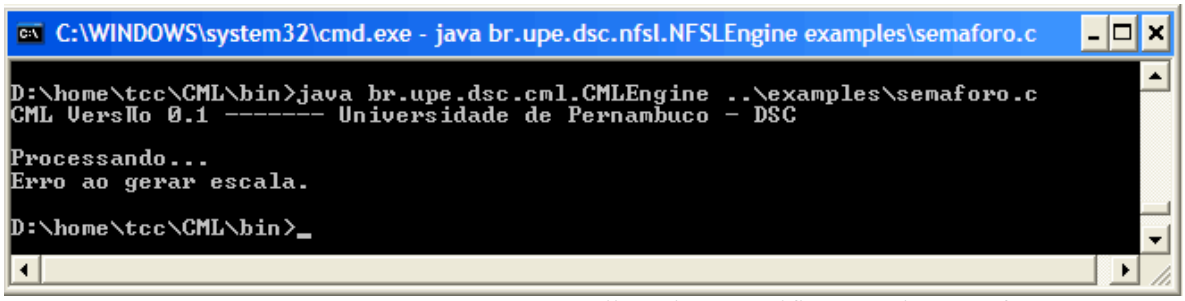

Figura 37. Resultado negativo das análises das especificações do Semáforo.

4.2 Sistema de monitoramento de veículos

No sistema descrito na seção anterior, apenas um processador é utilizado. A fim de exemplificar melhor a especificação de um sistema de tempo real, o sistema apresentado nesta seção utiliza é dois processadores, o que permite abordar muitos conceitos que não foram abordados na seção anterior. Esse é um sistema real utilizado em [18].

O sistema de monitoramento de veículos consiste em um conjunto de sensores, que são utilizados para verificar se os componentes de um veículo estão funcionando corretamente. Se algum componente falhar, o sistema informa ao motorista através de um painel externo. Esse sistema utiliza dois processadores, pois um processador só, não poderia atender aos vários sensores, além do micro-controlador utilizado (8051) possuir apenas quatro portas de entrada/saída de 8 bits.

A Tabela 3 mostra o conjunto de tarefas que compõem o sistema, elas são responsáveis por verificar o funcionamento do motor (*TV* e *TR*), freios (*TB*), água (*TW*), temperatura (*TT*) e câmbio (*TG*). No total são 14 tarefas, incluindo uma tarefa de comunicação (*M1*). Cada componente tem duas tarefas, uma tarefa para leitura do sensor e outra para o mapeamento dos dados obtidos pela leitura. Por exemplo, a tarefa TV_0 é responsável por ler a velocidade, e a tarefa TV_1 por mapear os dados que representam a velocidade. A tarefa *TRA* é uma exceção, pois ela leva os dados obtidos para um sistema externo que os exibe em um painel conectado a ele.

Tabela 3. Especificação do Sistema de Monitoramento de Veículos.

Tarefa	*Release*	WCET	*Deadline*	Período	Proc./ Canal	De	Para
TV_0	0	231	20000	120000	P1	-	-
TV_1	20000	5487	40000	120000	P1	-	-
TB_0	20000	221	40000	120000	P1	-	-
TB_1	40000	236	60000	120000	P1	-	-
TR_0	40000	232	60000	120000	P1	-	-
TR_1	60000	238	80000	120000	P1	-	-
TRA	80000	2444	120000	120000	P1	-	-
TW_0	0	237	20000	120000	P2	-	-
TW_1	20000	241	40000	120000	P2	-	-
TT_0	20000	259	40000	120000	P2	-	-
TT_1	40000	234	60000	120000	P2	-	-
TG_0	40000	224	60000	120000	P2	-	-
TG_1	60000	236	80000	120000	P2	-	-
M_1	-	1700	-	-	B1	TG_1	TRA

As Figuras 38, 39 e 40 ilustram as tarefas alocadas ao processador *P1*. Elas têm a responsabilidade de verificar velocidade (Figura 38 e 40) e freios (Figura 39). Todas elas utilizam método de escalonamento não-preemptivo.

```
/**                                  /**
 * @type T                           * @type T
 * @name lerVelocidade               * @name mapearVelocidade
 * @processor P1                     * @processor P1
 * @scheduling NP                    * @scheduling NP
 * @phase 0                          * @phase 0
 * @release 0                        * @release 20000
 * @computing 231                    * @computing 5487
 * @deadline 20000                   * @deadline 40000
 * @period 120000                    * @period 120000
 */                                  */
void lerVelocidade(){ ... }      void mapearVelocidade() { ... }
```

Figura 38. Especificação das tarefas *TV0* e *TV1*.

```
/**                                  /**
 * @type T                           * @type T
 * @name lerFreio                    * @name mapearFreio
 * @processor P1                     * @processor P1
 * @scheduling NP                    * @scheduling NP
 * @phase 0                          * @phase 0
 * @release 40000                    * @release 60000
 * @computing 236                    * @computing 238
 * @deadline 60000                   * @deadline 80000
 * @period 120000                    * @period 120000
 */                                  */
void lerFreio (){ ... }          void mapearFreio (){ ... }
```

Figura 39. Especificação das tarefas *TB0* e *TB0*.

```
/**                          /**
 * @type T                    * @type T
 * @name lerRPM               * @name mapearRPM
 * @processor P1              * @processor P1
 * @scheduling NP             * @scheduling NP
 * @phase 0                   * @phase 0
 * @release 40000             * @release 60000
 * @computing 232             * @computing 238
 * @deadline 60000            * @deadline 80000
 * @period 120000             * @period 120000
 */                           */
void lerRPM (){ ... }        void mapearRPM (){ ... }
```

Figura 40. Especificação das tarefas *TR0* e *TR1*.

As Figuras 41, 42 e 43 ilustram as tarefas alocadas ao processador *P2*. Elas têm a responsabilidade de verificar água (Figura 41), temperatura (Figura 42) e câmbio (Figura 43). Todas elas, assim como as tarefas alocadas ao processador *P1*, utilizam método de escalonamento não-preemptivo.

É importante ressaltar que a tarefa *lerCambio* precede a mensagem M1 (Figura 44), e esta, por sua vez, precede a tarefa *transmitirResultados* (Figura 44), isso indica que a mensagem M1 segue da tarefa *lerCambio*, alocada ao processador P2 para a tarefa *transmitirResultados*, alocada ao processador P2. As tarefas sendM1 e receberM1 têm o papel de enviar/receber a mensagem M1.

```
/**                          /**
 * @type T                    * @type T
 * @name lerAgua              * @name mapearAgua
 * @processor P2              * @processor P2
 * @scheduling NP             * @scheduling NP
 * @phase 0                   * @phase 0
 * @release 0                 * @release 20000
 * @computing 227             * @computing 241
 * @deadline 20000            * @deadline 40000
 * @period 120000             * @period 120000
 */                           */
void lerAgua (){ ... }       void mapearAgua (){ ... }
```

Figura 41. Especificação das tarefas *TW0* e *TW1*.

```
/**                              /**
 * @type T                        * @type T
 * @name lerTemperatura           * @name mapearTemperatura
 * @processor P2                  * @processor P2
 * @scheduling NP                 * @scheduling NP
 * @phase 0                       * @phase 0
 * @release 20000                 * @release 40000
 * @computing 259                 * @computing 234
 * @deadline 40000                * @deadline 60000
 * @period 120000                 * @period 120000
 */                               */
void lerTemperatura (){ ... }    void mapearTemperatura (){ ... }
```

Figura 42. Especificação das tarefas *TT0* e *TT1*.

```
/**                          /**
 * @type T                    * @type T
 * @name lerCambio           * @name mapearCambio
 * @processor P2             * @processor P2
 * @scheduling NP            * @scheduling NP
 * @phase 0                  * @phase 0
 * @release 40000            * @release 60000
 * @computing 224            * @computing 236
 * @deadline 60000           * @deadline 80000
 * @period 120000            * @period 120000
 */                          * @preceds {M1}
void lerCambio(){ ... }      */
                             void mapearCambio(){ ... }
```

Figura 43. Especificação das tarefas TG_0 e TG_1.

```
/**                                    /**
 * @type T                             * @type M
 * @name transmitirResultados          * @name M1
 * @processor P1                       * @bus B1
 * @scheduling NP                      * @communication 1700
 * @phase 0                            * @preceds {transmitirResultados}
 * @release 80000                      */
 * @computing 2444                     void sendM1 (){ ... }
 * @deadline 120000                    void receiveM1 (){ ... }
 * @period 120000
 */
void transmitirResultados (){ ... }
```

Figura 44. Especificação das tarefas *TRA* e *M1*.

Essas especificações são primeiramente traduzidas em um formato XML (Figura 45) pelo ambiente CML, que internamente executa a ferramenta de análise de restrições temporais de sistemas de tempo real, passando como parâmetro o arquivo XML gerado. Esse processo é similar ao descrito na seção anterior. A diferença está apenas na especificação de cada tarefa, assim como na definição de uma mensagem que realiza comunicação entre duas tarefas alocadas em processadores diferentes. A versão completa da especificação do Sistema de Monitoramento de Veículos, no formato XML, pode ser encontrada no Apêndice B deste trabalho.

```
<realtime-table>
    <task release="0" period="120000" phase="0" processor="P1"
schedulingModel="NP"  oid="1088076" name="lerVelocidade">
        <time>
            <computing value="231"/>
            <deadline value="20000"/>
        </time>
    </task>
    <task release="20000" period="120000" phase="0" processor="P1"
schedulingModel="NP"  oid="131577" name="mapearVelocidade">
        <time>
            <computing value="5487"/>
            <deadline value="40000"/>
        </time>
    </task>
       .
       .
       .
    <message bus="B1" oid="6707027" name="M1">
        <time>
            <comunication value="1700"/>
            <grantBus value="0"/>
        </time>
        <preceds>
            <task-ref name="transmitirResultados"/>
        </preceds>
    </message>
       .
       .
       .
    <task release="60000" period="120000" phase="0" processor="P2"
schedulingModel="NP"  oid="14410104" name="mapearCambio">
        <time>
            <computing value="236"/>
            <deadline value="80000"/>
        </time>
        <preceds>
            <message-ref name="M1"/>
        </preceds>
    </task>
</realtime-table>
```

Figura 45. Especificação do Sistema de Monitoramento de Veículo no formato XML.

A Figura 46 ilustra o diagrama de tempo da escala gerada pelo gerador de escala, indicando que as restrições foram atendidas. Entretanto, se a especificações sofrerem alguma alteração, por exemplo, o período for reduzido para determinada tarefa, o gerador de escalas acusará que não é possível gerar uma escala baseado nessas especificações e o ambiente CML exibirá uma mensagem de erro, tal como na seção anterior (Figura 37).

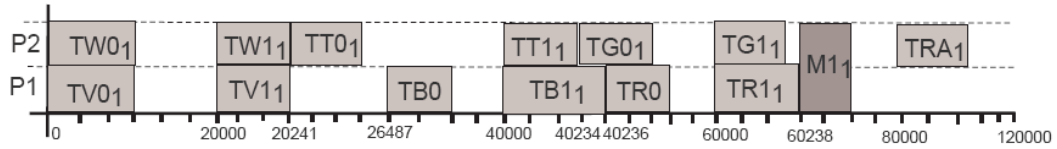

Figura 46. Diagrama de tempo da escala gerada para o Sistema de Monitoramento de Veículos.

4.3 Resumo

Este capítulo demonstrou a utilização da linguagem CML em dois sistemas de tempo real, com intuído de validar a proposta da linguagem. O primeiro sistema é sobre um Semáforo de Trânsito, que consistem em um sistema mono-processado, em que em que cada luz do semáforo (verde, amarelo e vermelho) é modela como uma tarefa, e essas tarefas obedecem a uma ordem de execução.

O segundo é um Sistema de Monitoramento de Veículos, que consiste em um conjunto de sensores utilizados para verificar componentes de um veículo, como acelerador, freios, temperatura do motor, dentre outros. Além disso, é necessário enviar os resultados para um painel externo conectado a esse sistema.

As tarefas especificadas têm que obedecer a uma série de restrições temporais, como tempo de computação/comunicação e *deadline*. Tais restrições são anotadas na própria tarefa, em seguida, o compilador CML as transforma em uma estrutura intermediária na linguagem de programação Java. Essa estrutura é então traduzida em um formato XML, que é o formato utilizado pela ferramenta que analisa as restrições temporais. Esta, por sua vez, informa se é possível ou não gerar uma escala de acordo com as especificações anotadas.

Capítulo 5

Conclusões e Trabalhos Futuros

Requisitos não funcionais é uma preocupação constante de desenvolvedores de software, principalmente para dispositivos com pouco poder de processamento, em que há a necessidade de assegurar certos aspectos da execução do software, em decorrência das limitações do hardware onde ele será executado. Muitas vezes, esses requisitos são inseridos em um documento, servindo até como parte de um contrato entre o contratante e o desenvolvedor do software. Entretanto, existe muita dificuldade de assegurar o cumprimento desses requisitos.

Ultimamente, tem sido comum a pratica de especificação comportamental de programas. Linguagens como JML (*Java Modeling Language*) [3][4], DBC for C (*Design by Contract for C*) [5][2], utilizam anotações no código fonte para especificar comportamentos do programa, escritos na linguagem de programação Java e C, respectivamente. Essas linguagens utilizam a técnica de Projeto por Contrato (*Design by Contract*) [2], utilizado primeiramente pela linguagem de programação Eiffel [8]. Contudo, especificação comportamental de programas desenvolvidos na linguagem C é uma tarefa muito difícil, pois o desenvolvedor tem total liberdade para trabalhar com ponteiros e referência de endereços de memória. Em contrapartida, existe uma grande preocupação com aspectos não funcionais do programa, como desempenho, segurança, consumo de energia, dentre outros.

Este trabalho procurou propor uma linguagem, denominada CML (*C Modeling Language*), para especificar requisitos não funcionais em programas desenvolvidos na linguagem de programação C. Essa linguagem se utiliza de ferramentas externas para realizar análises das restrições impostas pelas especificações.

Este capítulo apresenta as principais contribuições dadas por este trabalho, dificuldades e limitações encontradas, além dos trabalhos futuros.

5.1 Contribuições

Este trabalho teve como objetivo propor uma linguagem de especificação de requisitos não funcionais de aplicações desenvolvidas em C, focando em apenas um tipo de requisito, restrições temporais de sistemas de tempo real. Algumas contribuições são detalhadas a seguir:

Definição e modelagem da linguagem

Definição dos parâmetros utilizados para especificação de sistemas de tempo real. Esses parâmetros foram definidos de acordo com a abordagem proposta por Barreto [6] e Tavares [18], em que a especificação do sistema consiste em um conjunto de tarefas. Cada tarefa possui vários atributos, tais como o método de escalonamento (preemptivo, não-preemptivo), processador ou canal de comunicação onde a tarefa está alocada, o tipo (tarefa de comunicação ou tarefa comum), tempo de execução (ou comunicação) do pior caso, tempo de início da tarefa (*release*), tempo limite de finalização da execução da tarefa (*deadline*), a fase, e relações com outras tarefas (precedência e exclusão). Esses atributos são inseridos no código fonte em forma de anotações, dentro de blocos de comentários.

Construção do compilador da linguagem

O compilador da linguagem foi construído a partir da definição dos atributos necessários para compor a especificação de uma tarefa. A ferramenta JavaCC[21] foi utilizada para a geração do automática do *parser* da linguagem, a partir um arquivo de especificação. Esse arquivo possui um conjunto de *tokens* e a definição da gramática da linguagem.

As especificações expressas em forma de anotações no código fonte são traduzidas primeiramente em uma representação intermediária (estrutura de objetos Java). Uma vez que a linguagem pode ser estendida para lidar com várias ferramentas de análises (uma para cada tipo de requisito não funcional), a representação das especificações em um conjunto de objetos Java facilita a tradução para qualquer outro formato utilizado por determinada ferramenta de análise.

Interação com ferramenta de análise de restrições temporais

A ferramenta utilizada para realizar as análises utiliza como entrada um arquivo de especificações no formato XML. Portanto, a representação intermediária (estrutura de objetos Java) das especificações geradas pelo compilador, é traduzida para um arquivo XML no formato adotado pela ferramenta de análise.

5.2 Dificuldades e Limitações

Como toda ferramenta em desenvolvimento, a ferramenta utilizada para análise de restrições temporais ainda possui alguns problemas. Entretanto, ao tempo que os mesmos foram identificados, os criadores e mantenedores da ferramenta se prontificavam em esclarecer qualquer dúvida e na maioria dos casos, propor uma solução.

Como já foi dito ao longo do texto, a linguagem CML tem uma limitação quanto ao número de requisitos não funcionais suportados por ela, devido à limitação do escopo deste trabalho. Essa limitação pode ser superada nos trabalhos futuros.

5.3 Trabalhos Futuros

Neste trabalho, a linguagem CML foi apresentada suportando apenas um tipo de requisito não funcional, o de restrições temporais em sistemas de tempo real. Entretanto, há uma grande variedade de requisitos não funcionais que não foram incorporados à linguagem CML, como tamanho de código, desempenho, consumo de energia, segurança, entre outros. A linguagem

CML pode ser estendida para que esses tipos de requisitos também possam ser especificados, atendendo a outras necessidades além de restrições temporais para sistemas de tempo real. Isso pode ser feito estendendo a gramática da linguagem e implementando a geração de código apropriada para cada requisito incorporado ao ambiente.

Nesse contexto, a ferramenta PCAF [15] pode ser muito útil, pois trata-se de uma ferramenta muito poderosa para análise de consumo de energia em programas desenvolvidos na linguagem de programação C. Pretendemos estender a linguagem CML para suportar esse tipo de requisito e integrá-la a essa ferramenta.

O código fonte do ambiente CML, e dos sistemas utilizados para validação da proposta (Capítulo 4) estão disponíveis na web através do endereço http://www.dsc.upe.br/~fgaoj/cml/.

Apêndice A

Gramática da linguagem CML no formato JavaCC

```
options {
  JDK_VERSION = "1.5";
}

PARSER_BEGIN(CMLParser)

package br.upe.dsc.cml.compiler;

import java.io.InputStream;

import br.upe.dsc.cml.compiler.entities.*;

public class CMLParser
{
      private static Specification specification = new Specification();
      private AbstractTask currentTask = null;

      public static Specification parse(InputStream aInSt) throws
ParseException {
            // cria um parser
            CMLParser parser = new CMLParser(aInSt);

            // inicializa parser
            parser.init();

            return specification;
      }
}

PARSER_END(CMLParser)

<DEFAULT>
TOKEN : { <START_COMMENT : "/**" >  : IN_COMMENT_BLOCK }

SKIP : { <~[]> }


// quando @parametro eh visto no estado IN_COMMENT_BLOCK, eh um token
```

```
< IN_COMMENT_BLOCK >
TOKEN : { < TASK_TYPE : "@type" >
            | < TASK_NAME : "@name" >
            | < RELEASE : "@release" >
            | < PERIOD : "@period" >
            | < PHASE : "@phase" >
            | < PROCESSOR : "@processor" >
            | < BUS : "@bus" >
            | < SCHEDULING : "@scheduling" >
            | < COMPUTING : "@computing" >
            | < COMMUNICATION : "@communication" >
            | < DEADLINE : "@deadline" >
            | < PRECEDS : "@preceds" >
            | < EXCLUDES : "@excludes" >
            | < COMMUNICATION_VALUE: "M" >
            | < TASK_VALUE: "T" >
            | < SCHEDULING_VALUE: "NP" | "P" >
            | < STRING: <LETTER> (<LETTER> | <DIGIT>)* >
            | < INT: (<DIGIT>)+ >
            | < COMMA: "," >
            | < L_BRAC: "{" >
            | < R_BRAC: "}" >
            |  <#LETTER: ["$","A"-"Z","_","a"-"z"]>
            |  <#DIGIT: ["0"-"9"]>}

< IN_COMMENT_BLOCK >
SKIP :
{
 " "
|  "\t"
|  "\n"
|  "\r"
|  "*"
|  <"//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
|  <"/*" (~["*"])* "*" ("*" | ~["*","/"] (~["*"])* "*")* "/">}


// volta para o estado DEFAULT
< IN_COMMENT_BLOCK >
TOKEN : {  < END_COMMENT: "*/" > : DEFAULT }


// Especificacao da gramatica
void init() : { }
{
  (<START_COMMENT>(statement())+<END_COMMENT>)+
  | <EOF> { }
}

void statement() : { }
{
      taskType()
      | taskName()
      | release()
      | period()
      | phase()
      | bus()
      | processor()
      | scheduling()
      | computing()
```

```
      | communication()
      | deadline()
      | preceds()
      | excludes()

}

void taskType():
{
      AbstractTask task;
}
{
      <TASK_TYPE> (<TASK_VALUE> {task = new Task();}
                  | <COMMUNICATION_VALUE> {task = new
                                                CommunicationTask();})
      {
            specification.setCurrentTask(task);
      }
}

void taskName():
{
      Token value;
}
{
      <TASK_NAME> value=<STRING>
      {
            specification.getCurrentTask().setName(value.toString());
      }
}

void release():
{
      Token value;
}
{
      <RELEASE> value=<INT>
      {
            ((Task) specification.getCurrentTask()).
                  setRelease(Integer.parseInt(value.toString()));
      }
}

void period():
{
      Token value;
}
{
      <PERIOD> value=<INT>
      {
            ((Task) specification.getCurrentTask()).
                  setPeriod(Integer.parseInt(value.toString()));
      }
}
void phase():
{
      Token value;
}
{
      <PHASE> value=<INT>
```

```
	{
		((Task) specification.getCurrentTask()).
				setPhase(Integer.parseInt(value.toString()));
	}
}

void processor():
{
	Token value;
}
{
	<PROCESSOR> value=<STRING>
	{
		((Task) specification.getCurrentTask()).
				setProcessor(value.toString());
	}
}

void scheduling():
{
	Token value;
}
{
	<SCHEDULING> value=<SCHEDULING_VALUE>
	{
		((Task) specification.getCurrentTask()).
				setSchedulingType(value.toString());
	}
}

void computing():
{
	Token value;
}
{
	<COMPUTING> value=<INT>
	{
		((Task)specification.getCurrentTask()).
				setComputing(Integer.parseInt(value.toString()));
	}
}

void deadline():
{
	Token value;
}
{
	<DEADLINE> value=<INT>
	{
		((Task)specification.getCurrentTask()).
				setDeadline(Integer.parseInt(value.toString()));
	}
}

void precedes(): {}
{
	<PRECEDS> <L_BRAC> precedsList() <R_BRAC>
}

void precedsList():
```

```
{
 Token value;
}
{
      precedsElement() (<COMMA> value=<STRING>
      {
            specification.getCurrentTask().getPreceds().add(
            specification.findTaskByName(value.toString()));
      })*
}

void precedsElement():
{
      Token value;
}
{
      value=<STRING>
      {
            specification.getCurrentTask().getPreceds().add(
                  specification.findTaskByName(value.toString()));
      }
}

void excludes(): {}
{
      <EXCLUDES> <L_BRAC> precedsList() <R_BRAC>
}

void excludesList(): {}
{
      excludesElement() (<COMMA> excludesElement())*
}

void excludesElement():
{
      Token value;
}
{
      value=<STRING>
      {
            System.out.println(value.toString());
            specification.getCurrentTask().getExcludes().add(
                  specification.findTaskByName(value.toString()));
      }
}


void bus():
{
      Token value;
}
{
      <BUS> value=<STRING>
      {
            ((CommunicationTask)specification.getCurrentTask()).
                  setBus(value.toString());
      }
}

void communication():
```

```
{
      Token value;
}
{
      <COMMUNICATION> value=<INT>
      {
            ((CommunicationTask)specification.getCurrentTask()).
                  setCommunicationTime(Integer.parseInt(value.toString()));
      }
}
```

Apêndice B

Especificação do Sistema de Monitoramento de Veículos no formato XML

```
<realtime-table>
    <task release="0" period="120000" phase="0" processor="P1"
schedulingModel="NP"  oid="1088076" name="lerVelocidade">
        <time>
            <computing value="231"/>
            <deadline value="20000"/>
        </time>
    </task>
    <task release="20000" period="120000" phase="0" processor="P1"
schedulingModel="NP"  oid="131577" name="mapearVelocidade">
        <time>
            <computing value="5487"/>
            <deadline value="40000"/>
        </time>
    </task>
    <task release="20000" period="120000" phase="0" processor="P1"
schedulingModel="NP"  oid="30377347" name="lerFreios">
        <time>
            <computing value="221"/>
            <deadline value="40000"/>
        </time>
    </task>
    <task release="40000" period="120000" phase="0" processor="P1"
schedulingModel="NP"  oid="21402567" name="mapearFreios">
        <time>
            <computing value="236"/>
            <deadline value="60000"/>
        </time>
    </task>
    <task release="40000" period="120000" phase="0" processor="P1"
schedulingModel="NP"  oid="9443463" name="lerRPM">
        <time>
            <computing value="232"/>
            <deadline value="60000"/>
        </time>
    </task>
```

```
    <task release="60000" period="120000" phase="0" processor="P1"
schedulingModel="NP"  oid="3827495" name="mapearRPM">
        <time>
            <computing value="238"/>
            <deadline value="80000"/>
        </time>
    </task>
    <task release="80000" period="120000" phase="0" processor="P1"
schedulingModel="NP"  oid="4875744" name="transmitirResultados">

        <time>
            <computing value="2444"/>
            <deadline value="120000"/>
        </time>
    </task>
    <message bus="B1" oid="6707027" name="M1">
        <time>
            <comunication value="1700"/>
            <grantBus value="0"/>
        </time>
        <preceds>
            <task-ref name="transmitirResultados"/>
        </preceds>
    </message>
    <task release="0" period="120000" phase="0" processor="P2"
schedulingModel="NP"  oid="15672056" name="lerAgua">
        <time>
            <computing value="227"/>
            <deadline value="20000"/>
        </time>
    </task>
    <task release="20000" period="120000" phase="0" processor="P2"
schedulingModel="NP"  oid="31510384" name="mapearAgua">
        <time>
            <computing value="241"/>
            <deadline value="40000"/>
        </time>
    </task>
    <task release="20000" period="120000" phase="0" processor="P2"
schedulingModel="NP"  oid="5439109" name="lerTemperatura">
        <time>
            <computing value="259"/>
            <deadline value="40000"/>
        </time>
    </task>
    <task release="40000" period="120000" phase="0" processor="P2"
schedulingModel="NP"  oid="12848256" name="mapearTemperatura">
        <time>
            <computing value="234"/>
            <deadline value="60000"/>
        </time>
    </task>
    <task release="40000" period="120000" phase="0" processor="P2"
schedulingModel="NP"  oid="17881115" name="lerCambio">
        <time>
            <computing value="224"/>
            <deadline value="60000"/>
        </time>
    </task>
    <task release="60000" period="120000" phase="0" processor="P2"
schedulingModel="NP"  oid="14410104" name="mapearCambio">
        <time>
            <computing value="236"/>
```

```
            <deadline value="80000"/>
        </time>
        <preceds>
            <message-ref name="M1"/>
        </preceds>
    </task>
</realtime-table>
```

Bibliografia

[1] MOURA, A. *Especificações em Z: Uma introdução*. Editora da Unicamp, Campinas, Março 2001.

[2] MEYER, B. et al. Design *by Contract: The Lessons of Ariane*. IEEE Computer, 30(2):129–130, Janeiro 1997.

[3] VERZULLI, Joe. *Getting started with JML: Improve your Java programs with JML annotation*. IBM DeveloperWorks article, Março 2003.

[4] LEAVENS, G. e CLIFTON, C. *Lessons from the JML Project*. Verified Software: Theories, Tools, Experiments. Department of Computer Science, Iowa State University, TR #05-12a, Abril de 2005.

[5] DBC for C (*Design by Contract for C*). Disponível em <http://www.onlamp.com/pub/a/onlamp/2004/10/28/design_by_contract_in_c.html>. Último acesso em 01/09/2006.

[6] BARRETO, R. *A Time Petri Net-Based Methodology for Embedded Hard Real-Time Systems Software Synthesis*. Tese de Doutorado, Centro de Informática. Universidade Federal de Pernambuco, Abril 2005.

[7] MACIEL, Paulo Romero Martins. *Introdução às Redes de Petri e Aplicações*. X Escola de Computação. Campinas-SP, 1996.

[8] MEYER, Bertrand. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[9] RUMBAUGH, J.*UML Guia do Usuario*, Editora Campus.

[10] Junit. Disponível em <http://www.junit.org>. Último acesso em 01/09/2006.

[11] Javadoc Tool. Disponível em <http://java.sun.com/j2se/javadoc/>. Último acesso em 14/09/2006.

[12] ESC/Java 2.<http://secure.ucd.ie/products/opensource/ESCJava2/>. Última acesso em 01/09/2006.

[13] Compaq Research Center. Disponível em <http://research.compaq.com>. Último acesso em 14/11/2006.

[14] Ruby. Disponível em <http://www.ruby-lang.org>. Visitado pela última vez em 01/09/2006.

[15] OLIVEIRA JÚNIOR, M. N. *Analyzing Software Performance and Energy Consumption of Embedded Systems by Probabilistic Modeling: An Approach Based on Coloured Petri Nets*. 27th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (Petri Nets 2006). Turku, Finlândia, Junho de 2006.

[16] CPN-Tools, versão 1.4.0. Disponível em <http://wiki.daimi.au.dk/cpntools/cpntools.wiki>. Último acesso em 01/09/2006.

[17] SINGHAL, A. *Real time systems: A survey*. Relatório Técnico, Computer Science Department. University of Rochester, Dezembro de 1996.

[18] TAVARES, E. A. G. *A Time Petri Net Based Aproach for Software Synthesis in Hard Real-Time Embedded Systems with Multiple Processors*. Dissertação de Mestrado, Centro de Informática. Universidade Federal de Pernambuco, Março de 2006.

[19] KOPETZ, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
[20] LINS, R. D. *Compiladores Modernos*. Editora Campus. Rio de Janeiro, 2001.
[21] JavaCC (*Java Compiler Compiler*). Disponível em <http://javacc.dev.java.net/>. Último acesso em 07/11/2006.
[22] Sun Microsystems. Disponível em <http://www.sun.com>. Último acesso em 07/11/2006.
[23] Lex. Disponível em <http://plan9.bell-labs.com/magic/man2html/1/lex>. Último acesso em 07/11/2006.
[24] Yaac (*Yet Another Compiler-Compiler*). Disponível em <http://dinosaur.compilertools.net/yacc/index.html>. Último acesso em 07/11/2006.
[25] JFlex. Disponíevl em <http://jflex.de/>. Último acesso em 07/10/2006.
[26] CUP (*Constructor of Useful Parsers*). Disponível em <http://www2.cs.tum.edu/projects/cup/>. Último acesso em 15/11/2006.
[27] ARCOVERDE JR., A., ALVES, G., LIMA, R. M. F., MACIEL, P. R. M., OLIVEIRA JR, M., BARRETO, R. *EZPetri: A Petri net interchange framework for Eclipse based on PNML*. In: 1st International Symposium on Leveraging Applications of Formal Method - ISOLA, Nicosia, 2004.
[28] ExpectJ. Disponível em <http://expectj.sourceforge.net/>. Último acesso em 17/11/2006.
[29] Xerces. Disponível em <http://xml.apache.org>. Ultimo acesso em 14/10/2006.
[30] DOM (*Document Object Model*). Disponível em < http://www.w3.org/DOM/> . Último acesso em 14/10/2006.
[31] JDOM (*Java Document Object Model*). Disponível em <http://www.jdom.org>. Último acesso em 14/10/2006.
[32] STEWART, J. W., MIAO, K. *The 8051 Microcontroller*. Prentice Hall. 2ª Edição. 1999.