UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

VERIFICAÇÃO DE MODELOS UML DE SOFTWARE EMBARCADO COM *MODEL CHECKING*

MARCELO MONTEIRO CUSTÓDIO

MANAUS
2009

UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

MARCELO MONTEIRO CUSTÓDIO

VERIFICAÇÃO DE MODELOS UML DE SOFTWARE EMBARCADO COM *MODEL CHECKING*

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Amazonas, como requisito parcial para a obtenção do título de Mestre em Informática, área de concentração Engenharia da Computação.

Orientador: Prof. Dr. Raimundo da Silva Barreto

MANAUS
2009

MARCELO MONTEIRO CUSTÓDIO

VERIFICAÇÃO DE MODELOS UML DE SOFTWARE EMBARCADO COM *MODEL CHECKING*

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Amazonas, como requisito parcial para a obtenção do título de Mestre em Informática, área de concentração Engenharia da Computação.

Aprovado em 15 de Dezembro de 2008.

BANCA EXAMINADORA

Prof. Dr. Raimundo da Silva Barreto, Membro
Universidade Federal do Amazonas - Ufam

Prof. Dr. José Reginaldo Hughes Carvalho, Membro
Universidade Federal do Amazonas - Ufam

Prof. Dr. Vicente Ferreira de Lucena Júnior, Membro
Universidade Federal do Amazonas - Ufam

Prof. Dr. Edward David Moreno Ordonez, Convidado
Escola Superior de Tecnologia - EST, UEA

Agradecimentos

Este trabalho não teria sido concretizado sem a grande e muitas vezes crucial ajuda de várias pessoas. Mas antes de qualquer dessas pessoas veio a ajuda do Senhor Jesus que, mesmo através da ajuda de terceiros, manifestou seu grande poder e bondade para comigo. Portanto, a Ele toda a Glória pela conclusão deste trabalho! Deus me ergueu do monturo e de situações muito difíceis em minha vida e me permitiu iniciar os trabalhos do Mestrado. Durante o mesmo, ocorreram muitas adversidades. De todos os tipos. À semelhança do que aconteceu com o salmista, minha alma quase desfaleceu em vários momentos. Mas Deus me deu forças! E não somente isso, mas também me ensinou e corrigiu bastante. Enfim, obrigado, Senhor!

Minha família desempenhou um papel fundamental nesse período. Minha mãe, uma pessoa como não se vê por ai, uma pessoa feita de amor. Sempre alegre e estimuladora mas também cobrando nos momentos em que deveria fazê-lo. Obrigado, mãe, por nunca reclamar mas sempre agradecer. Meu irmão Rodrigo, um homem amoroso e paciente. Foi, por mais de um ano depois que iniciei o Mestrado, o arrimo de nossa família, sacrificando muitas de suas vontades e me pertindo contribuir em casa apenas com um valor irrisório, parte da bolsa que recebia de início. Minha irmã Izabela (Mel) foi a moça que mais nos deu alegria e exemplos e para quem fazemos tudo. Eu e essa família especial superamos vários obstáculos ao longo dos anos e crescemos em todos os aspectos, tendo inclusive prosperado financeiramente. Deus é Fiel!

Meus agradecimentos também à Coordenação do Programa de Pós-graduação em Informática do Departamento de Ciência da Computação da UFAM e ao prof. Dr. Raimundo da Silva Bar-

reto, meu orientador, por ter me aceitado no programa para assim contribuir com o estado da arte da Computação e com a sociedade brasileira também. Agradeço também aos comentários feitos pela banca da defesa, composta pelo prof. Barreto e pelos professores Dr. José Reginaldo Hughes Carvalho (UFAM), Dr. Vicente Ferreira de Lucena Júnior (UFAM) e Dr. Edward David Moreno Ordonez (UEA). Agradeço à Diretoria Administrativa do Ministério Público do Estado do Amazonas por ter ajustado meu turno de trabalho pra eu poder cursar a disciplina PAA, na época em que lá trabalhava e ao CNPq por mais de um ano de bolsa que financiou meus estudos no Mestrado.

Amigos são parte fundamental de nossas vidas. E esse é o momento de seu reconhecimento! Agradeço ao Mahatama Porto, um dos melhores amigos que já tive, pela paciência e voluntariedade comigo. Também a sua esposa Dani Porto, por ter sido prestativa em vários momentos. Ao Gilberto "GIBA"dos Santos pelos fontes LaTeX, pelos bons momentos nas "peladas"na FPF e principalmente pelas nossas conversas edificantes e descontraídas. Ao Saulo Jorge, pelos bons momentos de compartilhar a Fé. Agradeço também ao João Luis (good job mate!), Luis Evangelista, Kaio Rafael, Lucas Cordeiro, Ícaro de Oliveira, Michel Souza, Jackson Gervásio, Ênio Barbosa, Prof. Leandro Galvão, Jucimar Jr. e Rafael Soares. Agradeço também ao professor Ruiter Caldas pelo apoio no artigo do WTR 2007.

Por fim, ao Pr. Aurino Bandeira e Igreja Assembléia de Deus, Templo Central, pelo suporte espiritual e financeiro no momento em que precisei me ausentar de Manaus por ocasião do RTSS 2006. E ao amigo e irmão em Cristo, Paulo e esposa por terem me recebido com extrema atenção no Rio de Janeiro por ocasião do referido simpósio e me mostrado novamente (e com muito orgulho, diga-se de passagem) a Cidade Maravilhosa.

Enfim, a todos os que acreditaram em Deus e no meu trabalho e que me ajudaram direta ou indiretamente! A vocês o meu muito obrigado!

Minha é a prata, e meu é o ouro, disse o SENHOR dos Exércitos.

A glória desta última casa será maior do que a da primeira,

diz o SENHOR dos Exércitos, e neste lugar darei a paz,

diz o SENHOR dos Exércitos.

Ageu 2:8-9

Resumo

Os sistemas embarcados possuem inegável importância na sociedade atual. Eles possuem restrições temporais (quando são de tempo real), de gerência de consumo de energia, tamanho, peso etc que tornam o seu projeto e concepção mais complexos do que os sistemas convencionais. Dado o grande número de requisitos de todos os tipos, a alta complexidade dos softwares embarcados desenvolvidos bem como a grande possibilidade de catástrofes significativas em caso de falha e por fim a grande pressão de mercado por produtos cada vez mais rápido, fazem-se necessários métodos que possam assegurar uma correta, rápida porém intuitiva especificação e concepção dos projetos. Diante disso, o presente trabalho visa prover um método que acrescente ao atual estado da arte.

O objetivo do método então é prover uma abordagem que colete uma especificação de software embarcado em uma notação semi-formal, orientada a objetos e amplamente aceita pela Indústria, que é a Unified Modeling Language (UML), especificamente com seu Diagrama de Sequência, o qual é apto para capturar os aspectos dinâmicos de um sistema e um mecanismo de tradução dessa notação para a notação formal SMV, apta a ser utilizada pelo *model checker* de mesmo nome.

O objetivo do método é prover também um esquema de tradução dos diagramas de sequência em UML para uma notação formal, no caso a notação de Redes de Petri, o qual é adequada para verificação formal, gerando saídas de arquivos nos formatos APNN e PNML. O formato APNN é adequado para ser usado no Model Checking Kit (MCK).

Por fim, prover um esquema de tradução consultas de propriedade em alto nível para o formato de CTL puro adequado para ser usado no MCK e um programa em SMV e sua especificação

em CTL, formatos aptos a serem usados no model checker SMV. A verificação de propriedades é apenas qualitativa, isto é, que verificará apenas propriedades de execução do software embarcado, em oposição às propriedades quantitativas de tempo por exemplo, comuns em softwares de tempo-real. Todas essas funcionalidades são realizadas por uma ferramenta, chamada Ambiente de Verificação Formal de Software Embarcado.

Palavras-Chave: UML, Redes de Petri, Sistemas Embarcados, Métodos Formais.

Abstract

Embedded systems have undeniable relevance in modern society. They have temporal constraints (as long as they are real time ones), power consumption management, size, weight, etc which make their design more complex than the design of their desktop peers. Given the huge number of requirements of all kinds, the high complexity of embedded software as well as the big possibilities of critical damages in case of flaws and, at last, the even bigger pressure of market for new products faster, it make necessary methods which can assure correct, fast but intuitive specification and conception of designs. Considering this, this work aims to provide a method which contribute to the state of art.

The goal of the proposed method is to provide an approach which gather an specification of an embedded software in a semi-formal, object-oriented and Industry-accepted notation, which is Unified Modeling (UML), specifically their Sequence Diagram notation which is able to capture dynamic aspects of a system and a mecanism of translation of this notation into a formal one, called SMV, apropriate for being used by the SMV model checker.

The goal of the method is also provide an translation scheme of the sequence diagrams into another formal notation, the so called Petri Nets notations. Petri Net notation is well suited to formal verification. Finally, the goal of the method is to provide a mechanism of translation of high level properties queries into formal notation CTL. Property queries are only qualitative. All these functionalities are implemented in a tool called Ambiente de Verificação Formal de Software Embarcado.

Keywords: UML, Petri Nets, Embedded Systems, Formal Methods.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

> We must accept finite disappointment, but we must never lose infinite hope.
>
> Martin Luther King, Jr.

Os sistemas embarcados possuem inegável importância na sociedade atual. Eles proporcionaram grande melhoria de qualidade da vida moderna com aplicações como freios ABS, cirurgia com robôs, injenção eletrônica de combustível, casas inteligentes, sistemas de navegação de veículos, assistentes pessoais digitais (*Personal Digital Assistants* - PDAs) e aparelhos celulares, fornos micro-ondas, computadores vestíveis etc e promoveram um igualmente grande impulso econômico. Suas maiores características são a ubiquidade e pervasividade.

Este capítulo abre o presente trabalho e contextualiza o leitor com o universo dos sistemas embarcados. É também o propósito trazer as motivações e objetivos do presente trabalho e, ao final, a organização de todo o trabalho de dissertação.

1.1 Contexto

O presente trabalho se dá no contexto de sistemas embarcados. Os sistemas embarcados diferem dos computadores de mesa em vários aspectos. Eles são projetados para executar *tarefas específicas* e podem possuir restrições de gerência de consumo de energia, tamanho, peso, memória de dados e de tempo que certamente tornam o seu projeto e concepção mais complexos do que o dos sistemas convencionais. Um ponto importante é que algumas classes de sistemas embarcados podem pôr vidas ou funções de negócio críticas em risco. Sendo assim, estes sistemas devem ser tratados diferentemente do caso onde somente o custo da falha é o investimento do projeto.

As pessoas têm construído sistemas embarcados há décadas. Os primeiros microprocessadores eram tão limitados que suas funções primárias eram apenas gerenciar dispositivos de entrada e de saída. Acompanhar o desempenho dos mesmos requeria mais arte do que ciência propriamente dita.

O fenômeno no avanço na área de sistemas embarcados se deu basicamente devido aos avanços da tecnologia de sistemas micro-processados, bem como sua redução de preço. Tais avanços iniciaram-se na década de 1980 e resultaram em novas concepções de produtos. Os micro-controladores se tornaram mais baratos, menores e mais confiáveis. Com o tempo surgiram também novas demandas de aplicações embarcadas, exigindo, portanto, flexibilidade de configuração e manutenção. E tudo isso feito em um tempo cada vez menor. O software veio como a solução para esses problemas. A desvantagem primária de mover as funcionalidades para software é que elas executam reconhecidamente menos rápido do que as equivalentes em hardware. A flexibilidade de mudanças no software faz com que seja possível mover mais funcionalidades de hardware para software. Certa análise de mercado mostra que mais de 80% das funcionalidades do produto é implementada em software (SANGIOVANNI-VINCENTELLI, 2001).

1.2 Motivação

Como se percebe, os sistemas embarcados penetraram em todos os níveis e tipos de aplicações. Dado o grande número de requisitos de todos os tipos, a alta complexidade dos softwares embarcados desenvolvidos bem como a grande possibilidade de catástrofes significativas em caso de falha e, por fim, a grande pressão de mercado pela entrega cada vez mais rápida de produtos, fazem-se necessários métodos que possam assegurar uma correta e rápida especificação e concepção dos projetos.

Ou seja, tanto a característica de intuitividade no processo de desenvolvimento do produto quanto a corretude do software final desenvolvido são mandatórias. Sobre a corretude do software, métodos formais têm seu valor comprovado pela Academia e, cada vez mais, pela Indústria.

1.3 Objetivos

O objetivo principal deste trabalho é a definição de um método para, no que contexto de software embarcado: (i) Rápida e intuitiva modelagem por meio de notação semi-formal; (ii) Tradução automática deste modelo semi-formal para um modelo formal; (iii) Correta verificação do modelo formal recém-obtido através de métodos formais e (iv) uma proposta de especificação de consultas de propriedades de sistemas em linguagem natural com sua posterior tradução automática para uma notação formal, a qual pode ser usada na etapa anteriormente citada de verificação formal.

No que diz respeito à rápida intuitiva modelagem do software, o método visa prover uma abordagem que colete uma especificação de software embarcado em uma notação semi-formal, orientada a objetos e amplamente aceita pela Indústria e Academia, a Unified Modeling Language (UML), especificamente com seu Diagrama de Sequência, o qual é apto para capturar os aspectos dinâmicos de um sistema. A notação de Diagrama de Sequência foi escolhida por ser usada em menos projetos de software embarcado do que a tradicional *Statechart* ou Máquina de Estados, na UML 2.0, até o momento da escrita deste trabalho, e, por isso, constituir-se como um desafio.

A fim de realizar a segunda parte do objetivo, o de verificação formal do software embarcado, é também objetivo do método proposto neste trabalho prover um esquema de tradução automática dos diagramas de sequência UML para a notação formal SMV , apta a ser usada na ferramenta verificadora de modelos, ou *model checker*, de igual nome, detalhada a seguir.

Já no que diz respeito à verificação formal, o presente trabalho faz uso da verificação de propriedades através da técnica de Verificação de Modelos ou *Model Checking*. Não faz parte do escopo do presente trabalho implementar uma ferramenta de verificação de modelos ou *model checker*. Para esse propósito, é usado o *model checker* Symbolic Model Checker (SMV). Essa atividade toma como entrada o diagrama de sequência, já na notação formal SMV e então procede à verificação.

Por fim, em se tratando da proposta de especificação de consultas de propriedades de sistemas em linguagem natural com sua posterior tradução automática para uma notação formal, o presente método prevê uma equivalência entre termos na linguagem natural e termos de consulta em CTL.

Como objetivo secundário, o presente trabalho visa desenvolver uma ferramenta de software, de nome *Ambiente de Verificação Formal de Software Embarcado* (AVFSE), que realize todos os quatro objetivos principais descritos até aqui. Adicionalmente à tradução do diagrama de sequência para o formato SMV, o AVFSE também traduzirá o diagrama de sequência para a notação formal de rede de Petri e a sua posterior escrita em arquivo nos formatos APNN e PNML. O formato APNN é adequado para ser usado no *model checker* Model Checking Kit (MCK) (MCK, 2009). Já o formato PNML é amplamente aceito pela comunidade acadêmica e tem sido cada vez mais usado também por ser baseado em XML, um padrão da Indústria pra comunicação de dados e troca de informações entre ferramentas de software.

A ferramenta AVFSE proverá uma interface amigável para coleta de requisitos ou consultas de propriedade em alto nível com a consequente tradução automática das mesmas para o formato da Lógica da Árvore de Computação, do Inglês *Computation Tree Logics* (CTL), adequado para ser usado nos *model checkers* SMV e MCK. A verificação de propriedades é apenas qualitativa, isto

é, verifica apenas propriedades de execução do software embarcado, em oposição às propriedades quantitativas, por exemplo, de tempo, comuns em softwares de tempo-real. A seguir, um sumário dos principais pontos e propostas deste trabalho.

1.4 Sumário dos Principais Pontos

A seguir, a lista das atividades do método proposto pelo presente trabalho e da ferramenta que o realiza baseados no exposto na Seção 1.3:

- Abordagem de modelagem inicial do software embarcado em linguagem semi-formal, orientada a objetos e padrão da Indústria e Academia, no caso, a UML.
- Tradução automática desta modelagem para modelo numa notação formal, no o formato SMV.
- Verificação desse modelo via *Model Checking*, com a ferramenta SMV, do acrônimo, em Inglês, Symbolic Model Checker.
- Escrita em arquivo da Rede de Petri gerada nos formatos APNN e no cada vez mais usado PNML.
- Interface amigável para coleta das consultas de propriedade em linguagem natural.
- Tradução automática dessas consultas para CTL, do acrônimo, em Inglês, Computation-Tree Logics.

1.5 Organização da Dissertação

O presente trabalho está organizado como segue: O Capítulo 2 descreve os trabalhos relacionados com as metodologias verificação formal de modelos em UML e as de tradução de diagramas UML

para alguma variação da notação de Redes de Petri a fim de verificar formalmente sistemas embarcados. Ele também descreve abordagens alternativas a essa tradução para verificação formal de software embarcado.

A seguir, o Capítulo 3 fornece os principais conceitos e teorias necessários para o entendimento deste trabalho. Inicialmente, o Capítulo apresenta os fundamentos de software embarcado, o qual é o tema desta dissertação. Na sequência, ele traz as principais atividades necessárias à atividade de engenharia de software embarcada e a notação semi-formal da *Unified Modeling Language* (UML), em especial de seu Diagrama de Sequência. A próxima Seção detalha a notação formal das Redes de Petri. E a última Seção apresenta as noções de Métodos Formais, incluindo a notação da *Computation Tree Logics* (CTL) e a técnica de verificação de modelos *Model Checking*.

O próximo Capítulo, o Capítulo 4 descreve o método de verificação formal de software embarcado proposto neste trabalho. O Capítulo 5 apresenta um experimento a fim de comprovar fundamentos e pontos mais relevantes do trabalho. Ele apresenta um experimento com uma aplicação simples de controle sobre a qual são realizadas várias consultas de propriedades do sistema.

Finalmente, o capítulo 6 resume os resultados, tece as considerações finais, que incluem as conclusões do trabalho, os pontos relevantes do mesmo e as direções para trabalhos futuros.

Vários trabalhos têm sido realizados no sentido de permitir desenvolvimento de software embarcado de uma maneira formalmente rigorosa. O próximo Capítulo trata dos principais trabalhos relacionados ao atual, citando seus pontos fortes e os aspectos similares.

Capítulo 2

Trabalhos Correlatos

2.1 Jeng

O primeiro trabalho correlato é o de Jeng (JENG, 2002). O presente trabalho utiliza algumas regras de tradução de UML para Rede de Petri próximas às regras de tradução usadas no trabalho de Jeng. Naquele trabalho, tomava-se como entrada um diagrama de sequência da UML estendido e, através de composição de blocos, chegava-se à Rede de Petri Lugar/Transição final. O diagrama de sequência estendido é aumentado de alguns elementos descritivos que visam a aumentar o poder expressivo da notação permitindo ou ressaltando a presença de construções tais como sincronismo, confluência, escolha etc.

No trabalho de Jeng, a abordagem de tradução foi usada para provar a eficiência de um algoritmo de *slicing* para escalonamento de tarefas e, além disso, nenhuma verificação formal foi realizada. A modelagem do sistema é realizada com o diagrama de sequência, aumentado das extensões introduzidas no trabalho de Jeng (JENG, 2002). A fim de melhor descrever o comportamento do sistema, Jeng estendeu o diagrama de sequência com o poder da concorrência, escolha, sincronismo e confluência. A Figura 2.1 mostra um exemplo de diagrama de sequência aumentado da Notação Diagrama de Sequência Estendida

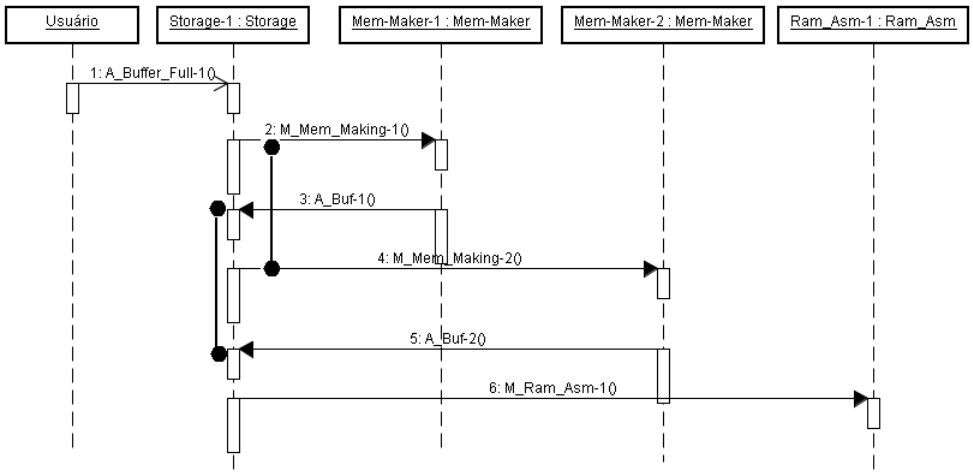

Figura 2.1: Notação Diagrama de Sequência Estendida

O processo de tradução introduzido no trabalho de Jeng (JENG, 2002) consiste em oito regras. É válido apresentar, nas próximas Subseções, um resumo daquelas que possuem similaridades com as regras de tradução apresentadas neste trabalho.

2.1.1 Regra de Conversão de Envio de Mensagem

O envio de mensagem é transformado em um lugar como mostra a Figura 2.2, onde *msg* é um atributo de *Object2*. O número dentro do colchete seguindo o asterisco denota a marca de iteração e é convertido em um peso de arco.

2.1.2 Regra de Conversão de Concorrência

Como mostrado na Figura 2.3, o intervalo rotulado por uma linha sólida com extremidades sólidas também é chamado de grupo de concorrência.

2.1.3 Regra de Conversão de Escolha

A Figura 2.4 mostra o grupo de escolha, o qual é denotado por um intervalo rotulado com uma linha pontilhada cujas extremidades são vazias. Somente uma mensagem neste grupo pode ser

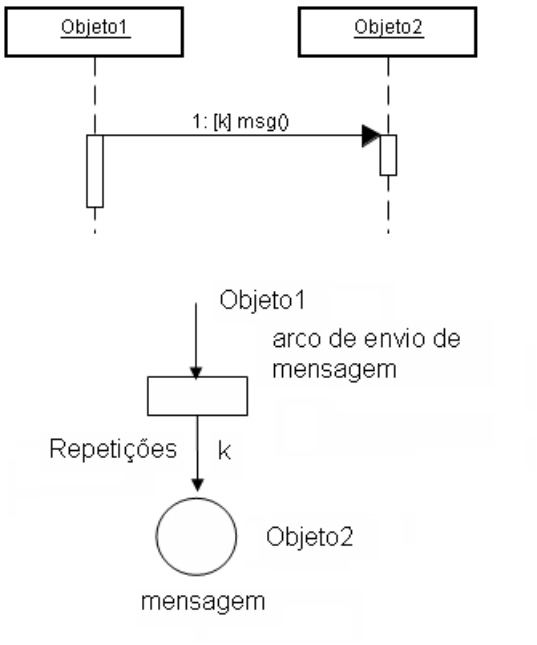

Figura 2.2: Regra de Conversão de Envio de Mensagem.

disparada por vez.

2.1.4 Regra de Conversão de Sincronismo

A Figura 2.5 mostra um grupo de sincronismo, o qual é denotado por um intervalo rotulado por uma linha sólida vertical em suas setas. Após todas as mensagens no grupo de sincronismo finalizarem suas execuções, o próximo passo pode ser executado. No caso da Figura 2.5, o próximo passo é a mensagem *method3 ()*.

2.1.5 Regra de Conversão de Confluência

Como mostra a Figura 2.6, o intervalo rotulado pela linha pontilhada vertical nas setas é chamado grupo de confluência. Diferente do grupo de sincronismo, o grupo de confluência permite que somente uma mensagem do grupo que finalize sua execução possa disparar a execução da próxima operação, no caso *method3 ()*.

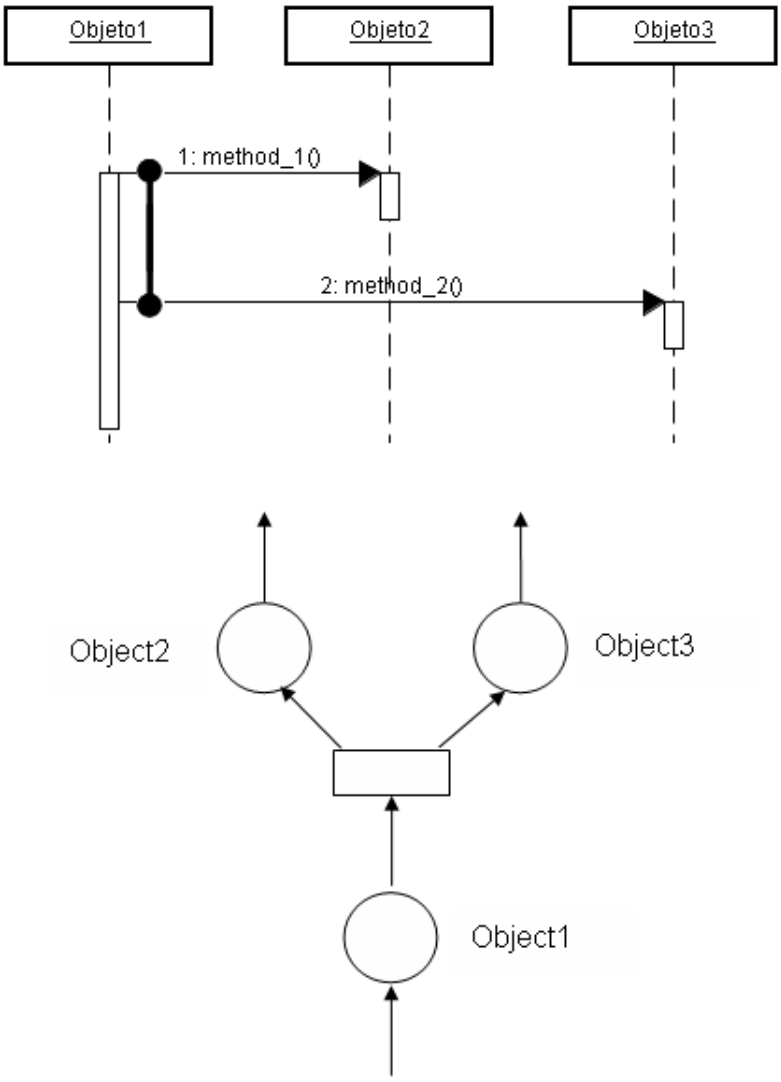

Figura 2.3: Regra de Conversão de Concorrência.

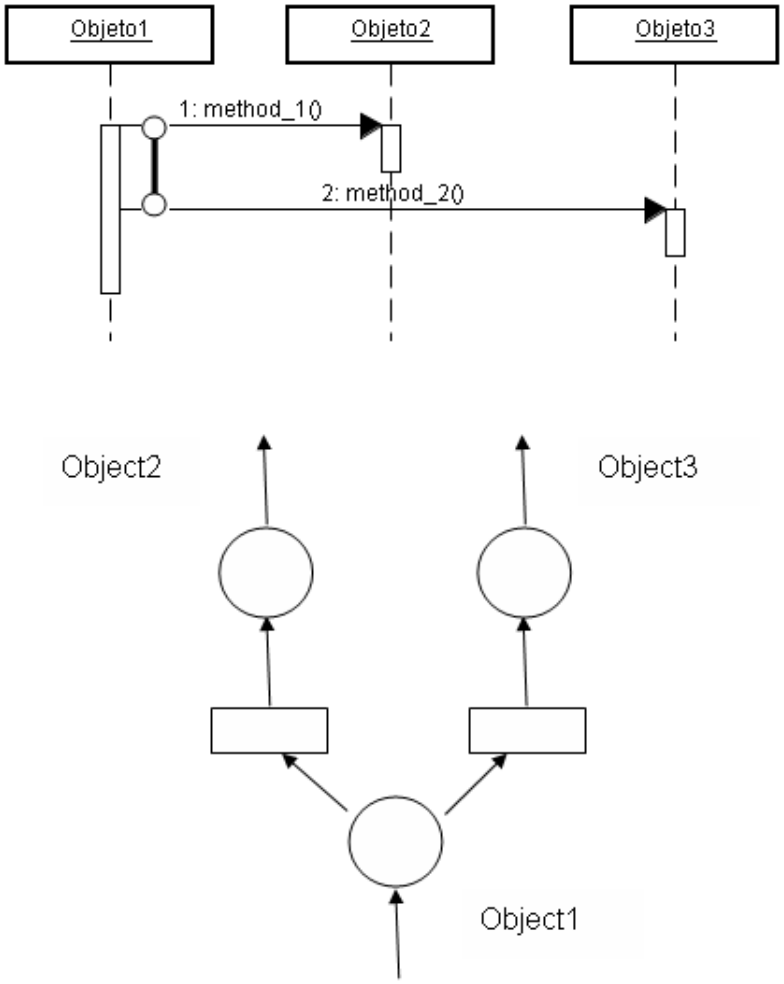

Figura 2.4: Regra de Conversão de Escolha.

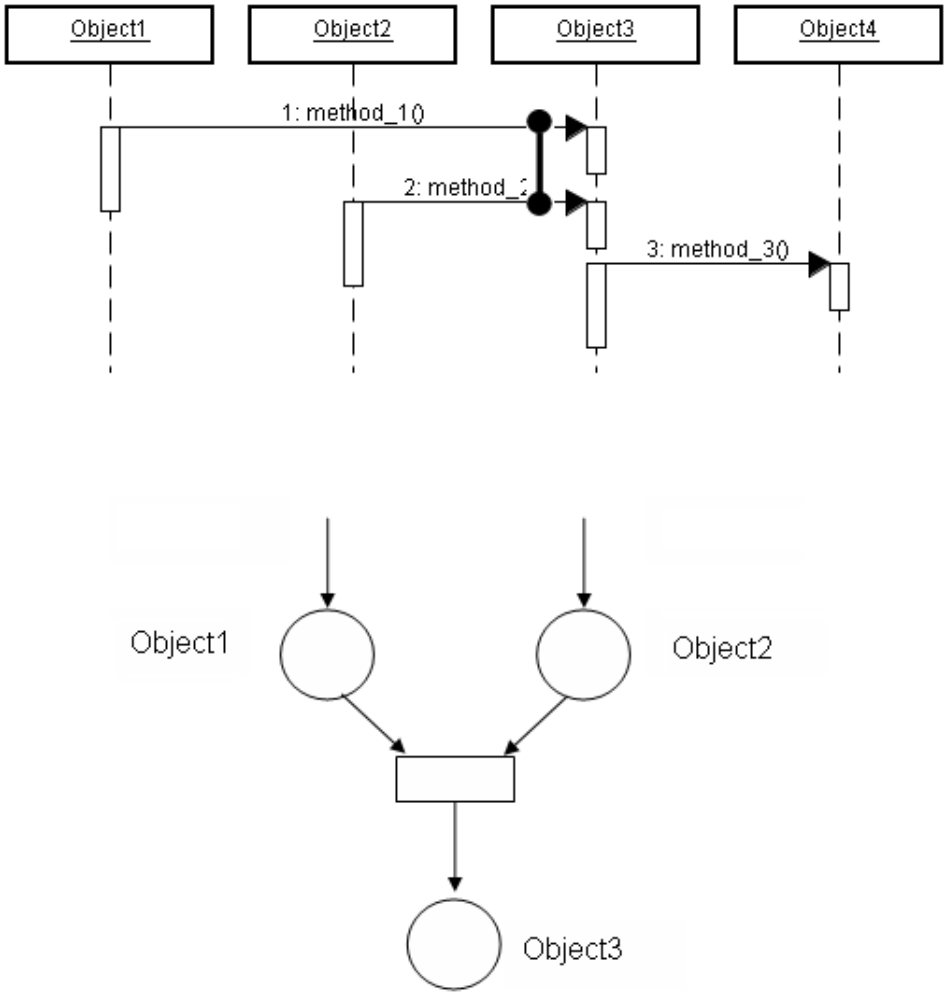

Figura 2.5: Regra de Conversão de Sincronismo.

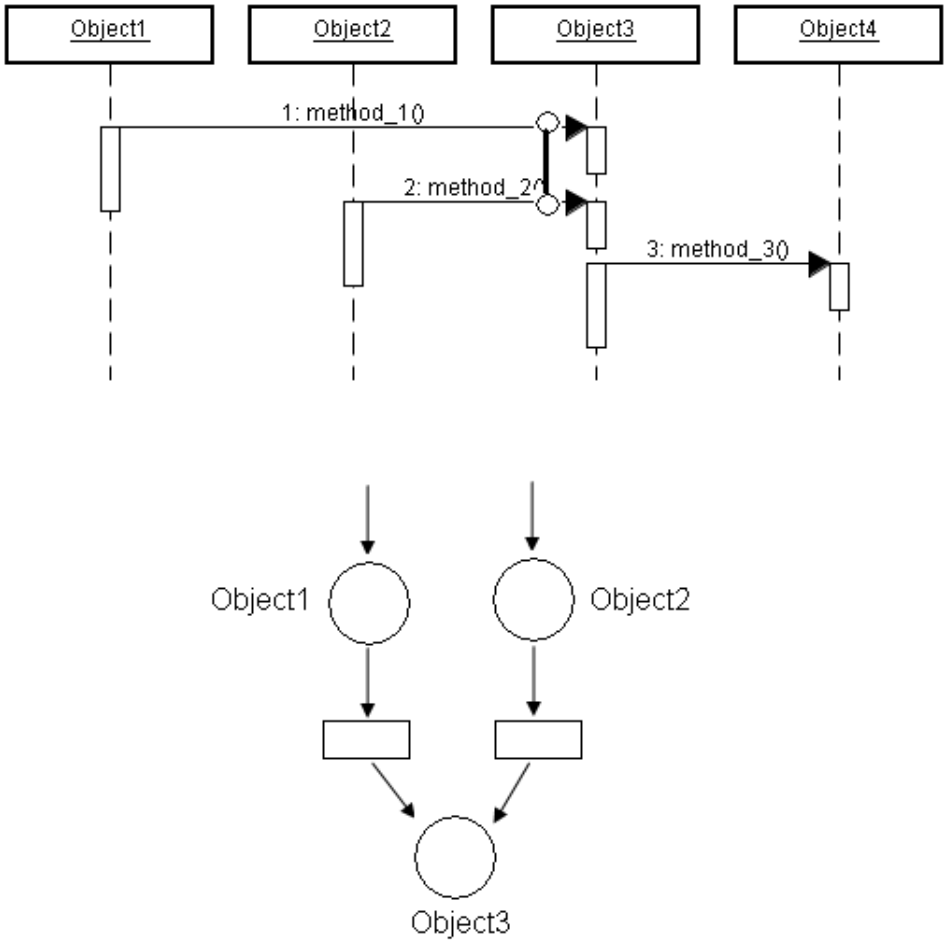

Figura 2.6: Regra de Conversão de Confluência.

2.2 Tribastone e Gilmore

No trabalho de Tribastone e de Gilmore (TRIBASTONE, 2008), é apresentado um procedimento automático para derivar-se modelos de álgebra de processo na ferramenta PEPA a partir de diagramas de sequência para conduzirem-se avaliações quantitativas. A ferramenta PEPA recentemente foi enriquecida com uma semântica de fluxo fluído facilitando a análise de modelos de escala e complexidades até superiores ao da Análise Markoviana. Nesse trabalho, o uso de PEPA no processo de engenharia de software, evita também o problema da explosão de estados.

2.3 Shen, Virani e Niu

O trabalho de Shen (SHEN, 2008) para verificação formal UML baseou-se no fato de que diagramas de sequência da UML 2.0 introduziram muitas novas estruturas de controle, tais como *CombinedFragments*, para expressar trocas de mensagem concorrentes. Estas novas características tornam a UML 2.0 mais expressiva do que a UML 1, entretanto, a ausência de descrições com semântica formal torna difícil para desenvolvedores de software e construtores de ferramentas desenhar e analisar diagramas de sequência. No trabalho de Shen, entretanto, é adaptado um padrão semântico para formalizar-se as construções de controle estruturadas de diagramas de sequência.

2.4 Mokhati, Gagnon e Badri

O trabalho de Mokhati (MOKHATI, 2007) apresenta um framework que suporta verificação formal de diagramas UML usando a linguagem Maude (MOKHATI, 2007). A abordagem considera características tanto estáticas quanto dinâmicas de sistemas orientados a objeto. O foco consiste nos diagramas de classe, estado e de comunicação. A linguagem formal e orientada a objetos Maude, basead em lógica de reescrita, suporta especificação formal e programação de sistemas concorrentes, bem como *model checking*. As maiores motivações para esse trabalho são: (1) amar-

rar a notação UML e a linguagem Maude, (2) preservar a coerência em descrições de sistemas orientados a objeto e (3) usar técnicas de *model checking* para formalmente suportar seu processo de verificação. As especificações em Maude geradas a partir de diagramas UML considerados são válidas por simulação e *model checking*. A abordagem é ilustrada usando-se um caso concreto.

2.5 Boutekkouk e Benmohammed

Uma abordagem aplicada à modelagem sistemas embarcados orientados a controle/dados é o trabalho de Boutekkouk (BOUTEKKOUK, 2009). Nele, é apresentado uma aplicação consistindo de uma rede de tarefas orientadas a dados e a controle que se comunicam via canais abstratos. Uma plataforma de hardware é modelada como um diagrama de estrutura da UML. O mapeamento da aplicação numa plataforma de hardware é modelada através de restrições UML. A partir de modelos UML, uma especificação na linguagem Maude é gerada. Usa-se esta especificação formal para formalmente validar-se as funcionalidades do sistema contra propriedades não desejadas e para se estimar o consumo de energia num alto nivel de abstração.

2.6 Broadfoot e Hopcroft

Segundo Broadfoot e Hopcroft (BROADFOOT, 2005), um dos maiores empecilhos para uma adoção mais estensiva de métodos formais é o hiato entre os requisitos capturados em uma notação informal e uma especificação formal e como se dá a tradução do primeiro para o último.

O trabalho de Broadfoot e Hopcroft (BROADFOOT, 2005) resolve essa barreira combinando dois métodos formais complementares, quais sejam: Engenharia de Software Cleanroom (ou "Sala Limpa") (CLEANROOM SOFTWARE ENGINEERING, 2009) e o arcabouço CSP/FDR (HOARE, 1985).

O Cleanroom provê os meios de se traduzir a especificação de requisitos informal em uma

especificação formal que é todavia acessível à partes interessadas mais importantes do projeto. É também função do Cleanroom prover um meio de realizar refinamentos sistemáticos a fim de derivar projetos e implementações a partir de especificações formais.

O CSP/FDR provê um meio de modelar as especificações e projetos Cleanroom para verificar suas corretudes. Também possibilita verificar automaticamente cada passo do refinamento. O processo Cleanroom é comentado melhor na Subseção 3.2.3.

O presente trabalho também propõe alternativas nesse sentido a fim de facilitar e automatizar essa tradução. A notação informal de modelagem são os Diagramas de Sequência e a notação final é a matematicamente precisa Redes de Petri. Também como no presente trabalho, o trabalho de Broadfoot e Hopcroft utiliza uma notação para descrever sistemas concorrentes. No caso daquele trabalho, é a notação CSP.

2.7 Gomes

O trabalho de Gomes (GOMES, 2005) apresenta uma metodologia de projeto de sistemas embarcados, mostrando as funcionalidades do sistema através dos Diagramas de Caso de Uso da UML e usando Redes de Petri como o principal formalismo de modelagem. Sistemas de Automação Predial são usados como um tipo específico de sistemas embarcados. O modelo dos sistemas é construído sobre a análise de cada Caso de Uso e sua tradução para um modelo parcial. Após isso, o conjunto de modelos parciais é combinado usando-se o operador de adição de redes, permitindo a construção de todo o modelo do sistema. A adição de modelos parciais suporta a introdução de novas funcionalidades ao modelo em uma maneira incremental, iterativa e adititva, apesar de comportamentos transversais poderem ser adicionados também. O trabalho de Gomes é aplicado a um sistema de monitoramento predial composto de três subsistemas: iluminação, HVAC (aquecimento, ventilação e ar condicionado ou *heating, ventilation and air conditioning*) e detecção de intrusos.

2.8 Blaskovic

Metodologias de verificação como *model checking* e prova de teoremas são partes importantes do processo de desenvolvimento de sistemas complexos. O correto comportamento de sistemas embarcados de tempo-real concorrentes e reativos, que é o contexto de tal trabalho, é dependente de restrições temporais, dentre outras (BLASKOVIC, 2003).

O trabalho de Blaskovic apresenta esforços de modelagem a fim de se reduzir a distância entre software embarcado concorrente, ferramentas de *model checking*, modelos de computação e especificações formais (BLASKOVIC, 2003). O desenvolvimento de código de entrada para o *model checking* a partir de uma especificação formal ou código fonte pode produzir centenas de linhas de código. A fim de automatizar tal tarefa, introduzida no trabalho de Blaskovic como extração de modelos, o *framework* ou arcabouço de meta-modelagem é desenvolvido para acabar com diferenças semânticas entre várias ferramentas de verificação de software em uma e entre descrição do sistema por outro lado.

O arcabouço de modelagem é um ambiente onde o modelo para experimentação com a solução do problema é construído. O modelo consiste de Processos de Comunicação Elementares Interconectados ou *Interconnected Elementary Communicating Processes* (ECP). Uma interpretação de interconexões reflete o modelo de computação usado: orientado a objeto, Redes de Petri, autômatos reativos comunicantes, modelos de coordenação, agentes inteligentes etc.

Os ECP são agrupados em três níveis de abstração para se beneficiar de aspectos teóricos como álgebras de processo ou Redes de Petri de alto nível. Falando a grosso modo, um modelo nesse contexto é uma coleção de ferramenta conectadas e a modelagem é o processo de integração de ferramentas.

2.9 Fernandes

O trabalho de Fernandes (FERNANDES, 2007) usa um estudo de caso sobre uma especificação de um controle de elevador. O artigo apresenta uma abordagem que pode traduzir uma dada descrição em UML em uma Rede de Petri Colorida (RPC). No presente trabalho, usam-se as Redes de Petri Convencionais. As descrições em UML devem estar especificadas na forma de Casos de Uso e de Diagramas de Sequência da UML 2.0. O modelo RPC constitui-se de uma representação única, coerente e executável de todos os comportamentos possíveis que são especificados pelos dados artefatos da UML. RPCs constituem-se de uma linguagem de modelagem formal que habilita a construção e análise de modelos de comportamento escaláveis e executáveis. Um uso combinado de UML e RPC pode ser útil em muitos projetos. O trabalho de Fernandes também se propõe a construir ferramentas de software para realizar a referida tradução.

2.10 Eshuis

Há também o de Eshuis (ESHUIS, 2006). Duas traduções de diagramas de atividade para a linguagem de entrada do NuSMV, um Model Checker simbólico, são apresentadas. Ambas traduções mapeiam um diagrama de atividades em uma máquina de estados finita e são inspiradas pela semântica de Statecharts existentes. As traduções em nível de requisitos definem máquinas de estados que podem ser verificadas eficientemente, apesar de assumirem uma hipótese de sincronismo perfeito. A tradução em nível de implementação define máquinas de estado que não podem ser verificadas tão eficientemente, mas que são mais realísticas visto que não se valem valer de uma hipótese de perfeito sincronismo. A fim de justificar o uso de tradução em nível de requisitos, é mostrado no trabalho de Eshuis que para um grande conjunto de diagramas de atividade e determinadas propriedades, ambas as traduções são equivalentes: independentemente de qual tradução é utilizada, a saída do *model checking* é a mesma. Ademais, para algumas propriedades lineares, a tradução em nível de implementação é equivalente a uma versão um pouco modificada da tradução

em nível de requisitos. Naquele trabalho, ambas as traduções são usadas para se verificar restrições de integridade de dados para um diagrama de atividades e um conjunto de diagramas de classe da UML que especificam os dados manipulados nas atividades.

Os diagramas de atividade da UML descrevem o ordenamento global de pedaços atômicos do comportamento, conhecidos como atividades. A notação é inspirada tanto por gráficos de fluxo quanto por redes de Petri. Diagramas de Atividade permitem a especificação de processos complexos de software que possuem paralelimos, laços e comportamento orientado a eventos. Eles podem ser usados para o comportamento de alguns casos de uso ou especificar o *workflow* ou processo de negócios de uma organização. Recentemente, eles foram adotados por diversas padrões de serviços da *web* para especificar uma conversação na mesma. O diagramas de atividade são também uma notação da UML para expressar aspectos dinâmicos do sistema. O presente trabalho usa outra notação para aspectos dinâmicos, a de diagram de sequência.

2.11 Bernardi

Vale citar o trabalho de Bernardi (BERNARDI, 2002), que estuda o uso dos Diagramas de Sequência e Statecharts da UML para a validação e avaliação de desempenho de sistemas. É assumido que o sistema é especificado como um conjunto de Statecharts e que os Diagramas de Sequência são usados para representar "execuções de interesse". A UML carece da falta de uma semântica formal e, por isso, não é possível aplicar, diretamente, técnicas matemáticas em seus modelos para fins de validação de sistema. Para alcançar essa meta, o trabalho de Bernardi propõe uma tradução automática de *Statecharts* e Diagramas de Sequência - também usados no presente trabalho - para Redes de Petri Generalizadas e Estocásticas e uma composição dos modelos de rede resultantes adequados para alguma meta de análise. A tradução para redes de Petri é baseada na sintaxe abstrata das colaborações da UML e em seu pacote de máquinas de estado (a partir do qual *Statecharts* e Diagramas de Sequência são derivados) e a construção de um modelo completo utiliza

extensivamente composição.

2.12 Yao

O trabalho de Yao (YAO, 2006) visa ajudar o desenvolvimento de aplicações de software de alta qualidade por meio de uma abordagem para verificação de consistência de modelos dinâmicos da UML baseados em técnicas de Redes de Petri. A notação de Redes de Petri Coloridas Estendidas (RPCE) é usada para descrever formalmente transições de estados de objetos individuais e interações entre esses objetos e é, portanto, capaz de verificar a consistência de modelos baseados na teoria de redes de Petri. No trabalho de Yao, são considerados, assim como no de Bernardi, diagramas de sequência e *statecharts*. A abordagem começa com uma estratégia para modelos dinâmicos da UML e então discute a tradução de *statecharts* com estados compostos para a notação RPCE. O grafo de alcançabilidade é gerado para conduzir o processo de verificação de consistência. O artigo discute todas as fases da abordagem e ilustra o conceito por meio de um exemplo.

2.13 Amorin

O trabalho de (AMORIN, 2006) apresenta uma metodologia para mapear a linguagem dos *Live Sequence Charts* (LSCs) para um modelo em Rede de Petri (RdP) equivalente como uma abordagem para análise e verificação de propriedades de sistemas embarcados. A linguagem LSC é uma linguagem para especificação de sistemas, permitindo ao projetista especificar o que deveria acontecer para todas as possibilidades de execução de um sistema, bem como modelar anti-cenários. Entretanto, a análise e verificação de propriedades do sistema não são possíveis. A fim de permitir o diagnóstico de especificações inconsistentes, além da simulação, verificação e análise devem ser consideradas. Portanto, a proposição de um modelo RdP para LSC permite a verificação e análise de sistemas descritos em LSC contribuindo para o aumento da confiança do projetista no processo

de desenvolvimento do sistema e redução dos riscos que podem levar a falhas no projeto.

2.14 Bonnefoi

O trabalho de Bonnefoi (BONNEFOI, 2007) é sobre a aplicação de métodos formais para modelar e analisar sistemas complexos no contexto do Sistema de Transporte Inteligente - *Intelligent Transport Systems* (ITS). Ele sugere que uma metodologia de especificação baseada num conjunto de diagramas da UML para gerar um conjunto completo de modelos analisáveis. a metodologia integra os requisitos do desenvolvimento incremental e modular de sistemas complexos. A análise feita no modelo formal é conduzida por critérios qualitativos, tal como no método proposto no presente trabalho, é verificada por ferramentas de model checking. As orientações propostas são ilustradas por um estudo de caso o qual considera carros em situações de tráfico, trocando informações sobre seus estados para alcançar consistência entre suas decisões de condução.

2.15 Zhao

A UML ainda carece de capacidades mais bem elaboradas de análise e verificação. Portanto, muitos modelos não podem realmente ser analisados em detalhes, em particular quando eles são usados para descrever sistemas distribuídos complexos (ZHAO, 2004). Sendo a falta de métodos de análise e verificação fatores limitadores do uso da UML, a qualidade dos mesmos perde com isso.

No trabalho de Zhao (ZHAO, 2004), uma técnica altamente automatizada pode formalmente transformar diagramas UML para fins de verificação. Primeiramente, a estrutura hierárquica do meta modelo da UML é revista assim como as relações entre as diferentes visões. Uma abordagem para transformar e verificar diagramas *statecharts* é desenvolvida baseando-se em transformações de grafos. Esta abordagem pode ser usada para transformar diagramas UML para Redes de Petri

enquanto preserva as propriedades de consistência dinâmicas. Finalmente, a abordagem é validade através de estudo de caso.

2.16 Trowitzsch

O trabalho de Trowitzsch (TROWITZSCH, 2006) propõe a modelagem de sistemas técnicos e seus comportamentos por meio das Máquinas de Estado da UML e do Perfil para Escalonabilidade, Desempenho e Tempo ou *Profile for Schedulability, Performance, and Time* (SPT) da UML. Esse perfil permite a descrição detalhada de aspectos quantitativos dos sistemas tais como escolhas de tempo e probabilísticas. Para os modelos resultantes uma transformação para Redes de Petri Estocásticas é estabelecida. Portanto, esse trabalho toma uma viés a mais, o de medição de desempenho, em relação ao trabalho apresentado aqui. As medidas de desempenho de Redes de Petri podem ser determinadas por simulação ou análise numérica. Uma parte do futuro Sistemas de Controle do Trem Europeu (*European Train Control System* - ETCS) serve como um exemplo de aplicação. A relação entre a qualidade de comunicação do ETCS e a distância mínima entre trens subsequentes é investigada também.

2.17 Resumo do Capítulo

Várias iniciativas ao longo dos anos têm se destacado no que diz respeito à adição de mais rigor formal no desenvolvimento de software embarcado. Seguindo a linha da presente dissertação, alguns trabalhos relacionados principalmente à tradução de modelos na notação UML para Redes de Petri - qualquer que sejam suas variantes - foram apresentados bem como alguns que se propõe a realizar verificação direta a partir de diagramas UML.

Pode-se notar por eles e também pela maioria dos trabalhos pesquisados até o presente momento que os diagramas de Máquina de Estados, os antigos *Statecharts*, são mais empregados

neste processo. Os Diagramas de Sequência portanto, constituíram-se como desafio para o presente trabalho. O próximo Capítulo, o de Conceitos Básicos, introduz os principais conceitos usados no presente trabalho como fundamentos para o seu desenvolvimento.

Capítulo 3

Conceitos Básicos

Este Capítulo apresenta os conceitos necessários à contextualização e compreensão desta dissertação. Ele aborda os seguintes temas: Sistemas Embarcados, Processo de Desenvolvimento de Software e Linguagem de Modelagem Unificada ou Unified Modeling Language (UML), Redes de Petri e Métodos Formais. A primeira Seção mostra as principais características, aplicações e limitações dos sistemas embarcados. Porém, o foco desta Seção é software embarcado, exatamente o objeto de estudo do presente trabalho. A seguir, a Seção sobre Processo de Desenvolvimento de Software mostra algumas das atividades correntes na Indústria e na Academia para a correta construção de software embarcado. A próxima Seção detalha a importância da UML e foca em seus Diagramas de Sequência, uma das notações para se conceber a visão dinâmica de um sistema. Os Diagramas de Sequência são usados no trabalho como notação para descrição em alto nível de aplicações. Na sequência, a Seção sobre Redes de Petri introduz este importante formalismo matemático, o qual especifica rigorosamente sistemas críticos. Por fim, métodos formais são apresentados, em especial a técnica de *Model Checking* ou Verificação de Modelos, a qual visa atestar a corretude das propriedades informadas do sistema em questão.

3.1 Sistemas Embarcados

Por volta do início da década de 80, os projetistas usavam microprocessadores de 16 bits para criar aplicações muito sofisticadas. Um exemplo é o 68000 (WOLF, 2002), cujos projetistas usaram para construir controles de motor de carro e que se baseava em algoritmos sofisticados de gerência de combustível. Usar o poder computacional do 68000 melhorou a eficiência e reduziu as emissões de carbono ao mesmo tempo em que tornou o motor mais fácil de partir em muitas situações. Alguns projetistas de algoritmos que executavam em sistemas de controle usavam métodos numéricos tais como filtros de Kalman (WOLF, 2002).

As impressoras a laser e a jato de tinta também emergiram nos anos 1980. Em seus mecanismos e processos internos também requeriam e até hoje fazem uso de suporte computacional, inclusive com características de tempo-real (WOLF, 2002). Telefones celulares usam co-processadores para processamento de sinal digital e microprocessadores para controle de botões e processamento de protocolo (WOLF, 2002). Por volta do início da década de 1990, um telefone celular típico continha cinco ou seis Processadores de Sinais Digitais (do Inglês *Digital Signal Processor* - DSP) e CPUs (WOLF, 2002). A infra-estrutura do telefone também usa DSPs embarcados e microprocessadores extensivamente.

Esta seção discorre sobre alguns tópicos e conceitos selecionados em sistemas embarcados. Após uma breve introdução, questões relacionadas a projeto de software embarcados são discutidas.

3.1.1 Visão Geral

Sistemas embarcados são, geralmente, sistemas digitais que executam um conjunto específico de funções dentro de um sistema maior. São sistemas controlados por microprocessadores ou microcontroladores onde, na maioria dos casos, o usuário não está ciente da existência de um computador como parte do sistema, uma vez que sua principal função não é a computacional.

Este tipo de sistema já está completamente inserido no mundo moderno, estando presente tanto no cotidiano doméstico, através de uma infinidade de dispositivos eletro-eletrônicos, quanto em aplicações de alta tecnologia como naves espaciais e equipamentos bélicos.

Mesmo possuindo valor significativo, a funcionalidade de um sistema embarcado é, normalmente, secundária em relação ao sistema maior no qual este está inserido. Esta posição secundária, aliada a diversidade de contextos nos quais este tipo de sistema é aplicado, explicam, em parte, as enormes restrições às quais tais sistemas estão sujeitos. É bastante usual a exigência de características como: confiabilidade garantida, baixos custos, requisitos de tempo-real, dimensões reduzidas, baixo consumo de energia, garantia de funcionalidade em ambientes atípicos, entre tantos outros requisitos que os diferenciam profundamente de outros sistemas computacionais de propósito geral.

Inicialmente, a maior parte dos sistemas embarcados era essencialmente baseada em hardware, utilizando-se por exemplo Circuito Integrado de Aplicação Especifica (ASIC - *Application Specific Integrated Circuit*). Entretanto, à medida que o avanço tecnológico dos processadores tem aumentado o poder computacional e reduzido as dimensões e preços desses componentes, um número cada vez maior de funcionalidades tem sido implementadas por meio de software. Atualmente, estima-se que mais de 80% das funcionalidades dos sistemas embarcados modernos esteja baseada em software (SANGIOVANNI-VINCENTELLI, 2001). Mesmo trazendo algumas desvantagens relacionadas a desempenho e consumo, esta migração proporciona vantagens que facilmente a justificam, como: flexibilidade, menores custos e maior acessibilidade. Assim, exerce influência direta na popularização e proliferação desse tipo de sistema. Essa proliferação tem demandado o desenvolvimento de sistemas cada vez mais complexos em prazos cada vez mais curtos.

Nesse contexto de inúmeras restrições, crescente complexidade e prazos muito curtos, metodologias de especificação, projeto e desenvolvimento, específicas para sistemas embarcados, desempenham um importante papel, sendo que em muitos casos, já não é possível fornecer as garantias necessáras com o uso de uma abordagem puramente empírica. Este trabalho se concentra no para-

digma denominado Hardware-Software Codesign que é apresentado mais a frente.

A seguir, um resumo das principais características de sistemas embarcados discorridas até aqui e que podem influenciar no processo de desenvolvimento dos mesmos.

3.1.2 Características de Sistemas Embarcados

Em termos gerais, algumas diferenças para aplicações *desktop* que influenciam o processo de desenvolvimento são apresentadas a seguir (KOOPMAN, 1996), (KOPETZ, 2002), (COOLING, 2003):

- A interface homem-máquina que consiste de dispositivos de entrada e saída não devem solicitar nenhum treinamento para operar o sistema. Em outras palavras, deve ser de fácil uso por parte do usuário.
- O custo de uma simples unidade de sistema embarcado deve ser a menor possível com o propósito de reduzir custos de produção. Deste modo, o projeto do sistema deve ser altamente otimizado para o custo do ciclo de vida e eficiência.
- O sistema embarcado tem funcionalidade fixa e estrutura rígida. O software é específico para a aplicação e reside em uma memória somente de leitura.
- A qualidade do software deve ser alta, pois não existe muita flexibilidade para mudar o software depois de ser liberado para o mercado.
- O tempo de vida de sistemas embarcados é freqüentemente longo. Desta forma, é necessário uma boa porta de diagnóstico para que seja possível realizar manutenção em campo.

3.1.3 Software Embarcado

Software embarcado é o tema deste trabalho e pode ser apenas uma das partes de um sistema embarcado maior integrado de hardware e software. As atividades necessárias ao seu desenvol-

vimento são melhor descritas na Seção 3.2. Porém em (SANGIOVANNI-VINCENTELLI, 2001) e (SANGIOVANNI-VINCENTELLI, 2001b) Sangiovanni-Vincentelli e Martin apresentam alguns problemas relacionados ao desenvolvimento de softwares embarcados e alguns desafios que qualquer metodologia proposta deve considerar.

Resumidamente os problemas relacionados em (SANGIOVANNI-VINCENTELLI, 2001) são:

- o uso de linguagens de programação que proporcionam alto desempenho, mas que são conhecidamente fracas em legibilidade e manutenibilidade (ex. C e assembly);
- a necessidade de suporte de hardware específico para depuração e avaliação de desempenho;
- paradigmas de programação que não fornecem garantias suficientes de que parâmetros de qualidade e prazos serão de fato atingidos;
- dificuldade de verificação da corretude do projeto;
- pouca atenção dada a restrições relacionadas a *deadline*, uso de memória e consumo de energia.

De acordo com (SANGIOVANNI-VINCENTELLI, 2001b), uma metodologia para desenvolvimento precisa obrigatoriamente considerar pelo menos os seguintes desafios:

- modelagem de propriedades não-funcionais;
- uso extensivo de componentes de software;
- arquitetura do sistema e software;
- validação e verificação em nível de sistema;

O presente trabalho visa atender a estas demandas. A Seção 3.2 descreve as principais atividades necessárias ao desenvolvimento de software embarcado.

É importante destacar que para o projeto de softwares embarcados considerados simples, não existe a necessidade de uma metodologia rebuscada. No entanto, uma abordagem empírica não pode ser considerada para aplicações mais complexas.

3.2 Processo de Desenvolvimento de Software

Um processo de desenvolvimento de software é um modelo de trabalho usado no desenvolvimento de um produto de software. Sinônimos incluem processo do ciclo de vida do software e processo de software (SOFTWARE DEVELOPMENT PROCESS, 2009). Há vários modelos para tais processos, cada um deles descrevendo abordagens para uma variedade de tarefas ou atividades executadas durante o processo. Um número cada vez maior de organizações de desenvolvimento de software implementam modelos de processos. Muitas delas são da área militar (SOFTWARE DEVELOPMENT PROCESS, 2009). Antes de detalhar as atividades pertinentes, é válido mostrar rapidamente o contexto no qual ele se encaixa, que é o de Engenharia de Sofware.

3.2.1 Engenharia de Software

Engenharia de Software é a aplicação de uma abordagem sistemática, disciplinada e quantificável ao desenvolvimento, operação e manutenção do software, bem como o estudo de tais abordagens. Isto é, a aplicação de engenharia ao software (PRESSMAN, 2001).

O termo Engenharia de Software primeiramente apareceu na Conferência de Engenharia de Software da OTAN (Organização do Tratado do Atlântico Norte) de 1968 e tinha o propósito de gerar reflexões sobre a "crise de software"corrente à época (PRESSMAN, 2001). Desde então, ela tem continuado como uma área de estudos dedicada à criação de software de qualidade, barato, manutenível e de rápida construção. Visto que o campo é ainda relativamente jovem comparado aos outros campos da Engenharia, há ainda muito trabalho e debate sobre o que Engenharia de Software realmente é.

Entretanto, a Engenharia de Software evoluiu para muito mais do que simplesmente a atividade de programação, como é apresentado na Subseção 3.2.2. E a Engenharia de Software para sistemas embarcados tem sido amplamente usada atualmente tanto por causa dos novos e complexos requisitos quanto por sua crescente ubiquidade. Um exemplo da aplicação desses conceitos foi o desenvolvimento software da cabine (ou *cockpit*) do novo Airbus A-380, que gerencia várias funções (A380, 2009).

3.2.2 Atividades do Desenvolvimento de Software

Num processo de desenvolvimento de software tanto convencional quanto embarcado, podem-se destacar as seguintes atividades básicas: Análise de Requisitos, Especificação, Arquitetura, Projeto, Implementação ou Codificação, Testes, Implantação e Manutenção. Na sequência, são apresentadas suas definições (SOFTWARE DEVELOPMENT PROCESS, 2009). Cada uma destas atividades produz um produto de saída. O foco do experimento presente trabalho é a aplicação de métodos formais ao projeto, mais especificamente, o projeto de baixo nível. Todavia, é válido mostrar resumidamente todas as atividades.

Análise de Requisitos

A atividade mais importante na criação de um produto de software é a extração dos requisitos dos clientes seguida da atividade de análise dos mesmos (SOFTWARE DEVELOPMENT PROCESS, 2009). Uma vez que os requisitos gerais são coletados do cliente, uma análise do escopo do desenvolvimento é feita. Frequentemente também é realizada uma análise ou investigação do domínio do problema, quando o desenvolvedor não é suficientemente familiar ao domínio do software a ser desenvolvido.

A Análise de Requisitos deve clarificar e definir requisitos funcionais e não funcionais e restrições de projeto (DOD, 2001). Requisitos funcionais descrevem as funções que o sistema deve executar; por exemplo, acender um LED quando algum tipo de estímulo externo for capturado.

Requisitos não funcionais definem quantidade, qualidade, cobertura e prazos. A restrições de projeto definem os fatores que limitam a flexibilidade do projeto, tais como condições do ambiente ou ameaças externas, condições do cliente ou da legislação em vigor, sendo muitas vezes vistos também como requisitos não funcionais (DOD, 2001).

Uma técnica conhecida para se documentar os requisitos potenciais de um novo software são os Casos de Uso (BURNS, 2001). Cada caso de uso provê um ou mais cenários que mostram como o sistema deve interagir com o usuário final ou outro sistema a fim de se alcançar alguma meta específica do negócio. Casos de uso tipicamente evitam jargões técnicos, preferindo a linguagem do usuário final ou especialista no domínio. Casos de uso têm frequentemente co-autoria por engenheiros de requisitos e as partes interessadas (*stakeholders*. Casos de uso contêm uma descrição textual de todos caminhos nos quais o sistema pode passar. Porém, eles não descrevem processos internos do sistema, nem explicam como o sistema será implementado. Eles simplesmente mostram os passos que um usuário segue para realizar uma tarefa e todas as maneiras de se interagir com esse sistema. A Unified Modeling Language (UML) agregou a seu vocabulário de diagramas essa técnica de Casos de Uso e é prática comum usá-los.

Além da UML, é válido citar, ainda que não seja o foco do presente trabalho, outras técnicas, métodos e notações visando facilitar a atividade de Análise de Requisitos. São elas: notação PSL (BURNS, 2001), notação CORE (BURNS, 2001), as atividades do projeto FOREST (BURNS, 2001) e a linguagem ALBERT desenvolvida pelo projeto Esprit II ICARUS (BURNS, 2001).

Especificação

A Especificação Funcional ou, simplesmente, Especificação é a atividade de definir precisamente o software a ser escrito, possivelmente de uma maneira rigorosa (SOFTWARE DEVELOPMENT PROCESS, 2009). No contexto dos softwares embarcados e da Engenharia de Sistemas de forma geral, ela ajuda a descrever os aspectos comportamentais do software a ser desenvolvido.

A Especificação ajuda a evitar duplicações e inconsistências, permite estimativas precisas do

carga de trabalho a ser empregada bem como dos recursos (FUNCTIONAL SPECIFICATION, 2009). Ela também atua como um meio-termo na negociação entre os projetistas do software e seus clientes, provê documentação de configuração e ainda orientações aos testadores do software final produzido (FUNCTIONAL SPECIFICATION, 2009).

Arquitetura

A arquitetura de um sistema de software ou, simplesmente, arquitetura de software, refere-se a uma representação abstrata daquele sistema (SOFTWARE DEVELOPMENT PROCESS, 2009). A arquitetura está concentrada em assegurar que o sistema de software irá atender aos requisitos do produto final bem como que novos requisitos possam ser contemplados no futuro. O termo arquitetura também se refere à documentação do sistema de software (SOFTWARE ARCHITECTURE, 2009). A documentação da arquitetura facilita a comunicação entre as partes interessadas (*stakeholders*) envolvidos no processo, documenta qualquer decisão sobre o projeto de alto-nível e permite o reuso de componentes do projeto e padrões entre os projetos (SOFTWARE ARCHITECTURE, 2009).

Há várias arquiteturas de software para sistemas embarcados. O presente trabalho, conforme explicado no Capítulo 4, verifica formalmente software que apresenta estes modelos de arquitetura: (i) Sistema Controlado por Interrupção, (ii) Multi-tarefa cooperativa e (iii) Distribuída.

Alguns sistemas embarcados são predominantemente controlados por interrupções. Isto significa que tarefas executadas pelo sistema são disparadas por diferentes tipos de eventos. Uma interrupção poderia ser gerada por exemplo por um temporizador (*timer*) em uma frequência pré-definida ou por um controlador da port serial recebendo um byte (SOFTWARE ARCHITECTURE, 2009). Isto caracteriza a arquitetura “Sistema Controlado por Interrupção”.

Estes tipos de sistemas usualmente executam uma tarefa simples em um laço principal mas esta tarefa não é muito sensível a atrasos (*delays*) inesperados. Às vezes, o gerenciador de interrupções adicionará tarefas mais extensas à estrutura de fila. Mais tarde, após o gerenciador de interrupções

ter finalizado, estas tarefas são executadas pelo laço principal.

Um sistema multi-tarefa não-preemptivo é muito similar ao esquema de laço de controle simples, exceto que o laço é escondido na biblioteca ou API do software. O programador define uma série de tarefas e cada tarefa recebe seu próprio ambiente no qual executará. Então, quando uma tarefa está ociosa, ela chama uma rotina. Essa rotina pode ser "pausa" (*pause*), "espera" (*wait*), "dar prioridade a" (*yield*), dentre outras (SOFTWARE ARCHITECTURE, 2009). As vantagens e desvantagens são muito similares ao laço de controle, exceto pelo fato de que adicionar-se novo software é mais fácil, simplesmente escrevendo-se uma nova tarefa ou adicionando-se à estrutura de fila (SOFTWARE ARCHITECTURE, 2009). Isto caracteriza a arquitetura "Multi-tarefa cooperativo".

Há ainda arquitetura distribuída consistindo de vários elementos de processamento e a monolítica, na qual o software é auto-contido e contém apenas uma camada com lógica de negócios mesclada à apresentação e a um possível acesso de dados (SOFTWARE ARCHITECTURE, 2009). Vale citar a diferença de Arquitetura de Software para o Projeto Detalhado, tema do tópico 3.2.2. A arquitetura de software, também descrita como "Projeto Estratégico", é uma atividade concentrada em restrições de projeto globais tais como paradigmas de programação, estilos arquiteturais, padrões de engenharia de software baseada em componentes, princípios de projeto e regulamentações em Lei (SOFTWARE ARCHITECTURE, 2009). Projeto Detalhado, também descrito como "Projeto Tático", é uma atividade concentrada em restrições de projeto locais tais como padrões de projeto (*design patterns*), idiomas de programação e refatoramentos.

Projeto

Projeto de Software, do termo em Inglês *Software Design*, é um processo de solução de problemas e planejamento em certo nível para uma solução de software (SOFTWARE DESIGN, 2009). Após os propósitos, especificações e arquitetura do software serem determinados, o projetista projetará sua solução em software efetivamente. Claramente o estágio mais importante no desenvolvimento

de software embarcado é a geração de um projeto consistente que satisfaça uma competente especificação de requisitos (BURNS, 2001).

Um projeto de software pode ser independente de plataforma apresentando um nível de abstração mais alto ou específico de plataforma, apresentando um nível de abstração mais baixo, sendo neste caso conhecido como Projeto Detalhado ou de Baixo Nível (SOFTWARE DESIGN, 2009). Além disso, várias notações ou linguagens podem ser usadas para se expressar o projeto, tais como a Unified Modeling Language, descrita na Seção 3.3, *Business Process Modeling Notation* (BPMN), *Jackson Structured Programming* (JSP), Fluxogramas (SOFTWARE DESIGN, 2009) ou Communicating Sequential Processes (CSP) todas aptas a especificar e analisar sistemas concorrentes.

No contexto do software convencional, a atividade de projeto consiste em se realizar refinamentos sucessivos do projeto levando a representações de nível de abstração cada vez mais baixo (PRESSMAN, 2001). No processo de engenharia de sistemas embarcados, o mesmo princípio vale. E, considerando-se que a UML, como explicado na Seção 3.3, não está limitada a um nível de abstração, pode-se utilizar algum dos seus diagramas em um Projeto Detalhado ou de Baixo Nível de um software embarcado, após os devidos refinamentos e projetos de mais alto nível. O projeto detalhado resultante poderia ser a representação dos processos e tarefas de um software, por exemplo. No experimento proposto pelo presente trabalho, técnicas de verificação formal, em particular de verificação de modelos, são aplicadas a um projeto detalhado de um software embarcado de controle.

Implementação ou Codificação

A implementação é a realização de uma especificação técnica ou projeto como um programa, componente de software ou sistema computacional (IMPLEMENTATION, 2009).

Firmware é o conjunto de intruções operacionais programadas diretamente no hardware de um equipamento eletrônico. É armazenado permanentemente num circuito integrado (chip) de

memória de hardware, como uma ROM, PROM, EPROM ou ainda EEPROM e memória flash, no momento da fabricação do componente (FIRMWARE, 2009). Firmwares estão presentes em computadores na forma de BIOS, celulares, iPODs, câmeras digitais, impressoras e virtualmente quaisquer equipamentos eletrônicos da atualidade, incluindo eletrodomésticos como fornos de microondas ou lavadoras (FIRMWARE, 2009).

No caso de microcontroladores, o firmware é implantado após um ciclo de desenvolvimento que envolve o uso de um conjunto de ferramentas que podem incluir editores, montadores, compiladores, depuradores, simuladores, emuladores e programadores da memória Flash (SOFFEL, 2003). É válido citar os passos normalmente tomados na programação ou implementação de software ou firmware para microcontroladores devido a sua grande popularidade no meio de software embarcado, quais sejam (SOFFEL, 2003):

1. Escrever o código em alguma linguagem como Assembly ou C por exemplo;
2. Traduzir o código para instruções de máquina de forma que possa ser efetivamente executado. Para isto existem montadores, compiladores, linkeditores e diversas bibliotecas;
3. Depurar o código com a ajuda de ferramentas de depuração, incluindo emuladores e
4. Programar a memória Flash para implantar uma versão funcional do sistema.

Algumas linguagens próprias para a programação de software embarcado, de maneira geral, incluem, mas não estão limitadas a: Assembly, Jovial (usada pela Força Aérea dos Estados Unidos), Coral 66 (padronizada pelo Ministério da Defesa do Reino Unido), C, C++, Ada, Modula-1 e Modula-2, PEARL (usada na Alemanha para aplicações de controle de processos), Mesa (introduzida pela Xerox), CHILL (para programação de aplicações de telecomunicação), Occam e versões de tempo real de Java (BURNS, 2001).

Teste, Implantação e Manutenção

O Teste de Software é uma investigação empírica conduzida para prover as partes interessadas com informação sobre a qualidade do produto ou serviço objeto do teste (KANER, 2006), com respeito ao contexto no qual é pretendido que ele opere. Isso inclui, mas não está limitado ao processo de executar um programa ou aplicação com o intuito de descobrir-se erros de software.

A Implantação inicia-se após o código estar apropriadamente testado, estiver aprovado para distribuição ou distribuição. A atividade anterior, "Implementação ou Codificação" mostrou um exemplo de implantação de software embarcado, no contexto de microcontroladores. Já a Manutenção é um melhoramento de software para lidar com problemas novos ou novos requisitos. Ela corrige ou aumenta o software (SOFTWARE ENGINEERING, 2009).

3.2.3 Metodologias de Desenvolvimento de Software

Um processo de software é um conjunto de atividades cuja meta é o desenvolvimento ou evolução de software (SOMMERVILLE, 2007). As atividades vistas até aqui são genéricas e compõe um processo de software. Um modelo de processo de software ou metodologia de desenvolvimento de software é uma representação simplificada de um processo de software, apresentada a partir de uma visão específica (SOMMERVILLE, 2007). Ou seja, as atividades vistas na Subseção 3.2.2 são realizadas de maneira personalizadas em uma metodologia específica.

A Indústria de software embarcado usa muitas vezes metodologias ou abordagens estruturadas tradicionais do domínio de sistemas de informação muito pela falta de métodos "testados e aprovados" (BURNS, 2001). Algumas das mais usadas são (BURNS, 2001):

- Modelo Cascata Clássico
- Modelo Cascasta Iterativo
- Modelo Prototipagem

- Modelo Evolucionário
- Modelo em Espiral

Entretanto, se o sistema embarcado em questão contemplar características de tempo-real sofrerá da falta de expressividade que essas metodologias possuem para o domínio de tempo-real (BURNS, 2001). No passado, apesar da natureza ubíqua das aplicações embarcadas, o desenvolvimento de tais sistemas não era o foco de atenção dentro da área de software. Certamente, na maioria dos casos, as técnicas de projeto de software tem sido desenvolvidas primeiramente para atender às necessidades dos desenvolvedores de aplicações de mesa (ou *desktop*) e então, subsequentemente, "adaptadas" numa tentativa de atender às necessidades dos desenvolvedores de aplicações de tempo-real e/ou embarcadas (PONT, 2008). Pont (PONT, 2008) argumenta que as técnicas de desenvolvimento de software resultantes de tais adaptações, embora não sem méritos, não podem atualmente atender às necessidades dos projetistas de sistemas embarcados.

Há ainda os mais formais, como Cleanroom que é um processo de desenvolvimento de software planejado para produzir software com um ceto nível de confiabilidade (CLEANROOM SOFTWARE ENGINEERING, 2009) e é aplicado principalmente a sistemas embarcados críticos (BROADFOOT, 2005). Tem como princípios básicos o desenvolvimento de software baseado em métodos formais. O desenvolvimento Cleanroom faz uso do método da estrutura de caixa para especificar e projetar um produto de software. Verificação que o projeto corretamente implementa a especificação é realizada através de revisão de equipe (CLEANROOM SOFTWARE ENGINEERING, 2009). O desenvolvimento Cleanroom usa uma abordagem iterativa, na qual o produto é desenvolvido em incrementos que gradualmente aumentam a funcionalidade geral implementada. A qualidade de cada incremento é medida contra padrões pré-estabelecidos para verificar que o processo de desenvolvimento tem sido executado aceitavelmente (CLEANROOM SOFTWARE ENGINEERING, 2009). Uma falha no alcance dos padrões significa um retorno à atividade de projeto. O teste de software nesse modelo é conduzido como um experimento estatístico.

Outro exemplo de método com características formais é o método B (ABRIAL, 1996). O método B é uma metodologia para desenvolver software usando o paradigma especificação e verificação que é realizada de maneira mais adequada através da prova de teorema. Para o suporte dessa metodologia existem ferramentas responsáveis pela análise sintática das especificações, enquanto que a parte semântica dessas sofre uma carência de suporte, pois há poucas ferramentas capazes de validá-las de forma rigorosa, automática e eficiente.

No contexto de desenvolvimento de sofware embarcado, não há uma única metodologia para sistemas embarcados (BURNS, 2001). Podem ser utilizadas várias metodologias para o desenvolvimento de sistemas embarcados, em especial de tempo-real tais como MASCOT, JSD, Yourdon, MOON, OOD, RTSA, HOOD, DARTS, ADARTS, CODARTS, EPOS, MCSE, PAMELA, HRT-HOOD, Octopus, dentre outras (BURNS, 2001). Cada uma delas tem suas vantagens ou desvantagens.

A Unified Modeling Language (UML), como o próprio nome diz, é uma linguagem e não um método. Consequentemente, para ser usado no contexto do desenvolvimento de software embarcado, é necessário ser inserida numa metodologia. Douglass (BURNS, 2001) propôs a metodologia ROPES (Rapid Object-oriented Process for Embedded Systems) para uso com a UML. A ROPES é baseada em um ciclo de vida orientado em direção à rápida geração de protótipos. As atividades são: análise, projeto, implementação e testes. Em comum com outras metodologias UML há o fato de que ROPES é orientada a casos de uso. Um conjunto de casos de uso é identificado e ordenado de acordo com certas prioridades, riscos e comunalidade. Estes são então usados para produzir protótipos do sistema.

O presente trabalho consiste na verificação formal por meio de *model checking* de um modelo de software. Como métodos formais podem ser usados em qualquer uma das atividades do processo de desenvolvimento de software (MÉTODOS FORMAIS, 2009), o método proposto aqui não fixa em uma metodologia específica e nem mesmo em uma atividade de alguma metodologia. A única orientação é que haja uma modelagem do software a ser verificado sob a notação do

Diagrama de Sequência da UML (que mais a frente será automaticamente traduzida para a notação Redes de Petri). O experimento apresentado no Capítulo 5 mostra o projeto detalhado de um software de controle na notação de Diagrama de Sequência.

3.3 Linguagem Unificada de Modelagem

A Linguagem Unificada de Modelagem (*Unified Modeling Language* - UML) é uma linguagem visual de propósito geral que é usada para especificar, visualizar, construir e documentar os artefatos de um sistema de software. Ela captura decisões e entendimentos sobre os sistemas a serem construídos. É usada para se entender, projetar, navegar em, configurar, manter e controlar a informação sobre os sistemas.

Por ser apenas uma notação e não uma metodologia completa de desenvolvimento de software, a UML não está limitada a uma etapa de do processo de desenvolvimento e nem a um nível de abstração específico, podendo seus diagramas serem usados para se expressar um modelo em qualquer passo do processo de desenvolvimento ou nível de abstração.

3.3.1 Breve Sumário

A UML pode ser usada em todos os métodos de desenvolvimento, estágios do ciclo de vida, domínios de aplicações e mídias. A linguagem de modelagem tem a capacidade de unificar a experiência passada sobre técnicas de modelagem e incorporar as melhores práticas de software atuais em uma abordagem padrão. A UML inclui conceitos semânticos, notações e guias. Ela tem partes estáticas, dinâmicas, de ambiente e organizacionais. Ela tem a capacidade de suportar através de ferramentas de modelagem visual interativa que têm geradores de código e geradores de relatórios. A especificação UML não define um processo padrão mas é particularmente útil com um processo de desenvolvimento iterativo. Ela também tem a capacidade de suportar a maioria dos processos de desenvolvimento orientado a objeto.

A UML captura informação sobre a estrutura estática e o comportamento dinâmico de um sistema. Um sistema é modelado como uma coleção de objetos discretos que interagem a fim de realizar um trabalho que, em última instância, beneficia um usuário externo. A estrutura estática define os tipos de objetos que são importantes para o sistema e suas implementações, assim como as relações entre os objetos. O comportamento dinâmico define a história dos objetos sobre o tempo e as comunicações sobre objetos para alcançar suas metas. Modelar um sistema a partir de vários pontos de vista separados porém relacionados permite que ele seja entendido para diferentes propósitos.

A UML não é uma linguagem de programação. As ferramentas podem prover geradores de código de UML em uma variedade de linguagens de programação, bem como construir modelos a partir de Engenharia Reversa de programas existentes. A UML também não é uma linguagem altamente formal criada para prova de teoremas. Há várias linguagens para isso, mas as mesmas não são fáceis de se entender ou usar para muitos propósitos. A Seção 3.5 discorre mais a respeito de métodos formais. A UML é uma linguagem de modelagem discreta. Não deve ser usada para se modelar sistemas contínuos tais como os encontrados em especial na Física. A UML é projetada para ser um linguagem de modelagem de propósito geral para sistemas de eventos discretos tais como aqueles construídos por software, firmware ou lógica.

3.3.2 Visões da UML

No que diz respeito aos vários conceitos e construções da UML, não há uma linha rígida entre mas, por conveniência, eles são divididos em várias visões (BOOCH, 1998). Uma visão é simplesmente um subconjunto de construções de modelagem da UML o qual representa um aspecto do sistema. A divisão em diferentes visões é algo de arbitrário, mas espera-se que seja intuitivo. Um ou dois tipos de diagrams provêm uma notação visual para os conceitos em cada visão.

Um sistema é composto por diversos aspectos: funcional (que é sua estrutura estática e suas interações dinâmicas), não funcional (requisitos de tempo, confiabilidade, desenvolvimento, etc.)

e aspectos organizacionais (organização do trabalho, mapeamento dos módulos de código, etc.) (BOOCH, 1998). Então o sistema é descrito em um certo número de visões, cada uma representando uma projeção da descrição completa e mostrando aspectos particulares do sistema. Cada visão é descrita por um número de diagramas que contém informações que dão ênfase aos aspectos particulares do sistema. Existe em alguns casos uma certa sobreposição entre os diagramas o que significa que um destes pode fazer parte de mais de uma visão. Os diagramas que compõem as visões contêm os modelos de elementos do sistema. As visões que compõem um sistema são:

- Visão de Caso de Uso. Descreve a funcionalidade do sistema desempenhada pelos atores externos do sistema (usuários). A visão de Caso de Uso é central, já que seu conteúdo é base do desenvolvimento das outras visões do sistema. Essa visão é montada sobre os diagramas de Caso de Uso e eventualmente diagramas de atividade.

- Visão Lógica. Descreve como a funcionalidade do sistema será implementada. É feita principalmente pelos analistas e desenvolvedores. Em contraste com a visão de Caso de Uso, a visão lógica observa e estuda o sistema internamente. Ela descreve e especifica a estrutura estática do sistema (classes, objetos, e relacionamentos) e as colaborações dinâmicas quando os objetos enviarem mensagens uns para os outros para realizarem as funções do sistema. Propriedades como persistência e concorrência são definidas nesta fase, bem como as interfaces e as estruturas de classes. A estrutura estática é descrita pelos diagramas de classes e objetos. O modelamento dinâmico é descrito pelos diagramas de estado, sequencia, colaboração e atividade.

- Visão de Componentes. É uma descrição da implementação dos módulos e suas dependências. É principalmente executado por desenvolvedores, e consiste nos componentes dos diagramas.

- Visão de concorrência. Trata a divisão do sistema em processos e processadores. Este aspecto, que é uma propriedade não funcional do sistema, permite uma melhor utilização do

ambiente onde o sistema se encontrará, se o mesmo possui execuções paralelas, e se existe dentro do sistema um gerenciamento de eventos assíncronos. Uma vez dividido o sistema em linhas de execução de processos concorrentes (*threads*), esta visão de concorrência deverá mostrar como se dá a comunicação e a concorrência destas threads. A visão de concorrência é suportada pelos diagramas dinâmicos, que são os diagramas de estado, sequencia, colaboração e atividade, e pelos diagramas de implementação, que são os diagramas de componente e execução.

- Visão de Organização. Finalmente, a visão de organização mostra a organização física do sistema, os computadores, os periféricos e como eles se conectam entre si. Esta visão será executada pelos desenvolvedores, integradores e testadores, e será representada pelo diagrama de execução.

3.3.3 Diagrama de Sequência

Os diagramas utilizados pela UML até sua versão 1.4 são compostos de nove tipos: diagrama de Caso de Uso, de classes, de objeto, de estado, de sequência, de colaboração, de atividade, de componente e o de execução (BOOCH, 1998). Todos os sistemas possuem uma estrutura estática e um comportamento dinâmico. A UML suporta modelos estáticos (estrutura estática), dinâmicos (comportamento dinâmico) e funcional. A Modelagem estática é suportada pelo diagrama de classes e de objetos, que consiste nas classes e seus relacionamentos. Os relacionamentos podem ser de associações, herança (generalização), dependência ou refinamentos. Os modelamentos dinâmicos são suportados pelos diagramas de estado, sequência, colaboração e atividade. E o modelamento funcional é suportado pelos diagramas de componente e execução.

Um diagrama de sequência mostra a colaboração dinâmica entre os vários objetos de um sistema (BOOCH, 1998). O mais importante aspecto deste diagrama é que, a partir dele, percebe-se a sequência de mensagens enviadas entre os objetos. Ele mostra a interação entre os objetos, alguma

coisa que acontecerá em um ponto específico da execução do sistema. O diagrama de sequência consiste em um número de objetos mostrado em linhas verticais. O decorrer do tempo é visualizado observando-se o diagrama no sentido vertical de cima para baixo. As mensagens enviadas por cada objeto são simbolizadas por setas entre os objetos que se relacionam.

Diagramas de sequência possuem dois eixos: o eixo vertical, que mostra o tempo e o eixo horizontal, que mostra os objetos envolvidos na sequência de uma certa atividade. Eles também mostram as interações para um cenário específico de uma certa atividade do sistema.

No eixo horizontal estão os objetos envolvidos na sequência. Cada um é representado por um retângulo de objeto (similar ao diagrama de objetos) e uma linha vertical pontilhada chamada de linha de vida do objeto, indicando a execução do objeto durante a sequência, como exemplo citam-se as mensagens recebidas ou enviadas e ativação de objetos. A comunicação entre os objetos é representada como linha com setas horizontais simbolizando as mensagens entre as linhas de vida dos objetos. A seta especifica se a mensagem é síncrona, assíncrona ou simples. As mensagens podem possuir também números sequenciais, os quais são utilizados para tornar mais explícito as sequência no diagrama. Em alguns sistemas, objetos executam concorrentemente, cada qual com sua linha de execução (*thread*). Se o sistema usa linhas concorrentes de controle, isto é mostrado como ativação, mensagens assíncronas, ou objetos assíncronos.

Os diagramas de sequência podem mostrar objetos que são criados ou destruídos como parte do cenário documentado pelo diagrama. Um objeto pode criar outros objetos através de mensagens. A mensagem que cria ou destrói um objeto é geralmente síncrona, representada por uma seta sólida.

Em síntese: o Diagrama de Seqüência é uma das ferramentas UML usadas para representar interações entre objetos de um cenário, realizadas através de operações ou métodos (procedimentos ou funções). Este diagrama é construído a partir do Diagrama de Casos de Uso normalmente (BOOCH, 1998). Primeiro, se define qual o papel do sistema (Casos de Uso). Depois, é definido como o software realizará seu papel (seqüência de operações).

O diagrama de sequência dá ênfase à ordenação temporal em que as mensagens são trocadas

entre os objetos de um sistema. Entende-se por mensagens os serviços solicitados de um objeto a outro, e as respostas desenvolvidas para as solicitações. Há várias ferramentas suportando a modelagem de Diagramas de Sequência, tanto proprietárias quanto de código aberto. Um exemplo de ótima ferramenta proprietária é a EventStudio System Designer 4.0 (EVENTSTUDIO, 2009). Um exemplo de uma excelente ferramenta de código aberto para modelagem de diagramas de sequência é a Jude UML Modeling Tool (JUDEUML, 2009).

3.3.4 Justificativa da Adoção da UML e de seu Diagrama de Sequência

O método proposto no presente trabalho, como detalhado no Capítulo 4, necessita de uma notação semi-formal e popular no meio da Indústria e Academia para a modelagem inicial do software embarcado. Nesta Subseção, são mostradas as razões para a adoção da UML como tal notação e de seu Diagrama de Sequência.

A comunidade de software embarcado tem um grande interesse na UML como um mecanismo padrão de expressividade. Apesar de a UML prover algum grau de padronização em termos de aspectos sintáticos do projeto do sistema, sua semântica é muito genérica (SANGIOVANNI-VINCENTELLI, 2001). O que faz da UML uma meta-linguagem no mínimo razoável para se modelar sistemas embarcados / de tempo-real são os seguintes atributos chave aqui somente citados e melhor explicados em (MARTIN, 2002):

- Conjunto heterogêneo de notações.
- O fato de que a UML não é uma única linguagem, mas um conjunto de notações, sintaxe e semântica para pertmitir a criação de famílias de linguagens para aplicações em particular. Em outras palavras, a UML é, como dito anteriormente, uma “meta-linguagem”.
- Mecanismos de extensão via perfis, estereótipos, marcações (*tags*) e restrições para aplicações particulares.

- Modelagem de caso de uso para descrever ambientes do sistema, cenários do usuário e casos de teste.

- Uma grande organização padronizadora conhecida como Object Modelling Group - OMG (OMG, 2009), a qual tem promovido o uso da UML entre projetistas e patrocinado sua evolução através de forças-tarefa, grupos de trabalho e similares. A OMG também promove o desenvolvimento de perfis específicos de domínio de aplicação.

- Suporte para especificação de sistemas orientados a objeto, projeto e modelagem causando apelo na comunidade de desenvolvedores.

- Suporte para semântica de máquinas de estado as quais podem ser usadas para modelagem e síntese.

- Suporte para decomposição e refinamento estrutural baseado em objetos.

A UML provê duas notações mais comuns para interações: os diagramas de sequência e os de colaboração. Todavia, outra motivação para se usar os de sequência e não os de colaboração que eles são bem mais usados na prática do que os de colaboração e mostram aproximadamente a mesma informação (DOUGLASS, 2003). Os diagramas de máquina de estado da UML também são bastante aplicados na modelagem de sistemas embarcados. A partir de diagramas de sequência pode-se sintetizá-los, como sugere o trabalho de Latronico (LATRONICO, 2001).

A UML não foi criada visando especificamente o escopo de sistemas embarcados. Porém, alguma noção de tempo foi incluída na mesma através dos diagramas de sequência (LAPLANTE, 2004).

Além disso, a maioria dos trabalhos vistos até aqui, no que diz respeito à tradução UML para Redes de Petri, foram desenvolvidos usando-se Diagramas *statecharts* ou, na UML 2.0, Máquinas de Estado. Portanto, o presente trabalho tomou como **desafio** tomar como notação UML o Diagrama de Sequência e propor meios de se utilizá-lo para os devidos fins de tradução automática

para Redes de Petri.

Um importante aspecto na modelagem de sistemas está associado à escolha do formalismo adequado para produzir um modelo do sistema. De fato, não há "formalismo correto", de uma perspectiva geral. De acordo com as necessidades de modelagem dos requisitos é que um formalismo específico deve ser escolhido baseando-se nas características do sistema e na experiência do projetista (GOMES, 2005). A massificação do uso da UML sugere que a mesma é de conhecimento da grande maioria dos projetistas de software embarcado, não sendo um problema portanto seu uso pelo presente método.

3.3.5 Fraquezas

Apesar de seu sucesso como notação visual e unificada na Indústria, a UML ainda sofre da falta de opções para análise e verificação (ZHAO, 2004). Portanto, muitos modelos não podem realmente ser analisados detalhadamente, em particular quando eles são usados para descrever sistemas distribuídos complexos. A falta de métodos de análise e verificação limita o uso da UML e reduz a qualidade dos modelos UML. Então faz-se necessário desenvolverem-se técnicas de análise e verificação de modelos o que é significante para os projetistas.

A UML também sofre da falta de semântica formal precisa específica para software embarcado, como citado em 3.3.4, a qual cria obstáculos para a verificação e validação formal do projeto do sistema (ZHAO, 2004). Por outro lado, modelos estabelecidos em muitos domínios matemáticos (como Redes de Petri, sistemas de transição, álgebras de processo etc) são precisos e poderiam ser analisados e verificados usando-se várias ferramentas (ZHAO, 2004).

Assim, transformações de modelos na notação UML para modelos naquelas notações são extremamente úteis para análise e verificação dos modelos originais em UML. Modelos UML são projetados em modelos matemáticos para transformação e os resultados da análise formal são anotados de volta nos modelos UML para esconder-se a matemática dos projetistas (ZHAO, 2004). O presente trabalho realiza atividades nesse sentido, conforme descrito no Capítulo 4. Outra fonte de

informações a respeito de deficiências da UML no contexto de sistemas embarcados é (MARWEDEL, 2006).

3.3.6 Outras Abordagens de UML para Sistemas Embarcados e de Tempo Real

Devido à crescente quantidade de software em sistemas embarcados, a UML tem ganhado importância para os mesmos. Então, várias propostas para extensões da UML, principalmente no que diz respeito a tempo-real, têm sido feitas à linguagem bem como outros mecanismos têm sido propostos a fim de se superar suas limitações neste contexto (MARWEDEL, 2006). O presente trabalho as cita para fins de conhecimento. Muitas delas poderiam perfeitamente adaptadas no contexto do presente trabalho.

UML-SPT

Em 1998, a OMG (Object Management Group) criou um grupo para estudar a aplicação da UML para sistemas de tempo real. A partir desse trabalho surgiu em 2003 um perfil da UML conhecido como Perfil para Desempenho de Escalonabilidade e Especificação de Tempo (*Profile for Schedulability Performance and Time Specification* - SPT) (UML-SPT, 2009) com objetivo de definir uma maneira de utilizar as funcionalidades da UML para modelar conceitos e práticas de sistemas de tempo real, ou seja, escalonabilidade e desempenho (UML-SPT, 2009).

UML-RT

A linguagem de modelagem ROOM (Real Time Object-Oriented Modeling) tem com o objetivo de especificar, visualizar, documentar e automatizar a construção de sistemas de tempo real. ROOM usa conceitos que captura padrões estruturais de comunicação entre componentes de software para suporte arquitetural de padrões de projeto. Os modelos de ROOM são composto de atores que

se comunicam entre si enviando mensagens via protocolos (WEIGUO, 2000). O ROOM têm sido bastante aplicado na indústria de telecomunicações para desenvolver softwares embarcados (GU, 2004) e utiliza basicamente dois tipos de diagramas: Diagrama de estruturas e Diagrama de Estados. Na representação do diagrama de estruturas o ROOM utiliza os seguintes elementos de construção: cápsulas, portas e conectores.

A Rational Software Corporation (atualmente incorporada pela IBM) fez uma parceria com a ObjectTime Ltd. Para desenvolver UML-RT UML-RT (UML for *Real Time*). UML-RT usa os mecanismos de extensão do UML para incorporar conceitos de ROOM do ObjectTime (WEIGUO, 2000). UML-RT é extremamente poderoso para modelar a natureza reativa dos sistemas de tempo real.

Como uma ferramenta de sucesso, UML-RT fornece um modelo para análise complexa de sistemas de tempo real dirigidos à eventos simulando a forma como os sistemas de tempo real realmente trabalham e, por último, ajuda a construir e gerar código para aplicações críticas de tempo real.

A metodologia utilizanda pelo UML-RT é um processo iterativo composto de diversos ciclos de desenvolvimento. O número de ciclos depende da característica do software, bem como da sua complexidade. A metodologia engloba as seguintes etapas de desenvolvimento:

- Especificação de Requisitos: Esta fase se caracteriza pela definição das funcionalidades do sistema, sem detalhes de implementação do mesmo. Também se decide quais funcionalidades serão desenvolvidas no ciclo corrente.

- Análise: Permite modelar o software de forma conceitual sem se preocupar com detalhes de implementação.

- Projeto: Nesta fase se deriva as classes que representam uma solução de implementação do sistema. É comum a utilização de diversos padrões de projeto para a solução do problema.

- Implementação: Se caracteriza pela escrita do codigo fonte a partir das classes especificadas na fase de projeto.

- Teste: Verificação das funcionalidades especificadas.

UML-RT é extremamente poderoso na modelagem de natureza reativa dos sistemas de tempo real. Porém, UML-RT não é formalmente bem definido. Isto é uma limitação relevante do UML-RT pois grande parte das aplicações de tempo real são críticas. Portanto a verificação de propriedades (segurança, utilidade etc), simulação de sistemas e geração de casos de testes são tarefas difíceis quando as especificações são escritas em notações semi-formais como a UML ou UML-RT (BIANCO, 2008).

UML-RT apresenta uma notação eficiente para projeto e implementação de sistemas, mas não é muito bem adaptado para representação requisitos ou especificações. Por exemplo, quando a modelagem do ambiente na qual um sistema de tempo real deve trabalhar, é freqüentemente necessário representar comportamentos não determinísticos ou eventos simultâneos. Estes fenômenos não são suportados pelo UML-RT (BIANCO, 2008).

UML-RT não suporta análise de escalonamento de tempo real e não gera uma implementação que atende às restrições de tempo. Os principais domínios de aplicações da UML-RT são sistemas de telecomunicações, que são geralmente sistemas não críticos de tempo real. Talvez, por este motivo, os projetistas da ferramenta CASE do Rose-RT não têm dado muita ênfase em aplicações de tempo real com forte restrição ao tempo (GU, 2004).

SysML

Em 2006, o Object Management Group (OMG) anunciou a adoção da Linguagem de Modelagem de Sistemas (*Systems Modeling Language* - SySML) como uma especificação final (HAUSE, 2006). A especificação da SysML foi em resposta à proposta conjunta elaborada pela INCOSE (International Council on Systems Engineering) e pela própria OMG para uma versão personali-

zada da UML 2.0 específica para lidar com as necessidades especiais de engenheiros de sistemas. A especificação da SysML foi desenvolvida por uma equipe grande incluindo fabricantes de ferramentas, usuários na indústria, agências governamentais norte-americanas e organizações em geral por um período de mais de 3 anos (HAUSE, 2006).

A notação SysML é uma linguagem visual de modelagem de sistemas complexos que estende a notação UML 2 a fim de prover suporte à especificacão, análise, projeto, verificação e validação de sistemas complexos que incluam componentes de hardware, software, dados, pessoal procedimentos e outras facilidades. Esta notação visa ser usada em diferentes metodologias incluindo análise estruturada, orientação a objeto dentre outras (HAUSE, 2006). Ela reusa um subconjunto dos conceitos e diagramas da UML 2 e aumenta-os com alguns novos diagramas e construções apropriadas para modelagem de sistemas.

Modelos Executáveis

Há muitas maneiras nas quais a UML pode ser usada. Alguns desenvolvedores a usam informalmente. Outros a usam como "planta-baixa" da arquitetura do software que é traduzida automaticamente para um esqueleto de código. Porém existe uma maneira ainda mais sofisticada de se fazer uso da mesma que é a UML Executável (MELLOR, 2007). Com ela, o desenvolvedor pode adicionar pedaços de código-fonte diretamente ao modelo tornando-o um modelo executável. Porém, se a saída gerada for modificada manualmente, o modelo não mais refletirá o código-fonte. Isto pode então acarretar uma engenharia "round trip" (reversa) gerando-se o modelo a partir do código, o que faz muito sentido visto que o modelo suporta uma relação de um-para-um com o código. Este ponto de vista é promovido por Bruce Powel Douglass (DOUGLASS, 2004).

OCL

A Linguagem de Restrição de Objetos (*Object Constraint Language* - OCL) é uma linguagem declarativa para descrever regras e restrições que se aplicam aos modelos da UML (OCL, 2006).

Ela foi desenvolvida pela IBM e é atualmente parte do padrão UML. Inicialmente, a OCL era apenas uma especificação formal de extensão de linguagem à UML. Atualmetne, ela pode ser usada com qualquer mecanismo *Meta-Object Facility* (MOF) da OMG, incluindo a própria UML. A OCL é uma linguagem textual precisa que provê restrições e expressões de consulta de objetos sobre qualquer recomendação padrão da OMG para transformar-se modelos, a conhecida especificação *Queries/Views/Transformations* (QVT) (OCL, 2006).

Restrições são regras aplicadas a vários elementos do modelo e podem ser pré-condições, condições estruturais, pós-condições, tempo e assim por diante (OCL, 2006). Elas permitem que o usuário use construções genéricas, tais como classes ou associações, e acrescente certa semântica.

Vaziri e Jackson (VAZIRI, 1999) apontam, entretanto, algumas deficiências da OCL, as quais são citadas aqui e melhor detalhadas em seu trabalho:

- OCL parece ser bastante próxima de uma linguagem de implementação do que de uma linguagem conceitual porque ela usa operações em restrições e seu tipo "sistema" é próximo aquele de uma linguagem de programação OO.
- As expressões OCL são em alguns momentos desnecessariamente verbosas.
- As expressões OCL são frequentemente difíceis de se ler desnecessariamente.
- Finalmente, a OCL não é uma linguagem independente (*stand-alone*): um modelo sempre necessita de um diagrama de classes UML acompanhando-o.

3.4 Redes de Petri

A teoria inicial das Redes de Petri foi apresentada na tese de doutoramento *Kommunikation mit Automaten* do Dr. C. A. Petri em 1962 na Faculdade de Matemática e Física da Universidade de Darmstadt, na então Alemanha Ocidental. O trabalho de Petri atraiu a atenção de A. W. Holt que, em conjunto com outros pesquisadores, desenvolveu muito da teoria, notação e representação

das redes de Petri. De 1970 a 1975, o grupo de estrutura da computação do MIT foi o mais ativo na condução da pesquisa sobre Redes de Petri. Em 1975, houve uma Conferência sobre Redes de Petri. No entanto, não houve publicação dos anais (DICESARE, 1991). Em 1979, pesquisadores de vários países europeus reuniram-se em Hamburgo para um curso avançado sobre a *Teoria Geral das Redes de Processos e Sistemas* (BRAUER, 1979). Seguiram-se diversos outros trabalhos propondo alterações ao modelo original, tais como redes com arcos inibidores e redes temporizadas determinísticas e estocásticas.

Atualmente, Redes de Petri é considerada uma técnica para especificação de sistemas concorrentes consolidada. Grupos de pesquisa em todo o mundo têm Redes de Petri como tema, desenvolvendo estudos sobre seus aspectos teóricos e suas aplicações. Apenas para constar, diversas técnicas de modelagem matemática de sistemas em muitas áreas da Ciência têm sido propostas. Barroca e McDermid (BARROCA, 1992) apresentam a seguinte classificação:

1. Técnicas Baseadas em Modelos. Fornecem descrições abstratas sobre estados e operações que transformam os estados. No entanto, não oferecem meios explícitos para especificar concorrência. Um exemplo é a linguagem Z (SPIVEY, 1985).

2. Técnicas Baseadas em Álgebras de Processo. Tais técnicas fornecem meios explícitos para se especificar concorrência. O comportamento dos processos é representado através de comunicações observáveis. Exemplos incluem CCS (MILNER, 1989), CSP (HOARE, 1985) e LOTOS (LOTOS, 1997).

3. Técnicas Baseadas em Lógica. Uma grande variedade de técnicas baseadas em Lógica têm sido propostas, onde se analisam as relações causais e aspectos relacionados à temporização. Um exemplo é a Lógica Modal de Ações.

4. Técnicas Baseadas em Redes. Essas técnicas modelam concorrência através de mecanismos implícitos de fluxo de *tokens* na rede. Este fluxo é controlado por condições que habilitam a realização de tarefas (eventos). O grande exemplo são as Redes de Petri.

3.4.1 Visão Geral

Uma rede de Petri ou rede de transição é uma das várias representações matemáticas para sistemas distribuídos discretos. Como uma linguagem de modelagem, ela define graficamente a estrutura de um sistema distribuído como um grafo direcionado com comentários. Possui nós de posição, nós de transição e arcos direcionados conectando posições com transições. A notação básica de Redes de Petri é mostrada na Figura 3.1.

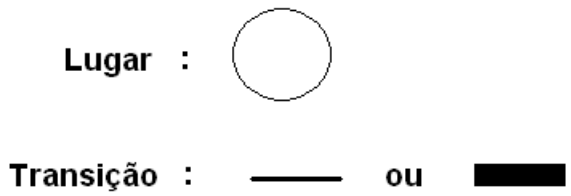

Figura 3.1: Elementos Básicos das Redes de Petri

Estes dois elementos são os vértices do grafo associado às rede de Petri. Os vértices são interligados por arcos dirigidos. Os arcos que interligam lugares às transições correspondem à relação entre as condições verdadeiras que, em um dado momento, possibilitam a execução das ações, enquanto os arcos que interligam transições aos lugares representam a relação entre as ações e as condições que se tornam verdadeiras com a execução das ações.

A qualquer momento durante a execução de uma rede de Petri, cada lugar pode armazenar um ou mais *tokens*. Diferente de sistemas mais tradicionais de processamento de dados, que podem processar somente um único fluxo de *tokens* entrantes, as transições de redes de Petri podem consumir e mostrar *tokens* de múltiplos lugares. Uma transição só pode agir nos tokens se o número requisitado de *tokens* aparecer em cada posição de entrada.

A execução de uma rede de Petri é não-determinística. Isso significa que múltiplas transições podem ser habilitadas ao mesmo tempo (cada uma pode ser disparada) e que nenhuma transição deve ser obrigatoriamente executada em determinado momento.

Transições agem em *tokens* de entrada por um processo denominado disparo. Quando uma

transição é disparada, ela consome os *tokens* de seus lugares de entrada, realiza alguma tarefa de processamento, e realoca um número específico de *tokens* nos seus lugares de saída. Isso é feito atomicamente. Como disparos são não determinísticos, redes de Petri são muito utilizadas para modelar comportamento concorrente em sistemas distribuídos. Com essas noções em mãos, segue a definição formal de Redes de Petri.

3.4.2 Definição Formal

Uma Rede de Petri (S, T, F, M_0, W, K) é dada por:

- S, um conjunto de lugares.
- T, um conjunto de transições.
- F, um conjunto de arcos também chamados de relações de fluxo. Ele é sujeito à restrição de que nenhum arco conecta dois lugares ou transições ou, mais formalmente, $F \subseteq (S \times T) \cup (T \times S)$.
- $M_0 : S \to \mathbb{N}$, conhecido como marco inicial, no qual para cada lugar $s \in S$, existem $n \in \mathbb{N}$ tokens.
- $W : F \to \mathbb{N}^+$, conhecido como um conjunto de pesos de arco, relaciona para cada arco $f \in F$ um $n \in \mathbb{N}^+$ denotando quantos *tokens* são consumidos de um lugar por uma transição ou, alternativamente, quantos tokens são produzidos por uma transição e colocados em cada posição.
- $K : S \to \mathbb{N}^+$, conhecido como restrições de capacidade, relaciona para cada posição $s \in S$ um número positivo $n \in \mathbb{N}^+$ denotando o número máximo de *tokens* que podem ocupar aquele lugar.

3.4.3 Propriedades Matemáticas

O estado de uma rede de Petri é representado por um vetor M, no qual o primeiro valor do vetor é a quantidade de *tokens* na primeira posição da rede, o segundo é a quantidade de *tokens* na segunda posição, e assim por diante. Tal representação descreve completamente o estado de uma rede de Petri.

Uma lista de transição de estados $\sigma = \langle M_{i_0} t_{i_1} M_{i_1} ... t_{i_n} M_{i_n} \rangle$, que pode ser simplificada em $\vec{\sigma} = \langle t_{i_1} ... t_{i_n} \rangle$ é chamada uma sequência de disparo se cada transição satisfaz o critério de disparo, isto é, se existem *tokens* suficientes na entrada de cada transição. Nesse caso, a lista de transição de estados de $\langle M_{i_0} M_{i_1} ... M_{i_n} \rangle$ é chamada *trajetória*, e M_{i_n} é chamado *alcançável* a partir de M_{i_0} através da sequência de disparos de $\vec{\sigma}$. Matematicamente: $M_{i_0} \lfloor \vec{\sigma} \rfloor > M_{i_n}$. Todas as sequências de disparo que pode ser atingidas em uma rede N com estado inicial M_0 são denotadas como $L(N, M_0)$.

A matriz de transição de estados W^- é $|T|$ por $|S|$ e representa a quantidade de *tokens* passados por cada transição de cada posição. Similarmente, W^+ representa a quantidade de *tokens* dados por cada transição para cada posição. A soma dos dois, $W^- + W^+$, pode ser utilizada para calcular a equação mencinada anteriormente de $M_{i_0} \lfloor \vec{\sigma} \rfloor > M_{i_n}$. Ela agora pode ser escrita simplesmente como $M_0 - M_n = W^T \bullet \sigma$, no qual σ é um vetor de quantas vezes cada transição foi disparada na sequência. Vale notar que somente porque a equação pode ser satisfeita não significa que ela pode utilizada; para isso devem haver *tokens* suficientes para cada transição ser disparada, isto é, somente com a equação não é suficiente dizer que o estado M_n pode ser atingido a partir do estado M_0.

Todos os estados que podem ser atingidos em uma rede N com estado inicial M_0 são denotados como $R(N, M_0)$. Surge o problema de alcaçabilidade: é verdadeiro que $M_w \in R(N, M_0)$, no qual M_w é, por exemplo, um estado inválido tal qual um elevador se movendo enquanto a porta está aberta?

A alcançabilidade dos estados pode ser representada pelo grafo de alcançabilidade no qual os

destinos de um grafo direcionado representam estados (por exemplo M), e transições de arcos entre dois dos tais estados. O grafo é contruído como a seguir: o estado inicial M_0 é definido e todos as possibilidades de transição são exploradas a partir desse estado; depois a partir os estados resultantes da primeira iteração, e assim por diante. Enquanto a alcançabilidade parece ser uma boa ferramenta para encontrar estados errôneos, o grafo construído possui estados demais para problemas práticos. Por essas razões, a lógica linear temporal com o método de tableau é geralmente utilizado para provar que tais estados não podem ser alcançados. Essa lógica usa a técnica de semi-decisão para encontrar se realmente um estado pode ser alcançado, ao procurar um conjunto de condições necessárias para o estado ser alcançado e provando que tais condições não podem ser satisfeitas.

3.4.4 Atividade

Uma transição t de uma rede de Petri (N, M_0) está

- L_0 viva, ou morta, *sse* ela não pode ser disparada, isto é, não está em nenhum $\vec{\sigma} \in L(N, M_0)$.
- L_1 viva *sse* ela pode ser disparada, isto é, está em uma sequência de disparo $\vec{\sigma}$ na qual $\vec{\sigma} \in L(N, M_0)$.
- L_2 viva *sse* para qualquer número k inteiro positivo, t pode ser disparado pelo menos k vezes em uma sequência de disparo $\vec{\sigma}$ na qual $\vec{\sigma} \in L(N, M_0)$.
- L_3 viva *sse* existe uma sequência de disparo $\vec{\sigma} \in L(N, M_0)$ na qual t é disparada infinitamente.
- L_4 viva, ou viva, *sse* em qualquer estado alcançável $m(\forall M \in R(N, M_0))$, t é L_1 viva.

Uma transição L_3 viva é automaticamente L_0, L_1 e L_2 da mesma maneira. Uma rede de Petri é L_k viva *sse* toda transição pertencente é L_k viva.

3.4.5 Justificativa da Adoção de Redes de Petri

A ferramenta descrita na Seção 4.3 escolheu a notação de Redes de Petri como notação alvo da transformação a partir de diagramas UML por causa não somente do grande número de técnicas de análise mas do número de pesquisas bem sucedidas na área de verificação de diagramas UML da visão dinâmica por meio de Redes de Petri. As potencialidades das redes de Petri têm sido aplicadas nos mais diversos contextos. Conceitos e relações comuns ao domínio computacional, como por exemplo: paralelismo e concorrência, compartilhamento de recursos, sincronização, memorização, limitação de recursos, podem facilmente ser representados. Este trabalho assume como modelo básico uma das variantes mais difundidas de redes de Petri, conhecida como redes de Petri lugar-transição (L/T). Esta classe de rede é tão amplamente utilizada que algumas vezes é chamada simplesmente de rede de Petri.

3.4.6 Limitações

Existem redes de Petri nas quais as posições são limitadas em um número máximo de *tokens*. Nesse caso, a limitação é uma propriedade inerente. Apesar disso, redes de Petri podem ser definidas sem limitações com uma propriedade estrutural. Uma rede de Petri limitada não estruturadamente é k-limitada se não existe estado alcançável no qual alguma posição contém mais que k *tokens*. Uma rede de Petri é segura se ela é 1-limitada. Limitações de certas posições em uma rede inerentemente limitada podem ser feitas em uma rede inerentemente não-limitada ao utilizar uma transformada de posição, no qual uma nova posição (chamada contra-posição) é criada, e todas as transições que levam x *tokens* na posição original levam x *tokens* para a contra-posição. O número de *tokens* em M_0 agora deve satisfazer à equação $place + counter - place = boundedness$. Assim, aplicar uma transformada de posição para todas as posições em uma rede limitada, e restringindo o estado inicial M_0 para satisfazer a equação acima, uma rede limitada pode facilmente ser transformada em uma rede não-limitada. Desta maneira qualquer análise usada em uma rede não-limitada pode

ser utilizada em redes limitadas. A recíproca não é verdadeira.

3.4.7 Extensões

Existem várias extensões para redes de Petri. Algumas delas são completamente compatíveis com o modelo tradicional (por exemplo, redes de Petri coloridas), algumas adicionam propriedades que não podem ser definidas no modelo tradicional (por exemplo redes de Petri temporizadas). Se elas pudessem ser definidas no modelo tradicional, não seriam realmente extensões, e sim maneiras mais convenientes de demonstrar a mesma coisa, de forma que poderiam ser transformadas com fórmulas matemáticas para o modelo tradicional, sem perda de informação. Extensões que não podem ser transformadas para o modelo tradicional são muitas vezes poderosas, mas geralmente não possuem feramentas matemáticas suficientes para análise como no modelo tradicional de redes de Petri. O termo redes de Petri de alto-nível é utilizado por vários formalismos de redes de Petri que estendem o formalismo posição/transição. Isso inclue redes de Petri coloridas, redes de Petri hierárquicas e outras extensões como segue:

- Em uma rede de Petri tradicional, *tokens* são indistinguíveis. Em uma rede de Petri colorida, cada token possui um valor. Em várias ferramentas para redes de Petri coloridas, os valores dos *tokens* são tipados, e podem ser testados e manipulados usando-se uma linguagem de programação funcional. Uma derivação de redes de Petri coloridas são as redes de Petri bem formadas, nas quais os arcos e expressões de guarda são restritas para tornar a rede mais fácil de ser analisada.

- Outra extensão popular para redes de Petri é a hierarquia: hierarquia na forma de diferentes visões suportando níveis de refinamento e abstração, assim como estudado por Fehling. Outra forma de hirarquia é encontrada nas chamadas redes de Petri Objeto ou Sistemas Objeto, no qual uma rede de Petri pode conter outra rede de Petri como token, introduzindo o conceito de redes de Petri aninhadas que se comunicam ao sincronizar transições entre

diferentes níveis.

- Um Sistema de Adição de Vetor com Estados (*Vector Addition System with States* - VASS) pode ser visto como uma generalização de uma rede de Petri. Considerado um autômato finito no qual cada transição é denominada por uma transição da rede. A rede de Petri é então sincronizada com o autômato finito, isto é, uma transição no autômato é feita no mesmo momento que a mesma transição na rede de Petri. É somente possível utilizar uma transição no autômato se a transição correspondente na rede de Petri está habilitada, e é somente possível disparar uma transição na rede de Petri se existe uma transição no mesmo estado no autômato.

- Redes de Petri Priorizadas adicionam prioridades para as transições, de forma que uma transição não pode ser disparada se uma transição de prioridade maior está habilitada. Desta forma, transições estão em grupos de prioridade e, por exemplo, um grupo de prioridade 3 só pode disparar se todas as transições estão desabilitadas nos grupos 1 e 2. Mesmo em um grupo de prioridade, disparos ainda assim são não-determinísticos.

- A propriedade não determinística é bem útil pois permite ao usuário abstrair um grande número de propriedades (dependendo da finalidade da rede). Em certos casos, entretanto, existe também a necessidade de modelar também temporizações. Para esses casos são utilizadas redes de Petri temporizadas, no qual existem transições que são temporizadas, e possivelmente transições não temporizadas (nesse caso, transições não temporizadas são prioritárias em relação as temporizadas). Desta forma, a propriedade de tempo também pode ser modelada além da de estrutura. Uma derivação de redes de Petri temporizadas são as redes de Petri estocásticas, que adicionam tempos não-determinísticos através de aleatoriedade ajustável nas transições. A distribuição exponencial é utilizada para cronometrar essas redes. Nesse caso, o grafo de alcançabilidade dessas redes pode ser usado como uma Cadeia de Markov.

Existem várias outras extensões para redes de Petri. Entretanto, é importante saber que, ao se adicionar complexidade nas redes devido à inclusão de propriedades, mais difícil fica utilizarem-se ferramentas tradicionais para calcular certas propriedades da rede. Por essa razão, é sempre uma boa idéia utilizar-se a rede mais simples possível para uma tarefa dada.

3.5 Métodos Formais

O termo métodos formais é usado para se referir a qualquer atividade que recaia sobre representações matemáticas de software incluindo especificações de sistemas formais, prova e análise de especificação, desenvolvimento transformacional e verificação de programa (SOMMERVILLE, 2007). Todas estas atividades são dependentes de uma especificação formal de software. Uma especificação formal de software é uma especificação expressa numa linguagem cujo vocabulário, sintaxe e semântica são formalmente definidos (SOMMERVILLE, 2007). Esta necessidade por uma definição formal significa que as linguagens de especificação devem ser baseadas em conceitos matemáticos cujas propriedades são bem entendidas. O ramo da Matemática usado é a Matemática Discreta e os conceitos matemáticos são emprestados da Teoria de Conjuntos, Lógica e Álgebra.

O uso de métodos formais tem crescido na ára de desenvolvimento de sistemas críticos onde propriedades tais como segurança e confiabilidade são muito importantes. O alto custo das falhas nestes sistemas significa que as companhias cada vez mais estão dispostas a aceitar o alto custo introdutório e a alta curva de aprendizado dos métodos formais a fim de assegurar que seus softwares são tão confiáveis quanto possível.

3.5.1 Objetivos dos Métodos Formais

Métodos formais tipicamente não visam verificar todos as funcionalidades durante o desenvolvimento do software. Ao invés disso, técnicas individuais são projetadas para otimizar uma ou duas

partes do ciclo de vida ou processo de desenvolvimento (BOWEN, 1995). Há ainda, segundo Bowen (BOWEN, 1995), três usos gerais para métodos formais:

1. Verificação de Consistência: é onde requisitos comportamentais do sistema são descritos usando-se uma notação matematicamente rigorosa.

2. Verificação de Modelos: máquinas de estado são usadas para verificar se uma dada propriedade é satisfeita sob certas condições.

3. Prova de Teoremas: axiomas sobre o comportamento do sistema são usados para derivar uma prova de que o sistema se comportará em uma dada maneira.

Embora não seja o foco do presente trabalho, é válido citar que métodos formais também oferecem oportunidades para o re-uso de requisitos (BOWEN, 1995). Sistemas embarcados são frequentemente desenvolvidos como famílias de produtos similares ou como reprojetos incrementais de produtos existentes. No caso da primeira situação, métodos formais podem ajudar a identificar um conjunto consistente de requisitos e abstrações para reduzir o esforço duplicado de engenharia. Para re-projetos, ter especificações formais para os sistemas existentes provê uma referência precisa para o comportamento mais básico e provê um modo de se analisar mudanças propostas (BOWEN, 1995).

Benefícios reconhecidos dos métodos formais é que eles produzem especificações as quais são mais claras, mais precisas, menos ambíguas e mais explícitas. Isto facilita o processo de validação e provê uma base melhor para os passos subsequentes do processo de desenvolvimento de software, e.g. projeto, projeto detalhado, testes. A base formal das linguagens de especificação também torna possível desenvolver ferramentas para escrever, analisar ou animar especificações.

3.5.2 Classificação dos Métodos Formais

Apesar de o uso de lógica matemática ser um tema recorrente na disciplina de métodos formais, não há um único melhor método formal. Cada domínio de aplicação requer diferentes métodos de

modelagem e diferentes abordagens de prova (NASA, 2001). Há um grande número de métodos formais sob desenvolvimento ao redor do mundo bem como casos de sucesso na aplicação dos mesmos a aplicações reais.

Uma das primeiras classificações de métodos formais (ANÁLISE, 1999) foi a proposta por Cohen (COHEN, 1986). O sumário de sua categorização encontra-se na tabela 3.1.

Classes de Métodos Formais	Exemplos
Orientados a Modelos	VDM, HDM, Z, HOS, GIST
Orientados a Propriedades	Prolog, IOTA, ANNA
Convergentes	CLEAR, Larch
Concorrentes	Redes de Petri, Lógica Temporal
Geradores de Programas	SAFE, CIP

Tabela 3.1: Classificação de Métodos Formais de Cohen

Entretanto, Marwedel propõe a seguinte classificação dos métodos formais de acordo com o tipo de lógica empregada (MARWEDEL, 2006): Lógica Proposicional, Lógica de Primeira Ordem e Lógica de Ordem Superior. No caso da Lógica Proposicional, os modelos consistem de fórmulas descritas com variáveis booleanas e concetores tais como E e OU. Visto que a lógica proposicional é decidível, é também decidível investigar se duas representação são equivalentes ou não. Por exemplo, uma representação poderia corresponder a portas lógicas de um circuito real e a outra, a sua especificação. Provar a equivalência então provaria o efeito de todas as transformações de projeto (e.g. otimizações para consumo de energia ou tempo) serem corretas.

Verificadores de tautologias podem frequentemente lidar com projetos os quais são muito amplos para permitir validação exaustiva baseada em simulação (MARWEDEL, 2006). A principal razão para o poder dos verificadores de tautologia recentes é que usam estruturas conhecidas como Diagramas de Decisão Binários (*Binary Decision Diagrams* - BDDs) (WEGENER, 2000), (MCMILLAN, 1993). A complexidade de verificadores de equivalência de funções booleanas representadas com BDDs é linear no número de nodos da BDD (MARWEDEL, 2006). Em contraste, a verificação de equivalência para funções representadas por somas de produtos é NP-Difícil

(MARWEDEL, 2006). Ainda assim, o número de nodos BDD necessários para se representar uma certa função tem que ser levado em consideração. Muitas funções podem ser eficientemente representadas com BDDs. Em geral, entretanto, o número de nodos de BDDs cresce exponencialmente com o número de variáveis. Nesses casos nos quais funções podem ser eficientemente representadas com BDDs, verificadores de equivalência baseados em BDD tem frequentemente substituído simuladores e são usados para verificar redes de portas lógicas com milhões de transistores (MARWEDEL, 2006). O presente trabalho faz uso da ferramenta de *model checking* chamada Symbolic Model Verifier ou SMV (SMV, 2008), a qual faz uso das estruturas de BDD.

A Lógica de Primeira Ordem (LPO) inclui quantificação, usando-se $\exists$ e $\forall$. Alguma automação para verificar-se modelos baseados nessa lógica é factível. Entretanto, visto que a LPO é indecidível em geral, existem muitos casos de dúvida (MARWEDEL, 2006). Finalmente, a Lógica de Ordem Superior (LOS) permite funções serem manipuladas como outros objetos (MARWEDEL, 2006). Para a LOS, provas podem ser automatizadas com bastante esforço e tipicamente devem ser feitas manualmente com algum suporte a provas.

Uma outra classificação é a de Lamsweerde (LAMSWEERDE, 2002). Ela separa os métodos formais em cinco categorias, quais sejam: baseados no histórico, baseados em estado, baseados em transição, especificação funcional e especificação operacional. Os baseados em histórico usam lógica temporal para se referir aos estados passados, presentes e futuros. As especificações referem-se a pontos e intervalos de tempo. Os baseados em estado caracterizam os estados do sistema em dado momento. Invariantes e pré e pós condições restringem a aplicação de operações do sistema. Os baseados em transições representam as transições de um estado a outro. Os de especificação funcional representam um sistema como coleção estruturada de funções e os de especificação operacional representam o sistema como uma coleção estruturada de processos. Exemplos incluem as Redes de Petri. Observando-se a classificação proposta por Lamsweerde, é válido citar que o método atual, conforme as características apontadas no Capítulo 4 tem características de métodos baseados em histórico e de especificação operacional.

3.5.3 Métodos Formais no Processo de Desenvolvimento de Software

O emprego de métodos formais dentro do processo de desenvolvimento é recomendado pelo Software Engineering Body of Knowledge (SWEBOK) dentro da categoria de métodos e ferramentas de Engenharia de Software (SWEBOK, 2004). Eles são rotulados como métodos de desenvolvimento baseados em Matemática. Métodos formais podem ser usados em qualquer etapa do processo de desenvolvimento de software (TREMBLAY, 2000).

Tremblay (TREMBLAY, 2000) traz exemplos de aplicabilidade de métodos formais em várias atividades do processo de desenvolvimento. Modelos conceituais podem ser desenvolvidos usando-se notações formais (análise de requisitos). Especificações formais podem ser usadas para descrever-se o comportamento de módulos e componentes de software (projeto arquitetural). Dada uma especificação formal, propriedades podem ser verificadas formalmente (verificação); em certos casos, a especificação pode também ser animada (prototipagem) ou casos de teste podem ser automaticamente gerados (testes). Finalmente, usando-se uma metodologia apropriada, programas podem ser formalmente derivados a partir da especificação (codificação). O trabalho de Bourque *et al.* (BOURQUE, 1998) traz uma apresentação mais detalhada de como métodos formais se encaixam no processo de desenvolvimento de software.

Apesar de não haver somente uma atividade fixa onde usar métodos formais, vale citar que há algumas propostas como a de Sommerville (SOMMERVILLE, 2007) propondo usos específicos em certas atividades. Nela, se uma especificação formal de software é desenvolvida, isso usualmente vem depois dos requisitos do sistema terem sido especificados mas antes do projeto detalhado. E há também um pequeno laço de coleta de *feedback* entre a especificação detalhada dos requisitos e a especificação formal.

Davis (DAVIS, 2005) aponta para a necessidade de se aplicar métodos formais através da técnica de "Corretude por Construção" (DAVIS, 2005) detalhada em seu trabalho e também por todo o processo de desenvolvimento de maneira estrita. Neste caso, não importaria o quanto alguém tentasse aplicar métodos formais para a atividade de análise se a atividade de validação de sistemas

falhasse. Então, todos os esforços teria sido infrutíferos e re-trabalho seria gerado.

A conhecida técnica de verificação formal conhecida como *Model Checking* pode ser usada para se verificar muitos problemas interessantes, desde que eles possam ser representados num nível significativo de abstração por um sistema de estados finitos com vários ou apenas alguns estados (EMERSON, 2007). Portanto, qualquer modelo de qualquer uma das atividades do processo desenvolvimento descrito em 3.2 pode se valer das práticas de *model checking*, desde que representado por uma notação baseada em estados finitos. O experimento deste trabalho considera o uso de *model checking* sobre um modelo de baixo nível de software embarcado de controle. Este modelo é fruto de uma atividade de projeto detalhado. A técnica de *model checking* é detalhada na Subseção 3.5.5.

3.5.4 Motivos da Não Popularização

Apesar da necessidade de métodos formais no dia a dia do processo de desenvolvimento de software e da promessa de sucesso que os envolve, com exceção daqueles domínios de aplicação onde o uso de tais métodos é obrigatório, métodos formais não são amplamente encontrados na indústria no que diz respeito ao desenvolvimento de software até por não serem do domínio geral de todos os projetistas (BROADFOOT, 2005). Ademais, muitos que experimentaram o uso de métodos formais não pretendem repetir a experiência (BROADFOOT, 2005).

Dadas as necessidades de uso dos mesmos, o principal motivo de ainda não serem amplamente explorados na indústria é o conhecido hiato entre a pesquisa acadêmica e a indústria e as razões para o mesmo incluem: falta de escalabilidade dos métodos, acessibilidade limitada a não especialistas e imaturidade das técnicas e ferramentas (BROADFOOT, 2005). A seu favor, a indústria de software e sistemas embarcados tem o fato de que não tem demonstrado ser relutante em treinar seus desenvolvedores em novas técnicas.

Broadfoot (BROADFOOT, 2005) enumera alguns pontos baseados em observações e experiências práticas na indústria que explicam melhor os fatores citados no parágrafo anterior. São eles:

(i) Mudanças são custosas e disruptivas; (ii) Práticas de desenvolvimento centradas em testes e (iii) Pontos de início: requisitos informais *versus* especificações formais.

Sobre o fato de que mudanças são custosas e disruptivas, é válido dizer que a maioria dos negócios considerará métodos formais apenas se os problemas que elas enfrentarem forem suficientemente importantes e não adequados para serem resolvidos por métodos existentes.

O método formal proposto à organização não deve excluir do processo de desenvolvimento as partes interessadas críticas as quais não são especialistas em métodos formais. Apesar de os mesmos não entenderem muitas vezes os difíceis conceitos e notações matemáticas que os métodos formais usam, eles ainda possuem um conhecimento essencial do domínio do produto e isso os qualifica para participar da verificação da especificação formal.

O segundo tópico são as práticas de desenvolvimento de software embarcado geralmente centradas em testes. Elas se caracterizam por enfatizar a detecção e remoção de defeitos. Há uma crença genuína na indústria de que defeitos de software sào um fato da vida e de que a chave para sua melhoria é detectá-los e corrigí-los o mais rápido e sem custos possível. Isto naturalmente significa testar. Contudo, a proposta de métodos formais é mover a ênfase da detecção e remoção de defeitos para a prevenção dos mesmos.

O terceiro ponto diz respeito ao fato dos requisitos informais *versus* especificações formais e tem bastante relevância no trabalho atual. Um método formal deve iniciar com uma especificação formal. Entretanto, a prática convencional na indústria é de que eles se iniciam compilando-se uma especificação informal de requisitos. Este documento é tipicamente um importante documento contendo centenas ou milhares de páginas descrevendo o comportamento desejado e características do sistema. É preparado por uma parte interessada *stakeholder* crítica do projeto tal como um analista do negócio, especialistas do domínio, especialistas nos requisitos e clientes e reflete todo o conhecimento e experiência do negócio sobre o domínio do produto. Este documento não é entretanto estático e evolui ao longo do tempo.

A fim de se aplicar um método formal deve-se começar desenvolvendo-se uma especificação

formal a partir de uma especificação informal de requisitos e, durante o desenvolvimento, deve-se ser capaz de manter a especificação formal sincronizada com mudanças à especificação informal. Tendo feito isto, é necessário proceder ao teste de que a especificação formal descreve exatamente o mesmo sistema da especificação informal de requisitos.

Entretanto, é não realístico que (i) analistas de negócio, especialistas do domínio, especialistas em requisitos, clientes e usuários finais apresentem seus requisitos numa especificação formal; (ii) apresentar a estas partes interessadas uma especificação formal de tal maneira que elas não possam entendê-la e esperar que as partes façam a verificação citada no parágrafo anterior; (iii) esperar que tal especificação seja aceita como base para um acordo contratual; ou (iv) requerer que as partes interessadas seja treinadas em métodos formais. Em resumo, as únicas pessoas com conhecimento do domínio do problema necessário para fazer a verificação do parágrafo anterior, ou seja, a verificação da especificação formal não podem fazê-lo porque elas não o entendem.

Portanto, Broadfoot aponta um hiato entre esses requisitos informais e a especificação formal, a qual para ser aceita, precisa ser validada. O presente trabalho visa suprir tal necessidade provendo um meio informal de se especificar requisitos numa linguagem como a UML e, posteriormente traduzir o modelo automaticamente para a notação formal de Redes de Petri. Isso elimina a necessidade de algum parte interessada ter que realizar validações pois a passagem dos requisitos informais para a especificação formal é feita de maneira automática. O trabalho de Broadfoot também oferece alternativas nesse sentido (BROADFOOT, 2005). A Seção 2.6 detalha este trabalho.

Sommerville aponta ainda outras razões para a não popularização dos métodos formais (SOMMERVILLE, 2007): (i) Sucesso das disciplinas de engenharia de software; (ii) Mudanças no mercado nos anos 80; (iii) Escopo limitado dos métodos formais e (iv) Escalabilidade limitada dos métodos formais.

O uso de outros métodos de engenheraria de software tais como métodos estruturados, gerência de configuração e encapsulamento de informação (quando num contexto de uma linguagem orientada a objetos) nos processos de projeto de software resultaram em melhorias na qualidade do

software. As pessoas que sugeriram que a única maneira de se melhorar a qualidade do software era usando-se métodos formais estavam claramente enganadas.

Mudanças de mercado nos anos 1980 promoveram a qualidade de software como o problema chave da engenharia. Entretanto, desde então, o problema crítico para muitas classes de desenvolvimento de software não é qualidade mas *time-to-market*. O software deve, nestes casos, ser desenvolvido rapidamente. Técnicas para rápido desenvolvimento de software não funcionam adequadamente com especificações formais, ainda segundo Sommerville. Naturalmente, a qualidade é ainda um fator importante, mas ela deve ser alcançada num contexto de entrega rápida do produto de software.

Sobre o escopo limitado dos métodos formais, Sommerville afirma que eles não são bem adaptados para especificar interfaces de usuário e interação com os mesmos. Sobre o problema da escalabilidade dos métodos formais, o autor afirma que a maioria dos projetos em que são aplicados são projetos de núcleo ou *kernel* relativamente pequenos e críticos. À medida que o sistema aumenta de tamanho, o tempo e esforço requerido para desenvolver-se uma especificação formal aumenta disproporcionalmente.

Particularmente, a verificação formal não pode ser considerada como a solução definitiva de todos os problemas e a garantia final da corretude. Ela está também sujeita a muitas das limitações da tradicional prática de teste de software, a saber, que o teste não pode provar a total ausência de erros, mas apenas a presença ou não de alguns específicos (LAPLANTE, 2004).

Ademais, a evolução da notação é lenta, apesar de contínua, na comunidade de métodos formais, segundo Laplante (LAPLANTE, 2004). Pode levar vários anos a partir de quando a notação foi criada até a mesma ser adotada na indústria. Possivelmente o maior desafio na aplicação de métodos formais no domínio de sistemas embarcados e de tempo real é escolher apropriadamente a técnica para aplicar ao problema (LAPLANTE, 2004). E, para tornar os métodos formais usáveis por um grande espectro de pessoas, os requisitos devem usam uma ou mais notações não matemáticas, tais como linguagens naturais, texto estruturado ou diagramas (LAPLANTE, 2004). Essa

mesma preocupação foi manifestada por Sommerville, conforme descrito nos parágrafos anteriores. Mais uma vez, o método proposto no presente trabalho visa trazer alternativas nesse sentido.

Entretanto, as organizações que têm feito o investimento em métodos formais e perseverado em seu uso têm reportado menos erros no produto de software final, sem um aumento nos custos do desenvolvimento, o que é melhor. Parece que métodos formais podem ser efetivos no quesito custo se seu uso for limitado a partes centrais do sistema e se a companhia estiver disposta a fazer um alto investimento inicial nesta tecnologia (SOMMERVILLE, 2007). A relação custo-benefício da aplicação dos métodos formais está cada vez mais se tornando viável (DAVIS, 2005).

3.5.5 Verificação de Modelos - *Model Checking*

A Verificação de Modelos ou *Model Checking* é a abordagem mais bem sucedida que emergiu no que diz respeito à verificação de modelos. A idéia essencial atrás de uma ferramenta de *model checking* é receber os requisitos do sistema ou seu projeto (modelos) e uma propriedade (chamada especificação) que o sistema resultante deve satisfazer. A ferramenta então gera como saída "SIM" se o dado modelo satisfaz as dadas especificações e gera um contra-exemplo caso contrário. Este contra-exemplo detalha porque o modelo não satisfaz a especificação (KATOEN, 1999). Estudando-se o contra-exemplo, o usuário da ferramenta pode identificar o erro no modelo, corrigí-lo e verificá-lo então novamente. Assegurando-se que o modelo satisfaça suficientemente as propriedades do sistema, pode-se aumentar o grau de confiabilidade da corretude do modelo. Os requisitos do sistema são chamados modelos porque eles representam os requisitos ou projeto.

Uma ferramenta de *Model Checking* conhecida como SMV é utilizada no presente trabalho para verificar a corretude de modelos na notação Rede de Petri. As próximas Subseções mostrarão as vantagens dessa técnica e que justificam a sua adoção aqui.

Benefícios do *Model Checking*

A seguir, um resumo dos principais benefícios da técnica de *Model Checking*, de acordo com Katoen (KATOEN, 1999):

- Abordagem geral com aplicações para verificação de hardware, engenharia de software, sistemas multi-agentes, protocolos de comunicação, sistemas embarcado e dai por diante.

- Suporta verificação parcial: um projeto pode ser verificado contra uma especificação parcial, considerando-se apenas um subconjunto de todos os requisitos. Isso pode resultar em eficiência melhorada, desde que o usuário desta técnica possa restringir a validação para verificar apenas os mais importantes requisitos ignorando os menos importantes.

- Estudos de caso têm mostrado que a incorporação de *model checking* no processo de projeto não o atrasa mais do que as técnicas de Simulação e de Testes o fazem. Em muitos estudos, o uso da técnica de *model checking* levou a um tempo menor de desenvolvimento até. Ademais, graças a várias técnicas modernas, os verificadores de modelo ou *model checkers* são capazes de lidar com espaços de estados maiores (um exemplo com 10^{130} estados foi reportado na literatura).

- Os *model checkers* podem potencialmente ser usados tão rotineiramente quanto os compiladores pelos projetistas visto que também não exigem tanta interação com o usuário.

- Crescente interesse por parte da Indústria: empregos são oferecidos para aqueles que têm proficiência no uso das técnicas de *model checking*. Além do mais, a Indústria tem construído seus próprios *model checkers*. Exemplos incluem os da Siemens e Lucent Technologies. Outras companhias que criaram grupos de pesquisa nesta área ou tem usado largamente as ferramentas incluem Intel, Cadence, Fujitsu e Dutch Railways para mencionar algumas.

- Forte fundamento matemático: modelagem, semântica, Teoria de Concorrência, Teoria de

Autômatos, Estrutura de Dados, Algoritmos de Grafos etc constituem a base de *model checking*.

Limitações do *Model Checking*

A seguir, um resumo das principais limitações da técnica de *Model Checking*, de acordo com Katoen (KATOEN, 1999):

- É menos adequado para aplicações com uso intensivo de dados, as chamadas *data-intensive applications*. Isto porque o tratamento de dados usualmente introduz espaços de estados infinitos.

- A aplicação do *model checking* está sujeita a questões de decidibilidade: para alguns casos particulares - como a maior parte dos sistemas de estados infinitos - o *model checking* não é eficientemente computável. Verificação Formal entretanto é, em princípio, aplicado a estes sistemas.

- Usando-se *model checking*, um modelo do sistema é verificado ao invés do sistema real em si. O fato de um modelo ter certas propriedades não implica que a realização do mesmo também as terá. Por esse propósito, técnicas complementares tais como testes sistemáticos são necessárias.

- Apenas os requisitos declarados são verificados: não há garantias da completude das propriedades desejadas.

- Encontrar uma abstração apropriada (tal como o modelo do sistema e propriedades apropriadas em lógica temporal) requer alguma experiência.

- Como qualquer ferramenta, o software de *model checking* pode não ser confiável. Entretanto neste caso o impacto não é tão grande visto que as ferramentas de *model checking* se baseiam em algoritmos padrão e bem conhecidos.

- É possível (em geral) verificar-se generalizações com *model checking*. Se, por exemplo, um protocolo, após verificado, seja considerado correto para 1, 2 ou 3 processos usando-se *model checking*, não é possível prover uma resposta para n processos, sendo n arbitrário. Apenas para casos particulares, isto é factível. *Model checking* pode, entretanto, sugerir teoremas para parâmetros arbitrários que, subsequentemente, podem ser verificados usando-se alguma técnica de verificação formal.

Ainda segundo Katoen, não é possível corretude absoluta com a técnica de *model checking* para sistemas de tamanho realístico. Porém, apesar de tais limitações, esta técnica pode prover um aumento significativo no nível de confiança no sistema.

Visto que no *model checking* é muito comum modelar-se o possível comportamento do sistema como autômato de estado finito, a técnica é inerentemente vulnerável ao problema prático de que o número de estados pode exceder o total de memória disponível do computador. Isto é em particular um problema para sistemas distribuídos e paralelos que podem ter muitos estados possíveis - o tamanho do espaço de estados de tais sistemas é, no pior caso, proporcional ao produto do tamanho do espaço de estados de seus componentes individuais. Este problema é conhecido como "Problema da Explosão do Espaço de Estados" A Subseção 3.5.5 detalha melhor tal problema.

Problema da Explosão do Espaço de Estados

As ferramentas de *Model Checking* enfrentam um problema de explosão combinatorial do espaço de estados, comumente conhecida como problema da explosão de estados. Tal problema deve ser resolvido a fim de se tornar viáveis vários problemas reais. Existem várias abordagens para se combater este problema. A seguir, um resumo das mesmas.

Algoritmos simbólicos não constroem o grafo de alcançabilidade para o autômato; ao invés disso, eles representam tal grafo implicitamente usando-se uma fórmula em lógica propocional. O uso de Diagramas de Decisão Binários (*Binary Decision Diagrams* - BDDs) tornou-se popular pelo trabalho de Ken McMillan. Algoritmos de Verificação de Modelos Limitados (*Bounded Model*

Checking) descrevem o autômato em termos de um número fixo de passos k e verificam se a violação de propriedade pode ocorrer em k ou menos passos. Isto tipicamente envolve codificar o modelo restrito como uma instância do problema SAT. O processo pode ser repetido com valores cada vez maiores de k até que todas as possíveis violações tenham sido corrigidas. Os diagramas de decisão binários (BDDs), introduzidos por McMillan (MCMILLAN, 1993) são utilizados na ferramenta de *model checking* do presente trabalho, o SMV .

Outra técnica é a de Redução de Ordem Parcial, o qual pode ser usada (em grafos explicitamente representados) para se reduzir o número de intervalos independentes de processos concorrentes que necessitam ser considerados. A idéia básica é que se não importa, para o conjunto de coisas que se pretende provar, se A ou B executado primeiro, então é perda de tempo considerar-se tanto intervalos de AB e BA.

Uma terceira técnica é a abstração. Ela tenta provar propriedades num sistema por meio de uma simplificação prévia. O sistema simplificado normalmente não satisfaz exatamente as mesmas propriedades que o original de tal forma que um processo de refinamento faz-se necessário. Geralmente, alguém requer que a abstração seja válida (as propriedades provadas para o sistema simplificado sejam também para o original); entretanto, mais frequentemente, a abstração não é completa (nem todas as propriedades verdadeiras do sistema original o são também na abstração). Um exemplo de abstração é, num programa, ignorar-se os valores de variáveis não booleanas e considerar-se apenas variáveis booleanas e o fluxo de controle do programa; tal abstração, apesar de parecer vaga, pode de fato ser suficiente para provar por exemplo propriedades de exclusão mútua.

Enfim, não existe uma técnica perfeita, ou seja, eficiente para todos os tipos de sistemas para combater o problema da explosão de estados. Combinar técnicas, na maioria das vezes, traz bons resultados (AMARAL, 2004).

3.5.6 Verificação através de CTL

A Lógica da Árvore de Computação, do Inglês *Computation Tree Logic* (CTL) , é um tipo de Lógica Temporal de Ramificações introduzida por Clarke e Emerson e usada em uma maneira um pouco modificada por Quielle e Sifakis para fins de *model checking*. Três operadores temporais são usados, a saber:

- E - pronuncia-se "para algum caminho"
- A - pronuncia-se "para todos os caminhos"e
- U - pronuncia-se "até"

Os operadores de caminho são:

- X - na próxima oportunidade
- F - eventualmente
- G - continuamente

Considere o modelo básico de computação das estruturas de Kripke, qual seja $K = (S, I, \delta, AP, L)$, onde:

S	Estados do sistema
$I \subseteq S$	Estados iniciais
$\delta \subseteq S$ x S	Relações de transição
AP	Proposições atômicas sobre os estados
$L : S \rightarrow 2^{AP}$ (rótulos)	Função de rotulamento

A Tabela 3.2 apresenta um resumo das estruturas de ramificação e modos temporais da CTL.

Dessa forma, a validade do sistema é dada por:

$K \models \varphi$ sse $K, S \models \varphi$ para todo $s \in I$, sse $I \subset \|\varphi\| K$

tipo	fórmula φ	$K, S_0 \models \varphi$ sse ...
atômica	$p \in AP$	p é verdade de S_0
proposicional	$\neg\varphi$	$K, S_0 \not\models \varphi$
	$\varphi \vee \psi$	$K, S_0 \models \varphi$ ou $K, S_0 \models \psi$
temporal	$EX\varphi$	Existe um caminho $S_0S_1 \ldots S.t.K, S_1 \models \varphi$
	$AF\varphi$	Para todos os caminhos $S_0S_1 \ldots$ existe $i \in Ns.t.K, S_i \models \varphi$
	$\varphi EU\psi$	Existe um caminho $S_0S_1 \ldots$ e $i \in Ns.t.K, S_i \models \psi$ e $K, S_j \models \varphi$ para todo $0 \leq j < i$
	$AX\varphi, EF\varphi, \ldots$	similar

Tabela 3.2: Estruturas de Ramificação e Modos Temporais

3.5.7 SMV

O SMV (acrônimo para Symbolic Model Checker) é uma ferramenta para verificar sistemas de estados finitos contra especificações em lógica temporal CTL. A linguagem de entrada do SMV foi projetada para permitir uma descrição de sistemas de estados finitos que variam dos totalmente síncronos aos completamente assíncronos, e dos detalhados aos abstratos. Alguém poderia prontamente especificar um sistema como uma máquina de Mealy síncrona ou como uma rede de processos abstratos e não determinísticos. A linguagem provê a possibilidade de descrições modulares hierárquicas e a definição de componentes reusáveis. Partindo do princípio de que a SMV é direcionado para se descrever máquinas de estado finitas, os únicos tipos de dados na linguagem são os finitos, quais sejam: Booleanos, escalares e arranjos fixos. Estruturas de dados estáticas também podem ser construídas.

O propósito primário da linguagem de programação do SMV é descrever a relação de transição de uma estrutura de Kripke finita. Qualquer expressão em cálculo proposicional pode ser usada para determinar esta relação. Isto provê uma grande quantidade de flexibilidade e, ao mesmo tempo, um certo perigo de inconsistência. Por exemplo, a presença de contradição lógica pode resultar em um *deadlock*. Isto poderia tornar certas especificações verdadeiros sem razão ao mesmo

tempo que tornar a descrição não implementável. Enquanto o processo de *model checking* pode ser usado para verificar-se *deadlocks*, é melhor evitar-se o problema quando possível por meio do uso de uma descrição de estilo mais restrito. O SMV suporta isso através de sua sintaxe de atribuição paralela. Outras ferramentas notáveis de *model checking* incluem: SPIN (HOLZMANN, 2004), NuSMV (NuSMV, 2009), MARIA (MARIA, 2009) (com determinadas extensões), PROD, MURφ e Model Checking Kit (MCK) (MCK, 2009).

3.6 Resumo do Capítulo

Este Capítulo trouxe a fundamentação teórica necessária ao desenvolvimento do método proposto no presente trabalho. O desenvolvimento de software embarcado ganha em corretude e o produto final, em confiabilidade, com as técnicas formais aqui apresentadas. Também há um ganho em agilidade no processo de desenvolvimento com a utilização do diagrama de sequência da UML.

O próximo Capítulo discorre a respeito do método desenvolvido no presente trabalho para capturar e verificar formalmente requisitos de softwares embarcados. Ele também mostra o desenvolvimento da ferramenta do trabalho. Esta ferramenta realiza integralmente o método proposto.

Capítulo 4

Método Proposto

Este capítulo descreve o método proposto durante o período de trabalho desta dissertação, fruto das pesquisas e levantamentos bibligráficos. O capítulo está dividido em 6 seções, quais sejam: Motivação, Descrição do Método, Ferramenta, Formato dos Arquivo, Pontos Fortes e Limitações. A primeira seção, Motivação, fornece as razões, dentro do contexto do desenvolvimento de software embarcado, para a elaboração do método proposto. A segunda seção, Descrição do Método, traz os detalhes do método proposto procurando atender às necessidades discutidas na Motivação. A seguir, seção Ferramenta traz as funcionalidades do software desenvolvido que realiza o método proposto. O formato dos arquivos de entrada e de saída são comentados na Seção seguinnte. Por fim, têm-se os pontos fortes e as limitações do método proposto.

4.1 Motivação

A essência do projeto de sistemas embarcados é implementar-se um conjunto de funcionalidades ao mesmo tempo em que se satisfazem restrições sobre características como desempenho, custo, emissões, consumo de energia e peso (SANGIOVANNI-VINCENTELLI, 2001). A escolha da arquitetura de implementação determina se os projetistas implementarão uma funcionalidade como

um componente de hardware ou software executando em um Controlador Lógico Programável (CLP).

Nos últimos anos, as funções demandadas por sistemas embarcados têm crescido tanto em número e em complexidade que o tempo de desenvolvimento tem se tornado cada vez mais imprevisível e incontrolável (SANGIOVANNI-VINCENTELLI, 2001). Esta complexidade tem forçado os projetistas a considerar implementações intrinsicamente flexíveis - aquelas que podem mudar rapidamente (SANGIOVANNI-VINCENTELLI, 2001). Por esta razão e porque os ciclos de fabricação de hardware são mais caros e consomem mais tempo, as soluções baseadas em software tem se tornado mais populares. O crescente poder computacional e seu correspondente diminuição de tamanho e de preço permitiu aos projetistas mover cada vez mais funcionalidades para software.

Entretanto, juntamente com esta mudança, veio uma crescente dificuldade, devido a requisitos cada vez mais complexos, na verificação de corretude do sistema (SANGIOVANNI-VINCENTELLI, 2001). Todavia, a verificação é crítica devido devido a considerações de segurança em vários domínios - monitoramento de transportes e ambientes, por exemplo. Nas aplicações tradicionais de computadores de mesa, estas questões de segurança tipicamente não são consideradas. Como demonstrando na Subseção 3.2.3, metodologias tradicionais de desenvolvimento de software de mesa têm sido adaptadas para o contexto do desenvolvimento de software embarcado.

O tempo de lançamento de produto, do inglês *time-to-market*, para softwares embarcados têm diminuído bastante devido à forte concorrência em nível global. Isso exerce forte pressão sobre as organizações para lançar produtos cada vez mais rápido. Obviamente toda essa carga recai, em última análise, sobre os desenvolvedores. Até como resposta ao tempo de lançamento de um produto cada vez mais curto, métodos cada vez mais intuitivos são necessários para se especificar, projetar e verificar software embarcado. Métodos intuitivos são necessários mas ainda assim que garantam a confiabilidade do produto de software final. O emprego de métodos formais para tal garantia tem crescido bastante na indústria, como detalha a Seção 3.5, e a mesma tem procurado alternativas viáveis nesse sentido.

Essas foram as motivações gerais para este trabalho. Motivações mais específicas incluem investigar o potencial e explorar notações alternativas ao Diagrama de Máquina de Estados, o antigo *statecharts*, da UML na modelagem de software embarcado; prover uma saída em PNML para a Rede de Petri gerada e uma forma mais amigável de se formular consultas de propriedades de sistemas. Como o projetista utiliza uma notação em nível de abstração mais alto, que é o caso dos Diagramas de Sequência da UML, obviamente ele também irá requerer um meio mais amigável de se formular consultas, preferencialmente um meio mais próximo ao da linguagem natural, com a posterior tradução para a notação formal e mais adequada para conduzir a consulta.

4.2 Descrição do Método Proposto

O método proposto no presente trabalho pode ser resumido na Figura 4.1 é explicado através das Subseções subsequentes. A idéia do método é prover suporte à verificação formal de modelos em qualquer uma das atividades do processo de desenvolvimento, descritas em 3.2.2. Portanto, os passos do presente método podem ser aplicados em qualquer etapa do processo e são independentes de nível de abstração, o que o torna mais flexível.

Os passos apresentados no método procuram atender às necessidades apresentadas na Seção 4.1, de Motivação, e são totalmente dependentes da ferramenta apresentada em 4.3.

4.2.1 Especificação Informal

O primeiro passo no método proposto é modelar o software a ser desenvolvido numa notação semi-formal, portanto mais rápida e fácil de ser usada pelo projetista. O objetivo é evitar que o projetista tenha contato direto com formalismos, visto que nem sempre é garantia de ele ter proficiência com os mesmos, conforme explana 3.5.4. Isso visa atender à demanda por métodos cada vez mais ágeis e tempos de lançamento de produto cada vez menores, conforme descreve 4.1. A Unified Modeling Language (UML) foi escolhida como tal notação por ser atualmente um padrão *de facto*

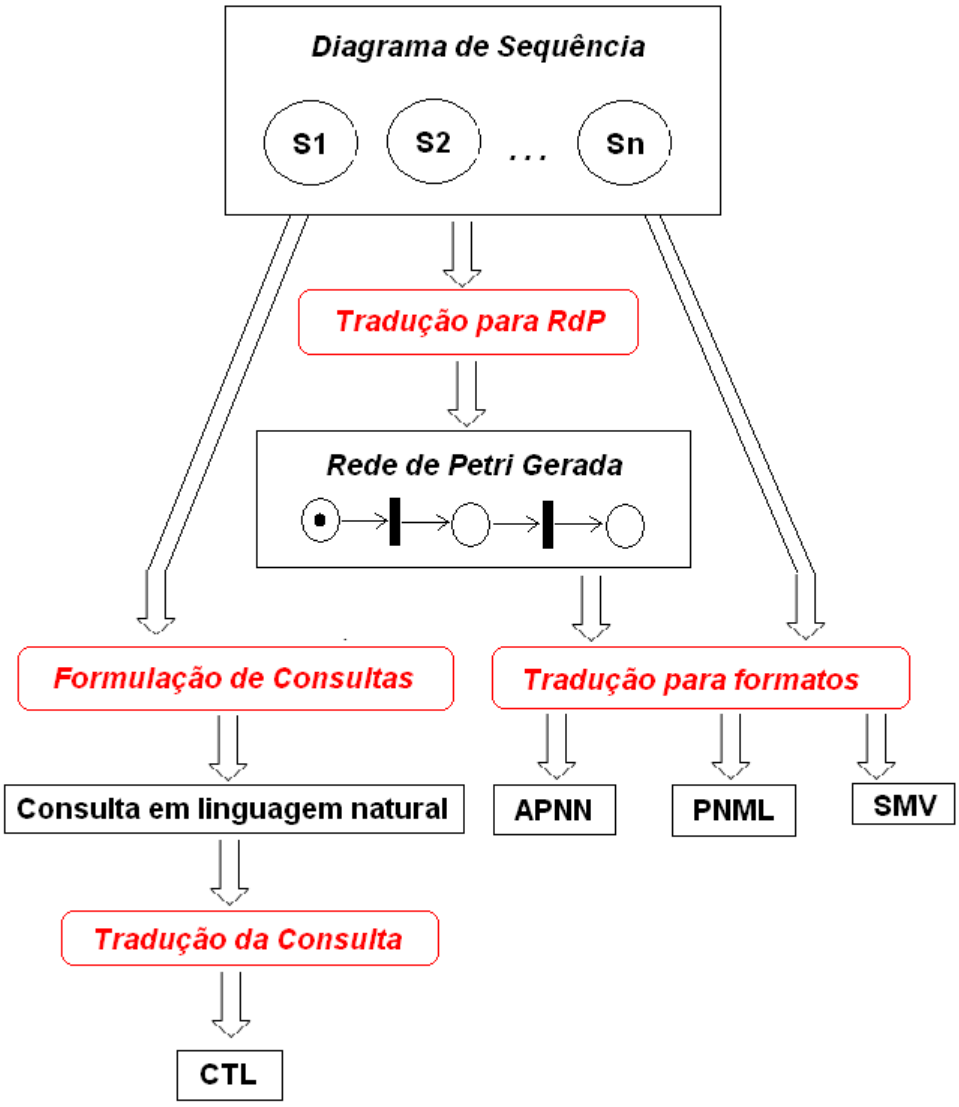

Figura 4.1: Métodos Proposto

da indústria e academia, o que facilita a aceitação do método proposto.

Há ainda outras vantagens no uso da UML como o fato de ser orientada a objetos e de ser uma linguagem de propósito geral, não sendo restrita a um domínio ou outro (ver 3.3.1). Isso traz a possibilidade de a mesma ser usada também no contexto de software embarcado.

A modelagem em UML, neste passo, é feita usando-se especificamente a notação de seu diagrama de sequência , apto a prover uma visão dinâmica do sistema em questão. Esta notação também foi escolhida pois a maioria dos trabalhos correlatos pesquisados, no melhor de nosso conhecimento, utilizam a notação de Máquinas de Estados, antigo *statecharts*, da UML para esta modelagem inicial e usar diagramas de sequência constituiu-se portanto em um desafio. Este passo está mapeado na Figura 4.1 na caixa com o texto "Diagrama de Sequência" a qual mostra também os objetos relevantes do domínio.

Esta especificação ou modelagem inicial informal toma a notação de Jeng (JENG, 2002) como auxílio ou complemento para melhor expressar paralelismo e escolha. A notação de Jeng é detalhada na Seção 2.1 A entrada desta atividade são os objetos oriundos diretamente dos requisitos do software, de uma modelagem arquitetural, de um projeto de mais alto nível ou detalhado. Por isso, a caixa com o texto "Diagrama de Sequência" da Figura 4.1 contém os objetos relevantes do domínio, sejam eles mais abstratos ou mais concretos, no nível de processos e tarefas por exemplo, como detalha o experimento no Capítulo 5. O cerne desta atividade é modelar objetos relevantes do domínio no nível de abstração desejado e prover a entrada para o próximo passo.

4.2.2 Tradução Automática para Modelo Formal

Continuando com o objetivo de eximir o projetista de lidar diretamente com notações e métodos formais, segue a tradução automática do diagrama de sequência para uma notação formalmente rigorosa, no caso, a do *model checker* SMV, de mesmo nome. Como dito anteriormente, de posse de um modelo formalmente especificado, é possível fazer uso de algum método de verificação formal de software, no caso do presente método, o *model checking*. Todavia, todo este trabalho é

```
ALGORTIMO Tradução

 VAR
  DS : Diagrama de Sequência;
  SMV : Programa em SMV;

 Para cada objeto de DS
  Adiciona-o como variável em SMV e seus precedentes com base em DS;
 Fim-Para

 Para cada variável de SMV
  Escreve-a no programa final com seus precedentes;
 Fim-Para

FIM-ALGORTIMO
```

Tabela 4.1: Algoritmo de Tradução

```
next(A) :=
case
 A = 0 & B = 1 & C = 1 :  1;
 1 :  A;
esac;
```

Tabela 4.2: Exemplo de Tradução

automático e transparente ao projetista. Outro problema evitado ou minimizado é o conhecido *gap* de, a partir de uma especificação de software, posteriormente ter que convertê-la para um projeto de software. Este processo de tradução está sujeito a todas as potenciais armadilhas conhecidas.

Algoritmo de Tradução

O algortimo de tradução é apresentado na Tabela 4.1 do diagrama de sequência para o formato SMV.

Por exemplo, a Tabela 4.2 mostra um objeto A do Diagrama de Sequência, tendo como precedentes os objetos B e C teria o seguinte bloco resultante no programa SMV final:

Percebe-se que o objeto A não deve ter executado ainda e, tendo sido executado, seu valor não se altera. Maiores informações sobre a notação, favor verificar a Seção 3.5.7.

4.2.3 Verificação Formal

A verificação formal, conforme detalhado na Seção 3.2, não está restrita a ser usada apenas em uma atividade do processo de desenvolvimento mas virtualmente em qualquer uma delas. Também nesta Seção muito foi comentado a respeito da importância dessa atividade. Não é intenção deste trabalho escrever uma ferramenta de *model checking*, mas a ferramenta para tal finalidade usada é o Symbolic Model Verifier ou SMV. Ele foi originalmente desenvolvido na Universidade Carnegie Mellon, nos EUA, e está atualmente disponível como uma ferramenta de pesquisa no *web site* dos Cadence Berkeley Laboratories, em Berkeley, estado da Califórnia, também nos EUA.

4.2.4 Formulação de Consultas

Não faria sentido permitir a especificação do software em um nível de abstração mais alto e solicitar ao projetista / usuário da ferramenta um conhecimento mais aprofundado do formalismo da execução das consultas, no caso, o Computational Tree Logic - CTL. Levando-se isso em consideração, o método apresenta a etapa de tradução de sentenças em linguagem natural para o formalismo da CTL.

A sentença para expressar os caminhos de execução são "‘em todos os caminhos de execução"’ equivalente ao operador A. A expressão "‘em alguns caminhos de execução"’ é equivalente ao operador E. A expressão "‘do primeiro ao último cenário"’ equivale ao operador "‘G"’. A sentença "‘em algum cenário à frente"’ equivale ao operador "‘F"’ e a expressão "‘necessariamente no próximo cenário"’, ao operador X. Com bases nestas expressões, as consultas em CTL são geradas.

4.3 Ferramenta Desenvolvida

Como dito anteriormente, a ferramenta de software desenvolvida nesta dissertação realiza integralmente o método proposto na Seção 4.2 o qual depende inteiramente dela para o seu sucesso.

Procurou-se escrever um software intuitivo, de fácil uso e que proporcionasse uma ótima experiência ao usuário. A linguagem escolhida para a codificação do software foi Java por suas características de orientação a objetos, portabilidade e diversidade da biblioteca.

A especificação de entrada é apresentada no formato XML. O formato do arquivo de entrada é apresentado na Subseção 4.4.1 e na Tabela 4.3. Tal especificação pode também ser gerada por uma ferramenta visual de Engenharia de Software Auxiliada por Computador (*Computer-Aided Software Engineering* - CASE). Esta especificação é lida no AVFSE, servindo como entrada para o mesmo.

A figura 4.1 apresenta um passo a mais não comentado nas Seções anteriores. É o passo "'Tradução para RdP"' que gera uma Rede de Petri, a partir do diagrama de sequência da entrada. A idéia é traduzir o modelo em alto nível, na notação da UML, para um modelo mais rigoroso, na notação Redes de Petri. As regras para tradução são baseadas em algumas apresentadas por Jeng (JENG, 2002). Naquele trabalho, tomava-se como entrada um diagrama de sequência estendido da UML e, através de composição de blocos, chegava-se à Rede de Petri Lugar/Transição final. O trabalho de Jeng e as cinco regras de tradução nas quais este passo da ferramenta se baseia são apresentados na Seção 2.1.

De posse da rede de Petri traduzida pode-se gerar um arquivo no formato Abstract Petri Net Notation - APNN (APNN, 2008), que pode ser usado como entrada para o *model checker* Model Checking Kit (MCK) (MCK, 2008) ou Petri Net Markup Language - PNML (PNML, 2008), conforme também coloca a figura.

Por fim, através de uma interface gráfica especial, o Ambiente de Verificação Formal de Software Embarcado permite ao projetista especificar, em linguagem natural, as consultas de propriedades a serem realizadas sobre o software embarcado em questão. A fórmula é então traduzida automaticamente para o formato CTL, adequado a ser usado em algum verificador de modelos (model checker) tal como o SMV (SMV, 2008) ou o MCK (MCK, 2008). Como foi dito, isso facilita o processo de desenvolvimento do software embarcado, livrando o projetista da obrigação

de dominar notações formais e agilizando todo o processo.

4.4 Formato dos Arquivos

A ferramenta descrita em 4.3 utiliza vários arquivos em diferentes formatos como entrada ou saída. Nas próximas seções, a descrição do formato dos mesmos.

4.4.1 Diagrama de Sequência

A seguir, a Tabela 4.3 mostra o formato do arquivo de entrada na ferramenta contendo o diagrama de sequência tendo as tarefas de aplicação como objetos.

A notação do diagrama de sequência é bastante trivial não exigindo nenhum conhecimento novo do projetista em relação à UML. Após uma breve descrição da aplicação contida na *tag* `aplicacao`, seguem-se as *tags* `tarefa` contendo como valor cada uma das tarefas de aplicação do sistema. A seguir, a *tag* `mensagem` que expressa a troca de mensagens entre os objetos relevantes do domínio, no caso as tarefas descritas anteriormente. Dentro de cada mensagem, há uma *tag* `de` com a tarefa de origem e uma `para`, com a de destino.

4.4.2 Arquivo APNN

A ferramenta gera um arquivo APNN (Abstract Petri Net Notation), um formato de arquivo capaz de ser usado no Model Checking Kit (MCK). Não será mostrada toda a descrição detalhada do mesmo bem como dos outros dois formatos de arquivos que vem a seguir. Para uma melhor descrição do APNN, favor referir-se a (APNN, 2008). A seguir, na Tabela 4.4, o formato básico de um arquivo APNN.

```
<?xml version="1.0"encoding="UTF-8"?>

<diagramadesequencia>

 <aplicacao>Descrição da Aplicação</aplicacao>

 <objetos>
  <objeto>
   <nome>Nome do Objeto de Domínio 1</nome>
   <classe>Classe do Objeto</classe>
  <objeto>
  <objeto>
   <nome>Nome do Objeto de Domínio 2</nome>
   <classe>Classe do Objeto</classe>
  <objeto>
   .
   .
   .
  <objeto>
   <nome>Nome do Objeto de Domínio N</nome>
   <classe>Classe do Objeto</classe>
  <objeto>
 </objetos>

 <mensagens>
  <mensagem>
   <nome>Nome da Mensagem</nome>
   <de>Tarefa de Origem</de>
   <para>Tarefa de Destino</para>
   <ordem>Ordem da Mensagem</ordem>
  </mensagem>
 </mensagens>

 <concorrencias>
  <concorrencia>
   <mensagens>
    <elemento>Objeto concorrente 1</elemento>
    <elemento>Objeto concorrente 2</elemento>
   </mensagens>
  </concorrencia>
 </concorrencias>

 <sincronizacoes>
  <sincronizacao>
   <mensagens>
    <elemento>Objeto em sincronização 1</elemento>
    <elemento>Objeto em sincronização 2</elemento>
   </mensagens>
  </sincronizacao>
 </sincronizacoes>

</diagramadesequencia>
```

Tabela 4.3: Formato do Diagrama de Sequência

```
\beginnet{ID-REDE}

\place{ID-LUGAR} {
 \name{NOME-LUGAR}
 \init{MARCAÇÃO-INCIAL}
 \capacity{CAPACIDADE}
}

\transition{ID-TRANS} {
 \name{NOME-TRANS}
}

\arc{ID-ARCO} {
 \from{ID-ORIGEM}
 \to{ID-DESTINO}
 \weight{PESO}
 \type{TIPO}
}

\endnet
```

Tabela 4.4: Formato do Arquivo APNN

4.4.3 Arquivo PNML

O segundo formato de saída é o Petri-Net Markup Language (PNML, 2008). Este formato foi escolhido por causa de sua grande adoção em várias ferramentas e, portanto, pela ampla possibilidade de compatibilidade com as mesmas. Na Tabela 4.5, o formato do arquivo em questão.

4.4.4 Arquivo SMV

O formato SMV serve como entrada para o *model checker* de mesmo nome que é usada no Capítulo 5 para realizar consultas de propriedades do sistema. Ele é melhor descrito em (MCMILLAN, 1993). A Tabela 4.6 traz o formato do arquivo SMV.

A "variável" é um dos lugares gerados na rede de Petri. O lugar inicial na rede é "lugar_Start" e ele sempre terá um *token* inicialmente. A seguir, na seção SPEC, tem-se uma fórmula CTL traduzida a partir da interface de formulação de consultas em alto nivel. Além do módulo principal `main`, há um módulo para cada processo. Cada um destes módulos contém no mínimo uma variável, que é uma tarefa de aplicação do sistema. Em cada um destes módulos, novos valores das variáveis são atribuídos dentro da construção "next".

```
<?  xml version="1.0" encoding="ISO-8859-1" ?>

<pnml xmlns="XMLNS">

 <net id="ID-REDE" type="URI">

 <name>
  <text>NOME-REDE</text>
 </name>

 <transition id="TRANS-ID">
  <graphics>
   <position x="0" y="0" />
  </graphics>
  <name>
   <text>TRANS-NOME</text>
   <graphics>
    <offset x="0" y="0" />
   </graphics>
  </name>
 </transition>

 <arc id="ARCO-ID" source="ID-ORIGEM" target="ID-DESTINO">
  <inscription>
   <text>PESO</text>
   <graphics>
    <offset x="0" y="0" />
   </graphics>
  </inscription>
  <graphics>
   <position x="0" y="0" />
  </graphics>
 </arc>

 <place id="LUGAR-ID">
  <graphics>
   <position x="0" y="0" />
  </graphics>
  <name>
   <text>LUGAR-NOME</text>
   <graphics>
    <offset x="0" y="0" />
   </graphics>
  </name>
  <initialMarking>
   <text>MARCAÇÃO-INICIAL</text>
  </initialMarking>
 </place>

 </net>

</pnml>
```

Tabela 4.5: Formato do Arquivo PNML

```
MODULE main
VAR
 <variável> :  boolean;
 <processo N> :  process <nome do módulo> (<lista de variáveis>);

ASSIGN
 init(lugar_Start) := 1;
 init(<variável>) := 0;

SPEC
 <fórmula CTL>

MODULE <nome do módulo> (<lista de variáveis>)
ASSIGN
 next(<variável>) :=
  case
   <variável>=0 :  1;
   1 :  <variável>;
  esac;
```

Tabela 4.6: Formato do Arquivo SMV

4.5 Pontos Fortes

Eis um resumo dos pontos fortes do método proposto. O primeiro ponto forte é a robustez. Segundo (NASA, 2001), muitas abordagens são usadas para se lidar com os espaços de estados imensos associados com sistemas reais como aplicar-se métodos formais para requisitos e projetos de alto nível onde muitos dos detalhes são abstraídos. Este é o caso do presente método. Intutividade e, por consequência velocidade de desenvolvimento, é também uma caraterística do método proposto no sentido de que faz uso da notação da UML para a entrada de dados. O incremento da notação de Jeng aos diagramas de sequência traz mais expressividade no que diz respeito a paralelismo e escolha.

Outro aspecto da intuitividade é na confecção de consultas de propriedades com posterior tradução para a notação CTL. Outra característica do método atual é que ele pode considerar o sistema distribuído entre vários processadores e as tarefas se comunicando via um barramento compartilhado, por exemplo. Com isso ele suporta Múltiplas arquiteturas. No experimento citado no Capítulo sobre experimentos há quatro processadores em questão (ver 5). A verificação formal executada automaticamente constitui um aspecto de corretude.

4.6 Limitações do Método Proposto

O uso da UML ordinária sem considerar alguma extensão ou perfil pode ser uma limitação também no sentido de que A UML, por ser de propósito geral, é apta para virtualmente qualquer domínio, mas, por outro lado, deixa a desejar muitas vezes em domínios específicos como é o de software embarcado e, mais ainda, o de software de tempo-real. A Seção 3.3.6 descreve algumas dessas abordagens alternativas. Vale citar também que o método não considera o operador *Weak Until* da CTL, ou "‘W"’.

4.7 Resumo do Capítulo

O presente Capítulo elucidou o método de verificação formal de software embarcado proposto no trabalho. A ferramenta que realiza o método e ainda acrescenta outros aspectos foi mostrada. Foram detalhados seus pontos fortes e limitações também. O próximo Capítulo põe à prova o método proposto aqui através de um experimento de software embarcado de controle.

Capítulo 5

Experimento

Este Capítulo detalha um experimento de verificação formal de um software embarcado de controle, baseado no método descrito no Capítulo 4 e na ferramenta, o Ambiente de Verificação Formal de Software Embarcado construída para realizar integralmente o método.

5.1 Descrição do Experimento

O cenário levado em consideração nos experimentos é o de uma aplicação simples de controle, a *Simple Control Application*, descrita orginalmente em (DINATALE, 1994). Ela consiste basicamente num modelo de processos P_1, P_2, ..., P_n, onde cada processo é dividido em uma sequência de tarefas T_{i_1}, T_{i_2}, ..., T_{i_n}. As tarefas são unidades computacionais indivisíveis do ponto de vista de escalonamento, neste contexto.

O sistema consiste fisicamente em de um dispositivo sensorial montado sobre uma plataforma motorizada que deve detectar e rastrear objetos específicos em um ambiente. O sistema é controlado por quatro processadores conectados por meio de um barramento. O primeiro processador controla o sensor e executa dois processos: SENSOR e SENSOR_MNG. O segundo processador controla os atuadores e executa os processos ACT_CONTRL e ACTUATOR_MNG. Ambos se comunicam com outro processador rodando a principal atividade de controle MAIN_CONTROL e

são responsáveis pela detecção de eventos significantes detectados no ambiente. O último processador, rodando o processo SIGNAL, é responsável pelo gerenciamento de tais eventos significantes detectados no ambiente (e.g., a geração de alarme).

O fluxo de execução da aplicação é o seguinte: O processo MAIN_CONTROL comunica-se remotamente ao processo SENSOR_MNG o pedido por uma nova atividade. O SENSOR_MNG, localmente solicita dados sensoriais do processo SENSOR. Quando os dados sensoriais são retornados, o processo SENSOR_MNG extrai características significantes do objeto a ser rastreado e comunica de volta os resultados para o processo MAIN_CONTROL. A seguir, o MAIN_CONTROL envia a informação da posição para o processo ACTUATOR_MNG que se comunica com o processo ACT_CONTRL para realizar o rastreamento do objeto identificado e mantê-lo no campo de visão dos sensores. Ao mesmo tempo, o processo MAIN_CONTROL verifica a presença de eventos significantes no sistema e envia um relatório ao processador rodando o processo SIGNAL, responsável pela detecção e ativação dos alarmes e avisos. Ademais, todas as tarefas são caracterizadas pelas restrições temporais padrão, a saber, deadline, tempo de liberação, de computação etc. Entretanto, neste experimento, no processo de tradução de diagramas de sequência para rede de Petri, não serão consideradas tais restrições. Apenas as unidades computacionais mínimas, as tarefas.

5.2 Diagrama de Sequência da Entrada

Os diagramas de sequência, conforme exposto na Seção 3.3, capturam os aspectos dinâmicos do sistema, enfatizando a ordem temporal dos eventos no mesmo. As partes de diagrama das Figuras 5.1, 5.2, 5.3, 5.4, 5.5 e 5.6 modelam tais aspectos no contexto do *Simple Control Application*. Como é obrigação do projetista entrar com tais diagramas e no formato de entrada adequado (ver 4.4.1), de acordo com o método do presente trabalho detalhado na Figura 4.1, os mesmos serão apenas apresentados aqui e não explicados.

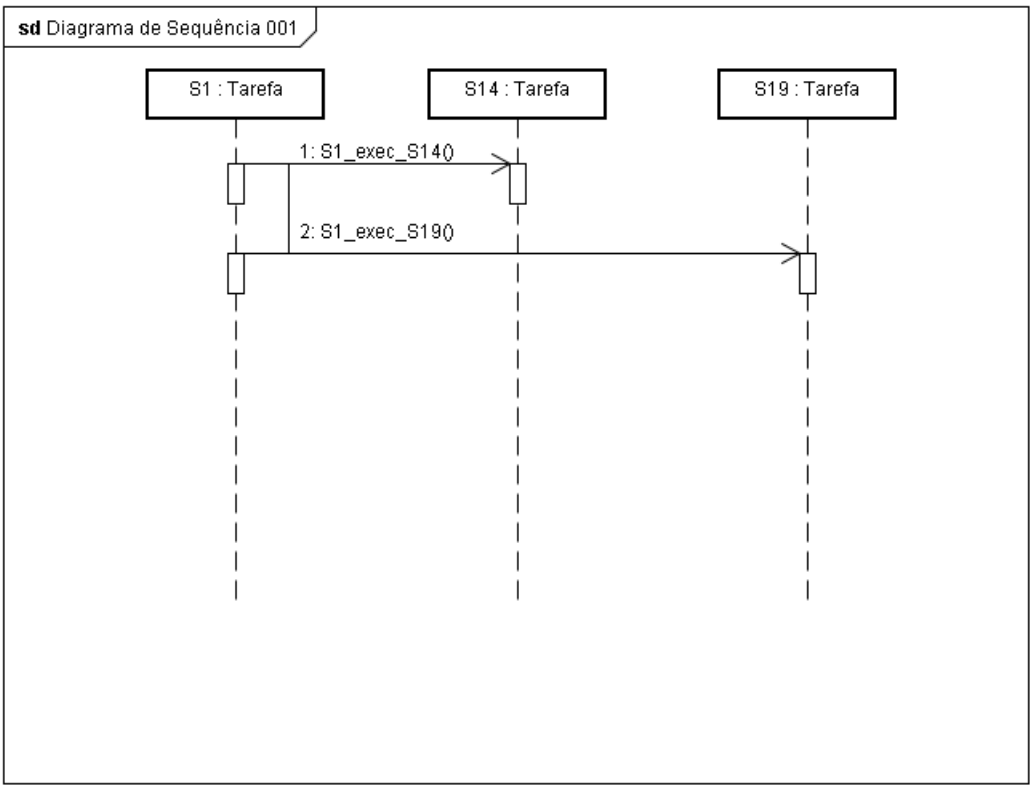

Figura 5.1: Diagrama de Sequência da Entrada. Parte 1 de 6

É válido porém dizer que, pela notação estendida introduzida na Seção 2.1, percebe-se a presença de várias linhas verticais entre mensagens e círculos sólidos caracterizando-se, portanto, situações de concorrência e de junção. Neste experimento, em particular, não há situações de escolha ou de confluência.

5.3 Utilizando a Ferramenta Para Realizar o Método

Esta Seção mostrará o passo a passo para se ter o software embarcado expresso no diagrama de sequência formalmente verificado. O primeiro passo é carregar o diagrama de sequência a partir do menu "Diagrama de Sequência", "Abrir".

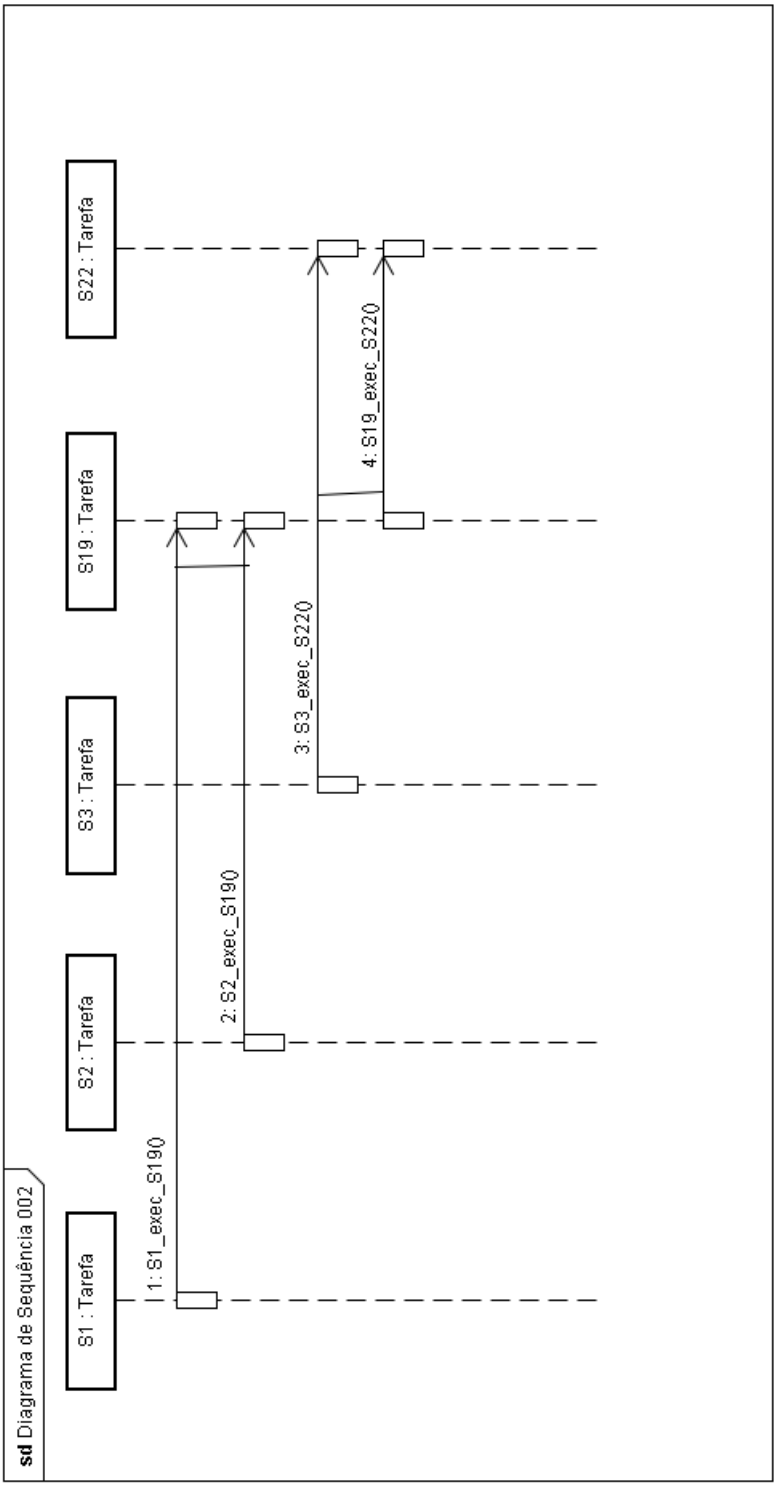

Figura 5.2: Diagrama de Sequência da Entrada. Parte 2 de 6

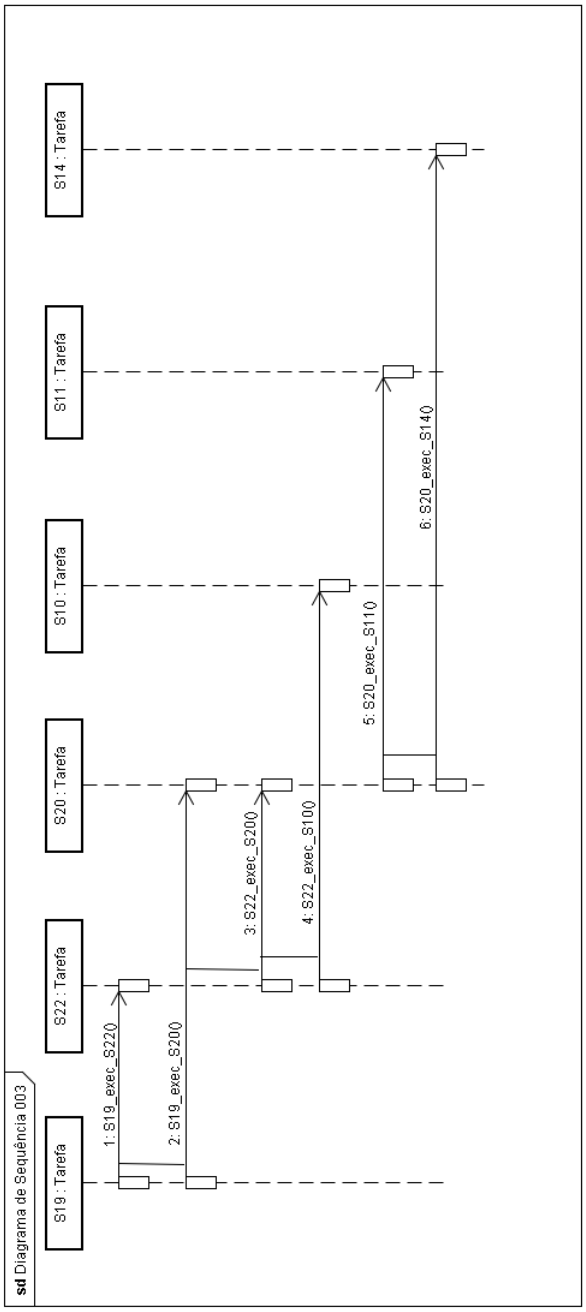

Figura 5.3: Diagrama de Sequência da Entrada. Parte 3 de 6

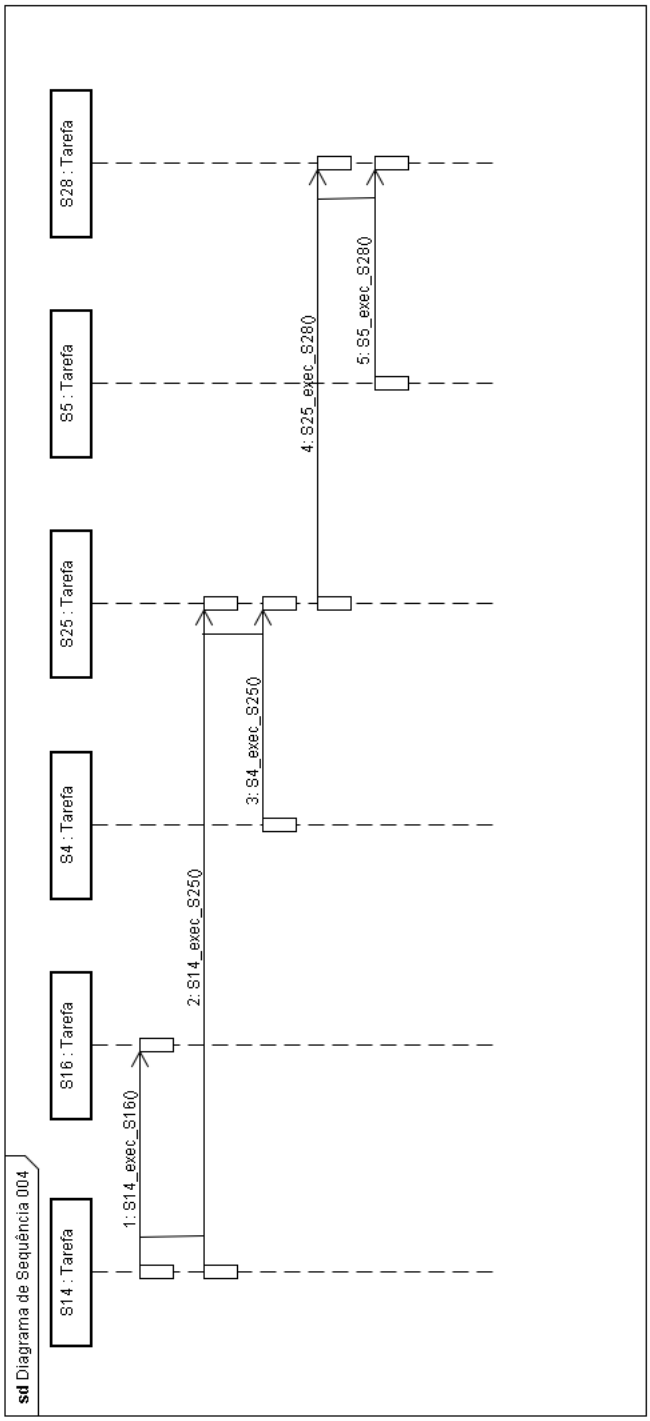

Figura 5.4: Diagrama de Sequência da Entrada. Parte 4 de 6

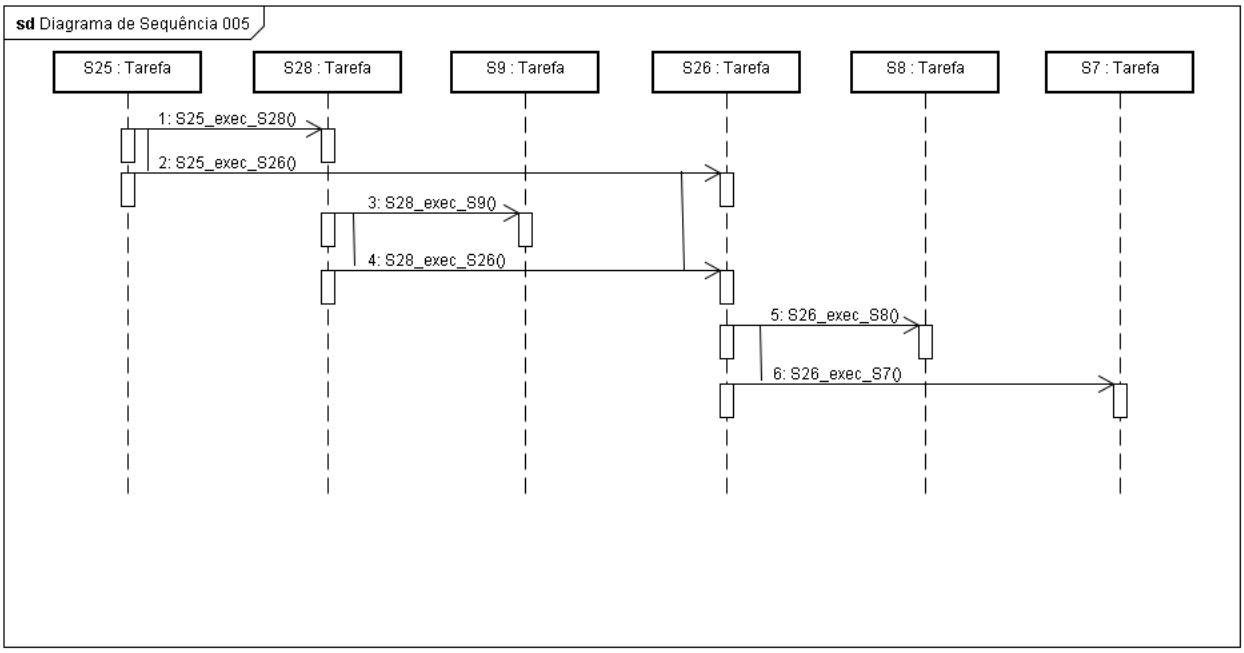

Figura 5.5: Diagrama de Sequência da Entrada. Parte 5 de 6

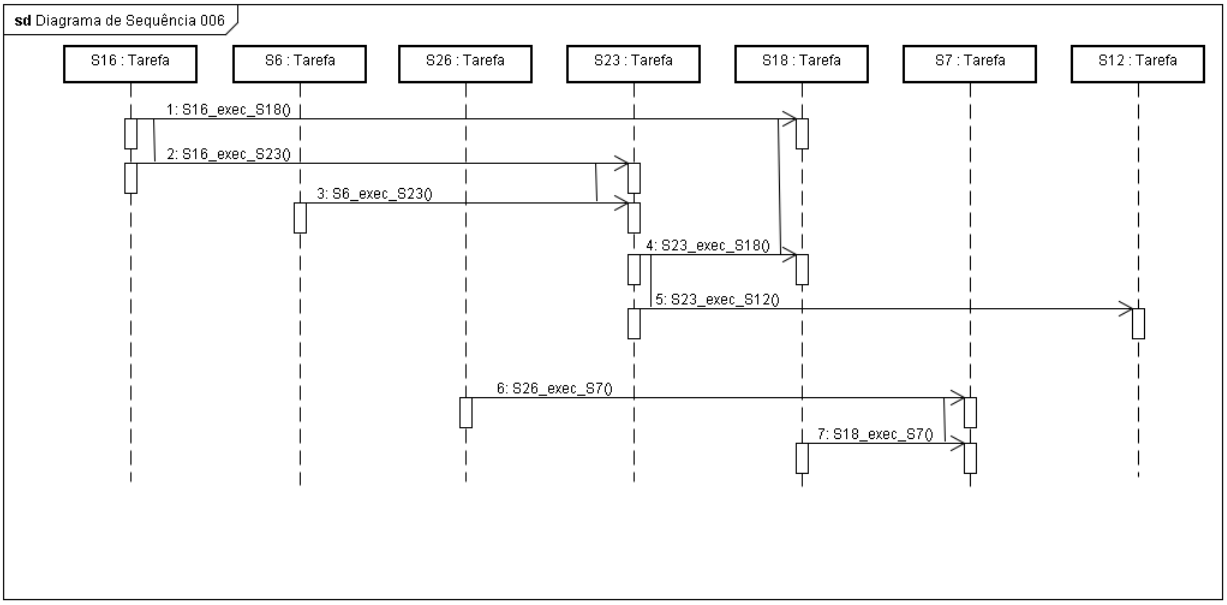

Figura 5.6: Diagrama de Sequência da Entrada. Parte 6 de 6

5.3.1 Tradução para Rede de Petri

Pode-se opcionalmente traduzir o diagrama de sequência de entrada para uma Rede de Petri num dos dois formatos de arquivos, previamente comentados, que são o PNML e o APNN. Para isto, basta acessar o Menu "Rede de Petri"e escolher uma das opções: "Traduzir para PNML"ou "Traduzir para APNN". Usando-se o software Snoopy 2.0, transforma-se a rede para o formato PNT, o que serve de entrada para o analisador Integrated Net Analyzer (INA). Por questões de espaço, a figura da Rede de Petri gerada não será mostrada aqui.

A seguir, as principais características da rede de Petri gerada, apenas citadas:

- Estaticamente livre de conflitos
- Dinamicamente livre de conflitos
- Pura
- Ordinária
- Homogênea
- Não-conservativa
- Não é uma máquina de estados
- Livre escolha
- Não viva
- Marcada com exatamente um *token*
- Limitada
- Estruturalmente limitada

- A rede não tem transições sem lugares antes ou sem lugares depois
- Conectada

5.3.2 Verificação Formal e Consultas de Propriedades

É necessário entender que os vários caminhos de execução possíveis, conforme especificado no AVFSE, constituem ramos de uma árvore de execução maior. E cada ramo possui vários nodos, os quais são os cenários de execução. De posse desse entendimento, fica mais claro a listagem dos contra-exemplos das consultas cujos resultados são falso. Tais listagens trazem uma sequência com os cenários, denominados lá "estados", que prova a falsidade da fórmula da propriedade em verificação, o que obviamente, torna a propriedade inválida para o sistema em questão.

Consulta I

A primeira consulta verifica se, em todos os caminhos de execução, do primeiro ao último cenário, o objeto $S1$ é executado. A Figura 5.7 mostra a tela do AVFSE e a fórmula CTL equivalente. Após pressionar-se o botão "Verificar", o diagrama de sequência é traduzido para a notação formal do SMV, apta a ser formalmente verificada através de *model checking* pela ferramenta SMV.

A fórmula "`AG ( S1 )`"é **FALSA**, conforme atesta a Figura 5.8 visto que é impossível que, em todos os caminhos de execução, em cada um de seus cenários, S1 seja executado. O contra-exemplo gerado pelo SMV demonstra que logo no primeiro cenário ou estado o objeto S1 possui valor zero ou simplesmente não é executado, invalidando a propriedade verificada nesta consulta.

Eis o *log* da execução no SMV para esta propriedade, juntamente com os recursos computacionais alocados:

```
-- specification AG S1 is false
-- as demonstrated by the following execution sequence
state 1.1:
S1 = 0
```

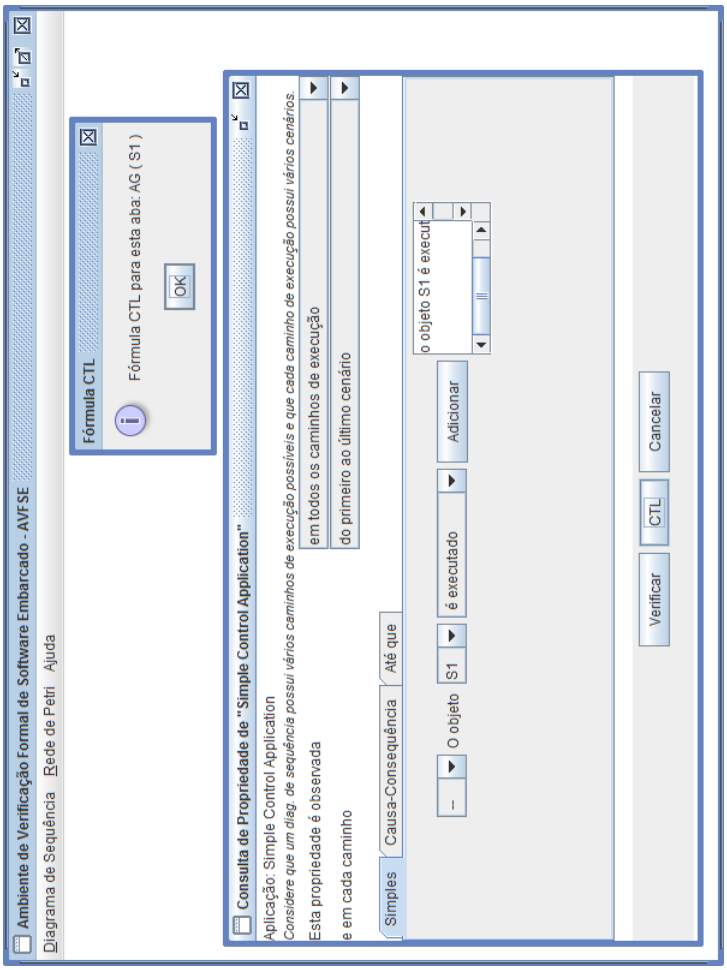

Figura 5.7: Primeira Consulta

Figura 5.8: Resultado da Primeira Consulta

```
S14 = 0
S16 = 0
S6 = 0
S23 = 0
S18 = 0
S12 = 0
S2 = 0
S19 = 0
S3 = 0
S22 = 0
S10 = 0
S20 = 0
S11 = 0
S4 = 0
S25 = 0
S5 = 0
S28 = 0
S26 = 0
S9 = 0
S8 = 0
S7 = 0


resources used:
processor time: 0.008 s,
BDD nodes allocated: 3447
Bytes allocated: 1045216
BDD nodes representing transition relation: 424 + 1
```

5.3.3 Consulta II

A segunda consulta verifica se, em alguns caminhos de execução, em algum cenário à frente, o objeto $S25$ é executado E o objeto $S16$ é executado E o objeto $S23$ é executado. A Figura 5.9 mostra a tela do AVFSE e a fórmula CTL equivalente. Esta propriedade é útil para verificar se processos X, Y e Z em algum ponto são executados em paralelo. Após pressionar-se o botão "Verificar", o diagrama de sequência é traduzido para a notação formal do SMV, apta a ser formalmente verificada

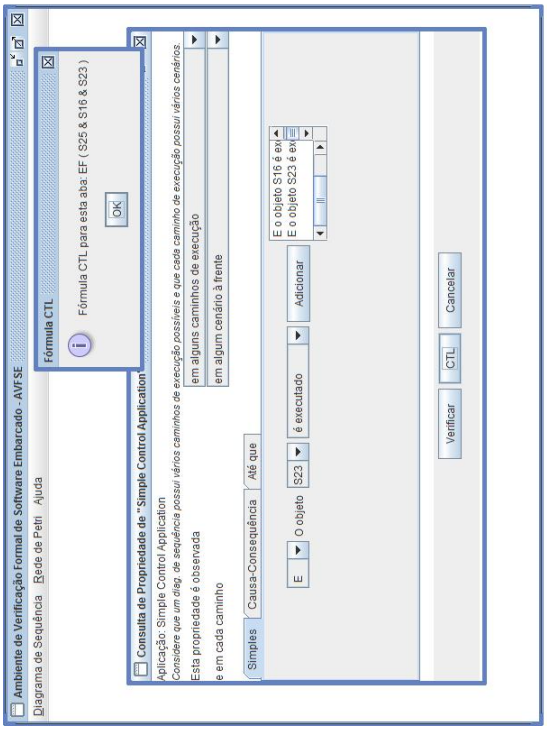

Figura 5.9: Segunda Consulta

através de *model checking* pela ferramenta SMV.

A fórmula "`EF ( S25 & S16 & S23 )`"é **VERDADEIRA**, conforme atesta a Figura 5.10.

5.3.4 Consulta III

A terceira consulta verifica se, em todos os caminhos de execução, do primeiro ao último cenário, os objetos $S4$, $S5$, $S25$, $S28$, $S26$, $S9$, $S8$ são executados, ou seja, se há um estado em que todos estes objetos são executados ao mesmo tempo. Estes objetos são os objetos do processador $P3$, de acordo com a descrição do experimento em 5.1. A Figura 5.11 mostra a tela do AVFSE e a fórmula CTL equivalente. Após pressionar-se o botão "Verificar", o diagrama de sequência é traduzido para a notação formal do SMV, apta a ser formalmente verificada através de *model checking* pela ferramenta SMV.

A fórmula "`AG ( S4 & S5 & S25 & S28 & S26 & S9 & S8 )`"é **FALSA**, conforme atesta a Figura 5.12. O contra-exemplo gerado pelo SMV demonstra que logo no primeiro cenário

Figura 5.10: Resultado da Segunda Consulta

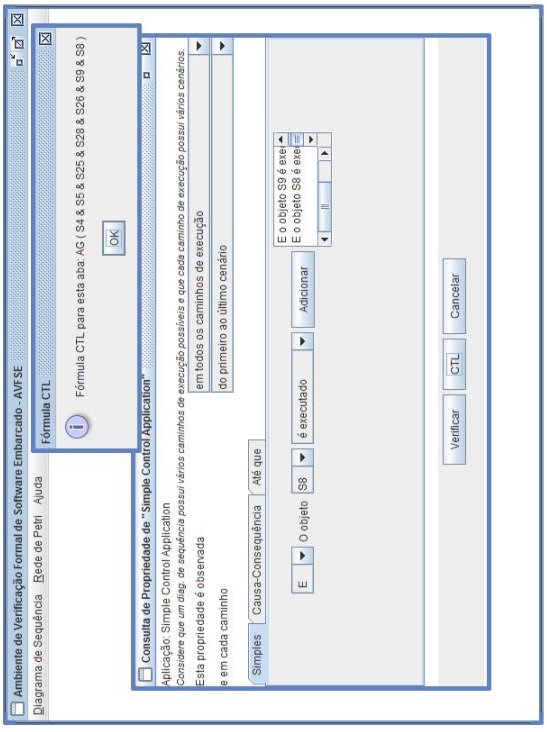

Figura 5.11: Terceira Consulta

Figura 5.12: Resultado da Terceira Consulta

ou estado o objetos não serão executados, tendo valor zero, invalidando a propriedade verificada nesta consulta.

Eis o *log* da execução no SMV para esta propriedade, juntamente com os recursos computacionais alocados:

```
-- specification AG (S4 & S5 & S25 & S28 & S26 & S9 & S8) is false
-- as demonstrated by the following execution sequence
state 1.1:
S1 = 0
S14 = 0
S16 = 0
S6 = 0
S23 = 0
S18 = 0
S12 = 0
S2 = 0
S19 = 0
S3 = 0
S22 = 0
S10 = 0
S20 = 0
S11 = 0
S4 = 0
S25 = 0
```

```
S5 = 0
S28 = 0
S26 = 0
S9 = 0
S8 = 0
S7 = 0


resources used:
processor time: 0.003 s,
BDD nodes allocated: 3466
Bytes allocated: 1045216
BDD nodes representing transition relation: 424 + 1
```

5.3.5 Consulta IV

A quarta consulta verifica se, em todos os caminhos de execução, do primeiro ao último cenário, SE o objeto $S1$ é executado ENTÃO o objeto $S7$ é executado às vezes. O "ENTÃO"equivale a afirmar que, a partir do momento, na árvore de execução, em que o antecedente (que neste caso é "o objeto S1 é executado") for verdadeiro, o consequente (que neste caso é "o objeto S7 é executado") será verdadeiro em algum dos caminhos à frente, ou seja, "às vezes". A Figura 5.13 mostra a tela do AVFSE e a fórmula CTL equivalente. Após pressionar-se o botão "Verificar", o diagrama de sequência é traduzido para a notação formal do SMV, apta a ser formalmente verificada através de *model checking* pela ferramenta SMV.

A fórmula "`AG ( (S1) -> (EF S7) )`"é **VERDADEIRA**, conforme atesta a Figura 5.14. Analisando-se o diagrama de sequência, pode-se concluir que, se S1 for executado, S7 o será também em algum ponto pelo menos no futuro é afirmar simplesmente que quase todo o sistema representado pelo diagrama de sequência será executado a partir do objeto inicial S1. Isto porque S14, executado após S1, é precedido por várias execuções, quais sejam: S2, S3, S19, S22 e S20. O objeto S16, executado após S14, precede S23 e este, por sua vez, precede S18, executado também após S16. O último objeto é S7, precedido por S18 e S4, S5, S25, S26 e S28. As exceções são

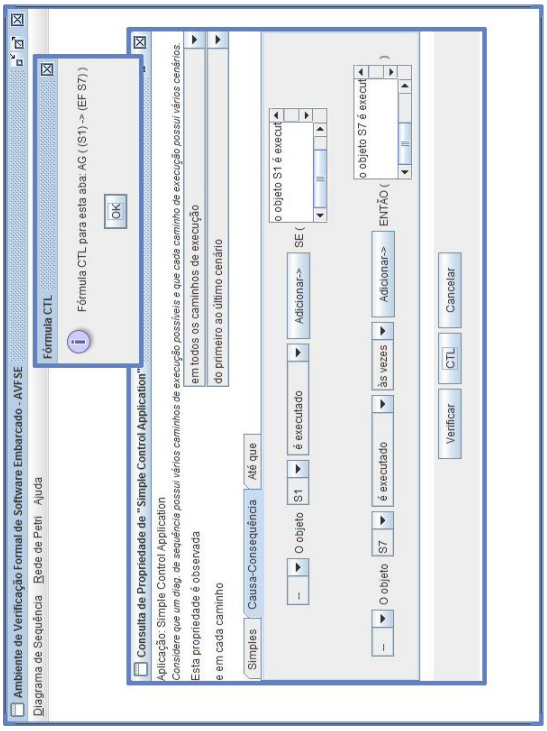

Figura 5.13: Quarta Consulta

os objetos S8, S9, S10, S11 e S12. Mas estes não têm o poder de evitar a execução de S7 caso algo ocorra com os mesmos. Portanto, esta fórmula, a despeito de sua simplicidade, é bastante abrangente.

Eis o *log* da execução no SMV para esta propriedade, juntamente com os recursos computacionais alocados:

```
-- specification AG (S1 -> EF S7) is true

resources used:
processor time: 0.015 s,
BDD nodes allocated: 3798
Bytes allocated: 1045216
BDD nodes representing transition relation: 424 + 1
```

Figura 5.14: Resultado da Quarta Consulta

5.3.6 Consulta V

A quinta consulta verifica se, em todos os caminhos de execução, do primeiro ao último cenário, SE o objeto S1 é executado ENTÃO o objeto S7 é executado sempre. A Figura 5.15 mostra a tela do AVFSE e a fórmula CTL equivalente. Após pressionar-se o botão "Verificar", o diagrama de sequência é traduzido para a notação formal do SMV, apta a ser formalmente verificada através de *model checking* pela ferramenta SMV.

A fórmula "`AG ( (S1) -> (AG S7) )`"é **FALSA**, conforme atesta a Figura 5.16 porque amarra a execução de S7 a todos os caminhos de execução e seus cenários posteriores assim que S1 for executados, o que certamente não é coerente. O contra-exemplo mostrado no *log* de execução do SMV comprova isso.

Eis o *log* da execução no SMV para esta propriedade, juntamente com os recursos computacionais alocados:

```
-- specification AG (S1 -> AG S7) is false
-- as demonstrated by the following execution sequence
state 1.1:
S1 = 0
S14 = 0
S16 = 0
```

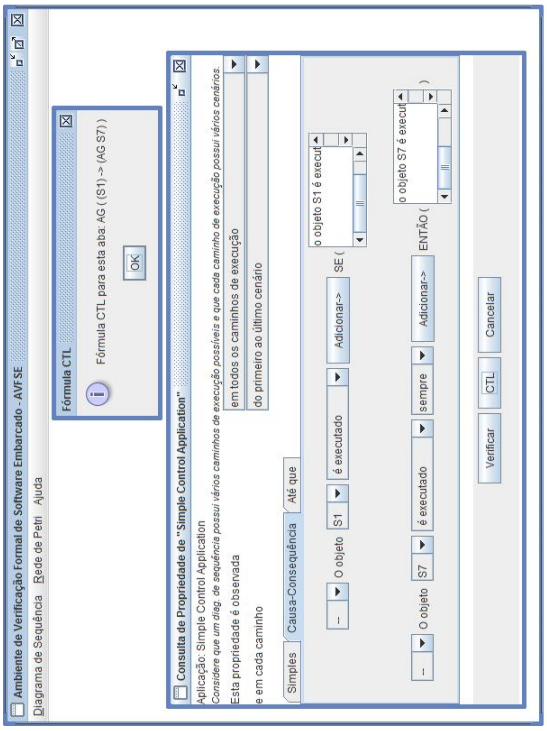

Figura 5.15: Quinta Consulta

Figura 5.16: Resultado da Quinta Consulta

```
S6 = 0
S23 = 0
S18 = 0
S12 = 0
S2 = 0
S19 = 0
S3 = 0
S22 = 0
S10 = 0
S20 = 0
S11 = 0
S4 = 0
S25 = 0
S5 = 0
S28 = 0
S26 = 0
S9 = 0
S8 = 0
S7 = 0

state 1.2:
S1 = 1
S6 = 1
S2 = 1
S3 = 1
S4 = 1
S5 = 1



resources used:
processor time: 0 s,
BDD nodes allocated: 3668
Bytes allocated: 1045216
BDD nodes representing transition relation: 424 + 1
```

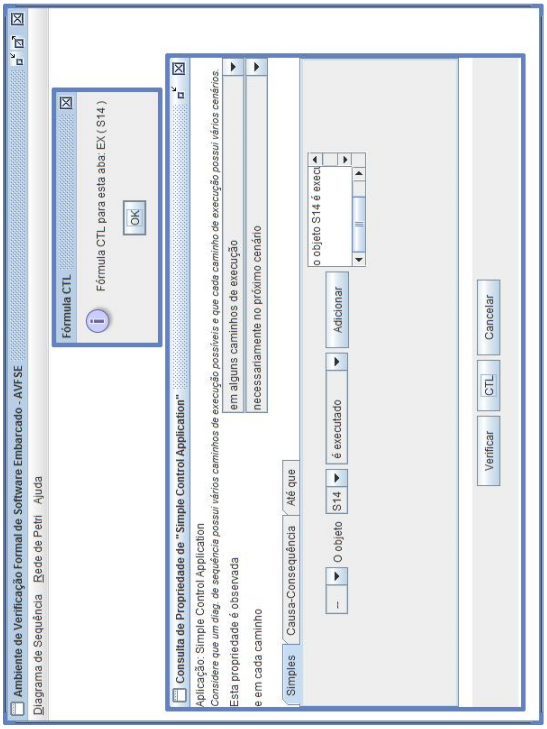

Figura 5.17: Sexta Consulta

5.3.7 Consulta VI

A sexta consulta verifica se, Em alguns caminhos de execução, necessariamente no próximo cenário, o objeto S14 é executado. A Figura 5.17 mostra a tela do AVFSE e a fórmula CTL equivalente. Após pressionar-se o botão "Verificar", o diagrama de sequência é traduzido para a notação formal do SMV, apta a ser formalmente verificada através de *model checking* pela ferramenta SMV.

A fórmula "`EX ( S14 )`"é **FALSA** porque considera em algum caminho de execução, imediatamente o seu primeiro cenário. Não há essa possibilidade logo de início porque, como demonstrado pelo contra-exemplo, logo no primeiro cenário S14 não é executado, conforme atesta a Figura 5.18 visto que é impossível que, em todos os caminhos de execução, em cada um de seus cenários, S1 seja executado. Por outro lado, "EX (S2)"é VERDADEIRO.

Eis o *log* da execução no SMV para esta propriedade, juntamente com os recursos computacionais alocados:

Figura 5.18: Resultado da Sexta Consulta

```
-- specification EX S14 is false
-- as demonstrated by the following execution sequence
state 1.1:
S1 = 0
S14 = 0
S16 = 0
S6 = 0
S23 = 0
S18 = 0
S12 = 0
S2 = 0
S19 = 0
S3 = 0
S22 = 0
S10 = 0
S20 = 0
S11 = 0
S4 = 0
S25 = 0
S5 = 0
S28 = 0
S26 = 0
S9 = 0
S8 = 0
S7 = 0
```

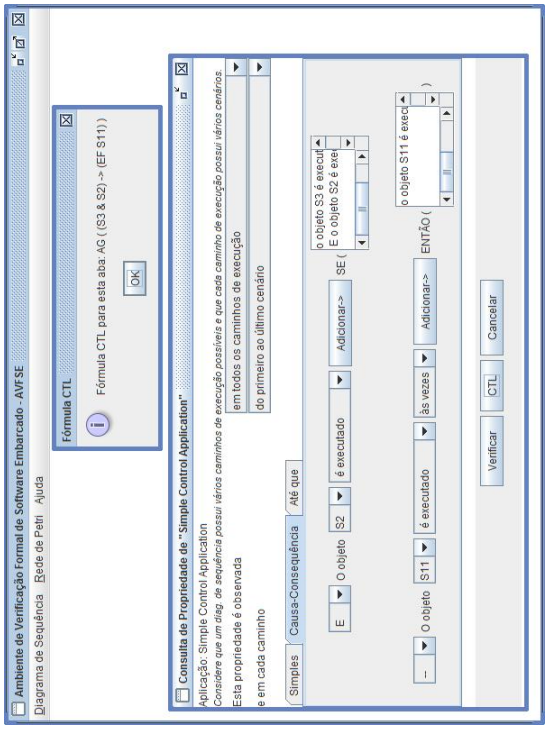

Figura 5.19: Sétima Consulta

```
resources used:
processor time: 0 s,
BDD nodes allocated: 3466
Bytes allocated: 1045216
BDD nodes representing transition relation: 424 + 1
```

5.3.8 Consulta VII

A sétima consulta verifica se, em todos os caminhos de execução, do primeiro ao último cenário, SE o objeto S3 é executado E o objeto S2 é executado ENTÃO o objeto S11 é executado às vezes. A Figura 5.19 mostra a tela do AVFSE e a fórmula CTL equivalente. Após pressionar-se o botão "Verificar", o diagrama de sequência é traduzido para a notação formal do SMV, apta a ser formalmente verificada através de *model checking* pela ferramenta SMV.

A fórmula "AG ((S3 & S2) -> (EF S11))"é **VERDADEIRA**, conforme atesta a Figura 5.20.

Figura 5.20: Resultado da Sétima Consulta

De igual forma, o ENTÃO pode ser entendido como "a partir do momento em que o antecedente é verdadeiro". A partir do momento em que se alcança um estado ou cenário em que tanto S3 quanto S2 são executados, ter-se-á, am algum caminho à frente, a execução de S11 também.

Eis o *log* da execução no SMV para esta propriedade, juntamente com os recursos computacionais alocados:

```
-- specification AG (S3 & S2 -> EF S11) is true

resources used:
processor time: 0.004 s,
BDD nodes allocated: 3388
Bytes allocated: 1045216
BDD nodes representing transition relation: 424 + 1
```

5.3.9 Consulta VIII

Esta consulta é do tipo "pUq", ou "P se verificará até que Q seja verdade". A utilidade prática desse tipo de consulta é verificar se o ordenamento e sequenciamento estabelecido entre as tarefas é de fato respeitado nos vários caminhos ou possibilidades de execução. A oitava consulta verifica se, Em todos os caminhos de execução, do primeiro ao último cenário, a condição "o objeto S1 é

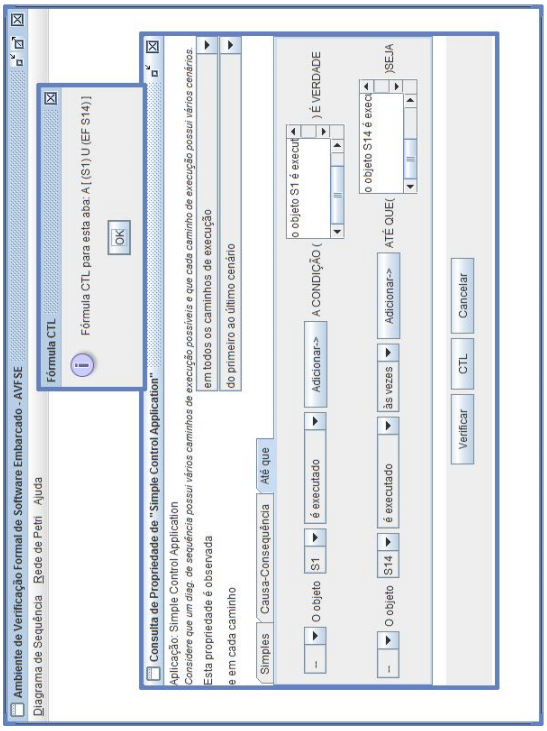

Figura 5.21: Sétima Consulta

executado"é verdade até que "o objeto S14 seja executado às vezes". A Figura 5.21 mostra a tela do AVFSE e a fórmula CTL equivalente. Após pressionar-se o botão "Verificar", o diagrama de sequência é traduzido para a notação formal do SMV, apta a ser formalmente verificada através de *model checking* pela ferramenta SMV.

A fórmula "A [(S1) U (EF S14)]"é **VERDADEIRA**, conforme atesta a Figura 5.22. Esta consulta implica também num ordenamento natural mostrando S1 necessariamente executando antes de S14.

5.4 Resumo do Capítulo

Este Capítulo detalhou consultas de propriedades à aplicação simples de controle, especificada em (DINATALE, 1994). Esta aplicação possui quatro processadores e várias tarefas distribuídas entre eles. Cada tarefa é um objeto do domínio da aplicação. De posse do diagrama de sequência com

Figura 5.22: Resultado da Oitava Consulta

os objetos do domínio, pode-se traduzir para a notação formal SMV e, de posse das consultas em CTL traduzidas automaticamente a partir de descrições em linguagem natural, realizar verificação de modelos com a técnica de *model checking*.

O próximo Capítulo tece as considerações finais do trabalho, faz um resumo do que foi apresentado até aqui e traz algumas possibilidades de estender o presente método em trabalhos futuros.

Capítulo 6

Considerações Finais

6.1 Conclusões

No universo dos softwares embarcados, o processo de desenvolvimento se torna mais árduo do que para sistemas desktop, devido a restrições de consumo de energia, peso e tamanho, tempo etc, conforme apresentado na Seção 3.1. Além disso, como também foi exposto, muitos desses sistemas, em caso de falhas em seu funcionamento, podem acarretar muitos danos à vida humana ou prejuízos financeiros. Portanto, fazem-se necessários métodos formais para a precisa verificação de tais sistemas logo nas primeiras fases do desenvolvimento, ainda em tempo de projeto e a consequente constatação de que foram corretamente projetados.

Todavia, fazem-se necessários também métodos ágeis no sentido de serem intuitivos e de fácil uso por parte do desenvolvedor principalmente por causa da crescente pressão de Mercado por novos produtos rapidamente, ou *time-to-Market*. Atendendo a tais demandas, o presente trabalho apresentou um método intuitivo de verificação formal de software embarcado, exposto no Capítulo 4. O método apresentado toma como entrada uma especificação dos objetos do domínio da aplicação na notação do Diagrama de Sequência da UML, apresentado na seção 3.3. O método proposto no presente trabalho visa atender aos aspectos apresentados em 4.1. Para tanto, ele procura se alinhar com a visão apresentada em (SANGIOVANNI-VINCENTELLI, 2001) a qual advoga um

fluxo matematicamente correto a partir da especificação do software embarcado até sua implementação.

Métodos formais referem-se a técnicas e ferramentas matematicamente rigorosas para especificação, projeto e verificação de sistemas de software e de hardware. O termo "‘matematicamente rigoroso"’ se refere às especificações usadas em métodos formais que são sentenças bem formadas em lógica matemática e que são verificações formais de deduções rigorosas naquela lógica (NASA, 2001). A importância dos métodos formais é que eles provêem meios de examinar simbolicamente todo o espaço de estados de um projeto digital, quer seja de hardware ou de software, e estabelecer uma corretude ou propriedade de segurança que é verdadeira para todas as possíveis entradas (NASA, 2001). Entretanto, sem um processo de software sólido não há possiblidade de se empregar com sucesso métodos formais (DAVIS, 2005).

A grande vantagem da notação UML utilizada pelo método proposto é a de livrar o projetista da necessidade de conhecer formalismos. Por isso ela é intuitiva e, como consequência, largamente utilizada pela Indústria. Para assegurar a tão esperada corretude, a ferramenta que realiza o presente método apresentou regras de tradução do diagrama de sequência para a notação Rede de Petri Lugar/Transição ou, simplesmente, Redes de Petri, apresentada na Seção 3.4. A notação Redes de Petri é suficiente formal e matematicamente rigorosa para qualquer tipo de verificação de sistemas críticos, sejam eles distribuídos ou não. Em particular, o presente trabalho adota a técnica de Verificação de Modelos ou *Model Checking*. Como exposto na seção 3.5.5, esta técnica apresenta algumas limitações no seu uso prático. O verificador de modelos ou *model checker* utilizado foi o SMV.

Por fim, não adiantaria muita coisa ter a especificação em alto nível se as consultas a propriedades ainda fossem escritas numa notação formal. Por isso, visando também agilizar o desenvolvimento, o presente método trouxe técnicas para traduzir consultas em linguagem natural para a notação *Computational Tree Logic* (CTL), própria para verificação em *model checkers*, tais como o MCK ou o próprio SMV.

Todos os passos do método são realizados por um ambiente de software, o Ambiente de Ve-

rificação Formal de Software Embarcado. O funcionamento da ferramenta e seu projeto foram detalhados na Seção 4.3. O experimento apresentado no Capítulo 5 valida as técnicas descritas neste trabalho.

Vários trabalhos relacionados têm sido desenvolvidos, como mostrou o Capítulo 2. Muitos deles usavam a notação de *Statechart*, Diagrama de Atividades ou mesmo o Diagrama de Sequência da UML como notação de modelagem e, então, realizam algum tipo de tradução para algum tipo de formalismo, principalmente Redes de Petri. Alguns deles utilizam a rede gerada final somente no contexto de Avaliação de Desempenho. O presente trabalho visa, como os outros, acrescentar ao estado da arte da verificação formal, em particular de software embarcado, de maneira que mais e mais práticas formais sejam usadas na Indústria.

6.2 Contribuições

A seguir, um sumário das principais contribuições obtidas pelo presente trabalho e pelo Ambiente que o realiza, cujos conceitos podem potencialmente ser usados para todo e qualquer tipo de software, embarcado ou não.

- Tradução automática do diagrama de sequência para a notação formal SMV.
- Verificação desse modelo via *model checking* simbólico, através da ferramenta SMV.
- Tradução automática de Diagrama de Sequência para outra notação formal, no caso Redes de Petri.
- Escrita da Rede de Petri final nos formatos APNN, PNML e SMV.
- Interface amigável pra coleta de consultas em linguagem natural.
- Tradução automática dessas consultas para CTL.
- Todos os conceitos aplicados a software embarcado.

- Experimento com software embarcado distribuído.
- Ferramenta para realizar o método.

6.3 Trabalhos Futuros

Este trabalho pode ser estendido para contemplar futuramente alguns novos pontos a fim de mais abrangentemente possibilitar a verificação formal de software embarcado. O primeiro ponto a ser investigado é a questão de perfis relacionados a tempo da UML, tais como o UML-SPT. Outro ponto a ser investigado são as extensões de tempo à notação de Redes de Petri L/T, tais como as Redes de Petri Temporais.

Referências

A380, AirBus A380 Wikipedia. Disponível em http://en.wikipedia.org/wiki/A-380. Acesso em Março de 2009.

ABRIAL, J-R. The B Book - Assigning Programs to Meanings. Cambridge University Press, 1996.

AMARAL, A. Estudo de Técnicas Usadas em Verificação Formal de Modelos para Prevenção do Problema "'Explosão de Estados"'. 2004.

AMORIN, L.; et al. Mapping live sequence chart to coloured petri nets for analysis and verification of embedded systems. In: SIGSOFT Softw. Eng. Notes, 2006.

ANÁLISE. Análise de Sistemas II: Artigos relacionados. Disponível em http://professores.unisanta.br/sobrino/ana-artmetrica.htm. Acesso em Março de 2009.

APNN, Abstract Petri Net Notation. Disponível em http://ls4-www.informatik.uni-dortmund.de/QM/MA/fb/publication_ps_files/APNN.ps.gz. Acesso em: Outubro de 2008.

BARROCA, L.; MCDERMID, J. Formal Methods: Use and Relevance for the Development of Safety-Critical Systems. The Computer Journal, 1992.

BERNARDI, S.; DONATELLI, S.; MERSEGUER, J. From UML sequence diagrams and statecharts to analysable petri net models. In: 3rd international Workshop on Software and Performance, 2002.

BIANCO, V. Del; LAVAZZA, L.; MAURI, M.; OCCORSO, G. Towards UML-based formal specifications of component-based real-time software. International Journal on Software Tools for Technology Transfer (STTT) - Springer-Verlag GmbH, 2008.

BLASKOVIC, B. Petri net modeling for reactive system verification. In: 7th International Conference on Telecommunications - ConTEL , 2003.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. The Unified Modeling Language User Guide. Addison Wesley, 1998.

BONNEFOI, F. Design, modeling and analysis of ITS using UML and Petri Nets. In: IEEE Intelligent Transportation Systems Conference - ITSC, 2007.

BOUTEKKOUK, F.; BENMOHAMMED, M. UML Modeling and Formal Verification of Control/Data Driven Embedded Systems. In: 14th IEEE International Conference on Engineering of Complex Computer Systems, 2009.

BOURQUE, P. Guide to the software engineering body of knowledge - A straw man version. Dept. dInformatique, UQAM, 1998.

BOWEN, J.; HINCHEY, M. Seven more myths of formal methods. In: IEEE Software, 1995.

BRAUER, W. Net Theory and Applications. Lecture Notes in Computer Science. Springer-Verlag, 1979.

BROADFOOT, G.; HOPCROFT, P. Introducing formal methods into industry using Cleanroom and CSP. Dedicated Systems e-Magazine, 2005.

BURNS; WELLINGS. Real-Time Systems and Programming Languages, 3rd edition Ada 95, Real-Time Java and Real-Time POSIX. Addison Wesley, 2001.

CLEANROOM SOFTWARE ENGINEERING, Wikipedia. Disponível em http://en. wikipedia .org/wiki/Cleanroom Software Engineering. Acesso em Março de 2009.

COHEN, B.; ARWOOD, W.; JACKSON, M. The Specification of Complex Systems. Addison-Wesley, 1986.

COOLING, J. E. Software Engineering for Real-Time Systems, First Edition. Addison-Wesley, 2003.

DAVIS, J. The affordable application of formal methods to software engineering. In: 2005 annual ACM SIGAda international conference on Ada.

DICESARE, F.; CHANG, S.; GOLDBOGEN, G. Failure Propagation Trees for Diagnosis in Manufacturing Systems. In: IEEE Trans. on Systems, Manuf. and Cybernetics, 1991.

DINATALE, M.; STANKOVIC, J. Dynamic end-to-end guarantees in distributed realtime systems. In: IEEE Real-Time Systems Symposium, 1994.

DOD, Department of Defense - DoD / USA. Systems Engineering Fundamentals. Defense Acquistion University Press, 2001.

DOUGLASS, B. Introduction to UML sequence diagrams, 2003. Disponível em `http://www.embedded.com/story/OEG20030521S0061`. Acesso em Março de 2009.

EMERSON, E. The Beginning of Model Checking: A Personal Perspective. 2007.

ESHUIS, R. Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol, 2006.

EVENTSTUDIO, EventStudio System Designer 4.0, Event Helix. Disponível em `http://www.eventhelix.com/`. Acesso em Março de 2009.

FERNANDES, J.; et al. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In: Sixth International Workshop on Scenarios and State Machines - SCESM 2007.

FIRMWARE, Wikipedia, 2009. Disponível em http://pt.wikipedia.org/ wiki/ Firmware. Acesso em Março de 2009.

FUNCTIONAL SPECIFICATION, Wikipedia. Disponível em `http://en.wikipedia.org/wiki/Programspecification`. Acesso em Março de 2009.

GOMES, L.; COSTA, A.; MEIRA, P. From Use Cases to Building Monitoring Systems through Petri Nets. In: IEEE International Symposium on Industrial Electronics, 2005. ISIE 2005.

GU, Z.; SHIN, G.. Synthesis of Real Time Implementation from UML-RT Models, 2004.

HAUSE, M. The SysML Modelling Language. In: Fifth European Systems Engineering Conference, 2006.

HOARE, C. Communicating Sequential Processes. In: Prentice Hall International, 1985.

HOLZMANN, G. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2004.

IMPLEMENTATION, Wikipedia, 2009. Disponível em http://en.wikipedia.org/wiki/Implementation. Acesso em Março de 2009.

JENG, M.; LU, W. Extension of UML and Its Conversion to Petri Nets for Semiconductor Manufacturing Modeling. In: IEEE International Conference on Robotics and Automation, 2002.

JUDEUML, Jude Design and Communication. Disponível em http://jude.change-vision.com/jude-web/product/index.html. Acesso em Março de 2009.

KANER, C. Exploratory Testing, 2006. Quality Assurance Institute Worldwide Annual Software Testing Conference, 2006. Disponível em http://www.kaner.com/pdfs/ETatQAI.pdf. Acesso em Março de 2009.

KATOEN, J. Concepts, Algorithms and Tools for Model Checking. 1999.

KOOPMAN, P. Embedded System Design Issues (The Rest of the Story). In: International Conference on Computer Design (ICCD96), 1996.

KOPETZ, H. Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, 2002.

LAMSWEERDE, A. Formal Specification: A Roadmap. In: ACM conference on The Future of Software Engineering, 2002.

LAPLANTE, P. Real Time Systems Design, 3rd Edition. CRC Press, 2004.

LATRONICO, E.; KOOPMAN, P. Representing Embedded System Sequence Diagrams as a Formal Language. In: 4th International Conference on UML, 2001.

LOTOS, A Formal description technique Based on Temporal Ordering of Observational Behaviour. 1987.

MARTIN, G. UML for embedded systems specification and design: motivation and overview. In: Design, Automation and Test in Europe Conference and Exhibition, 2002.

MARIA, The Modular Reachability Analyzer. Disponível em http://www.tcs.hut.fi/

Software/maria/index.en.html. Acesso em Março de 2009.

MARWEDEL, P. Embedded Systems Design. Springer, 2006.

MCK, Model Checking Kit. Institute of Formal Methods in Computer Science. Disponível em http://www.fmi.uni-stuttgart.de/szs/tools/mckit/. Acesso em Outubro de 2008.

MCMILLAN, K. Symbolic Model Checking. Kluwer Academic Publ., 1993.

MELLOR, S. Embedded Systems in UML, 2007. Disponível em www.omg.org/news/ whitepapers/ 050307_Embedded_Systems_in_UML_by_S_Mellor.pdf. Acesso em Março de 2009.

MÉTODOS FORMAIS, Wikipedia, 2009. http://pt.wikipedia.org/ wiki/ Métodos formais. Acesso em Março de 2009.

MILNER, R. Communication and Concurrency. Prentice Hall, 1989.

MOKHATI, F.; GAGNON, P.; BADRI, M. Verifying UML Diagrams with Model Checking: A Rewriting Logic Based Approach. In: Seventh International Conference on Quality Software - QSIC, 2007.

NASA. NASA Langley Formal Methods. What is Formal Methods? Disponível em http://shemesh.larc.nasa.gov/fm/fm-what.html. Acesso em Outubro de 2008.

NuSMV: a new symbolic model checker. Disponível em http://nusmv.fbk.eu/. Acesso em Março de 2009.

YAO, S.; SHATZ, M. Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. In: 15th international Conference on Computing, 2006.

OCL, Object Constraint Language (OCL), OMG, 2006. Disponível em http://www.omg.org/spec/OCL/2.0/. Acesso em Março de 2009.

OMG, The Object Modeling Group. Disponível em http://www.omg.org/. Acesso em Março de 2009.

PNML, Petri-Net Markup Language. Disponível em http://www.pnml.org/ version-2009/ version-2009.php. Acesso em Março de 2009.

PONT, M. Patterns for time-triggered embedded systems. ACM PRESS BOOKS, 2008.

PRESSMAN, R. A Software Engineering, a Practioner's Approach. McGraw-Hill, 2001.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and sofware design methodology for embedded systems. In: IEEE Design and Test of Computers, 2001.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. A vision for embedded software. In: International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'01).

SHEN, H.; VIRANI, A.; NIU, J. Formalize UML 2 Sequence Diagrams. In: 11th IEEE High Assurance Systems Engineering Symposium, 2008.

SMV, Symbolic Model Verifier. Disponível em `http://vuv-cad.eecs.berkeley.edu/-kenmcmil/smv`. Acesso em Outubro de 2008.

SOFFEL, V. Developing Embedded Systems - A Tools Introduction, 2003. Disponível em http:// www. devarticles. com/c/a/ Embedded-Tools/ Developing- Embedded- Systems– A- Tools - Introduction/. Acesso em Março de 2009.

SOFTWARE ARCHITECTURE, Wikipedia, 2009. Disponível em `http://en.wikipedia.org/wiki/Softwarearchitecture`. Acesso em Março de 2009.

SOFTWARE DESIGN, Wikipedia, 2009. Disponível em `http://en.wikipedia.org/wiki/Softwaredesign`. Acesso em Março de 2009.

SOFTWARE DEVELOPMENT PROCESS, Wikipedia, 2009. Disponível em http:// en. wikipedia. org/ wiki/ Software development process. Acesso em Março de 2009.

SOFTWARE ENGINEERING, Wikipedia, 2009. Disponível em `http://en.wikipedia.org/wiki/Softwareengineering`. Acesso em Março de 2009.

SOMMERVILLE, I. Software Engineering, 8th ed. Addison-Wesley Publishers Limited, 2007.

SPIVEY, J. Z Notation - A Reference Manual. Prentice Hall International, 1985.

TREMBLAY, G. Formal Methods: Mathematics, Computer Science, or Software Engineering? In: 13th Conference on Software Engineering Education & Training - CSEET '00.

TRIBASTONE, M.; GILMORE, S. Automatic Translation of UML Sequence Diagrams into PEPA

Models. In IEEE Quantitative Evaluation of SysTems, 2008.

TROWITZSCH, J.; ZIMMERMANN, A. Using UML state machines and petri nets for the quantitative investigation of ETCS. In: 1st international Conference on Performance Evaluation Methodolgies and Tools, 2006.

UML-SPT, UML Profile For Schedulability, Performance, And Time, Version 1.1. OMG. Disponível em `http://www.omg.org/technology/documents/formal/schedulability.htm`. Acesso em Março de 2009.

VAZIRI, M.; JACKSON, D. Some Shortcomings of OCL, the Object Constraint Language of UML, 1999. Disponível em `http://www.omg.org/docs/ad/99-12-05.pdf`. Acesso em Março 2009.

WEGENER, I. Branching programs and binary decision diagrams: theory and applications. Society for Industrial and Applied Mathematics, 2000.

WEIGUO, H.; GODDARD, S. Capturing an application's temporal properties with UML forReal-Time. In: High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000.

WOLF, W. What is Embedded Computing. Journal of IEEE, 2002.

ZHAO, Y.; et al. Towards formal verification of UML diagrams based on graph transformation. In: IEEE International Conference on E-Commerce Technology for Dynamic E-Business, 2004.