

ESPECIFICAÇÃO EXECUTÁVEL USANDO UMA
LINGUAGEM DE REDES DE PETRI NO
DOMÍNIO DE SISTEMAS EMBARCADOS

CHRISTOPHE SAINT-CHRISTIE DE LIMA XAVIER

ESPECIFICAÇÃO EXECUTÁVEL USANDO UMA
LINGUAGEM DE REDES DE PETRI NO
DOMÍNIO DE SISTEMAS EMBARCADOS

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Instituto de Ciências Exatas da Universidade Federal do Amazonas como requisito parcial para a obtenção do grau de Mestre em Informática.

ORIENTADOR: RAIMUNDO DA SILVA BARRETO

Manaus
Fevereiro de 2011

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

X3e Xavier, Christophe Saint-Christie de Lima
Especificação Executável usando uma Linguagem de Redes de Petri no Domínio de Sistemas Embarcados / Christophe Saint-Christie de Lima Xavier . 2011
72 f.: il. color; 31 cm.

Orientador: Raimundo da Silva Barreto
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Redes de Petri. 2. Bounded Model Checker. 3. Compiladores. 4. Prototipação. I. Barreto, Raimundo da Silva. II. Universidade Federal do Amazonas III. Título



UNIVERSIDADE FEDERAL DO AMAZONAS

INSTITUTO DE CIÊNCIAS EXATAS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

COORDENAÇÃO DO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

FOLHA DE APROVAÇÃO

**"Especificação Executável Usando uma Linguagem de Redes de Petri
no Domínio de Sistemas Embarcados"**

CHRISTOPHE SAINT- CHRISTIE DE LIMA XAVIER

Dissertação defendida e aprovada pela banca examinadora
constituída pelos Professores:

PROF. RAIMUNDO DA SILVA BARRETO – PRESIDENTE

PROF. RUTER BRAGA CALDAS – MEMBRO

PROF. CÍCERO AUGUSTO MOTA CAVALCANTE – MEMBRO

Manaus, 11 de fevereiro de 2011.

A Deus, pela graça de ter me permitido concluir este trabalho. A minha mãe, que sempre me mostrou o caminho do amor às pessoas, da educação e da verdade. Dedico também ao meu Orientador que sempre me protegeu e me guiou pelos caminhos corretos.

Agradecimentos

Eu agradeço a Deus e minha família por todos os momentos de minha vida estarem juntos nas vitórias e nas derrotas.

Também agradeço à contribuição de meus colegas acadêmicos e professores que me conduziram à esta vitória, em especial a Cristiane Pacheco, Jucimar Souza, Ricardo Câmara, Lady Daiana Pinto, Herbert Oliveira Rocha, Osman Seixas, Juan Colonna, Onilton Maciel, Vandermir Silva, Davi Viana e ao professor Ruiter Braga.

Aos professores Carlos Alberto Barata e Odette Mestrinho que, por suas ações, me motivaram a ir mais adiante.

Ao meu renomado orientador Professor Dr. Raimundo Barreto, que me passa referência a quem devoto a mais sincera e efusiva admiração.

A Professora Tayana Conte e Professor Horácio Fernandes que acreditaram em mim do início ao fim deste trabalho.

Ao Diretor do Instituto de Ciências Exatas de Itacotiara (UFAM-ICET), professor Cícero Mota Cavalcante e a sua secretária Estrela pelo apoio dado a minha formação profissional

A Ilma Sra. Elienai Nogueira, a todos os professores e os demais da coordenação técnica do Programa de Mestrado em Informática da UFAM por contribuíram de forma direta e indiretamente para minha formação acadêmica.

Ao meu amigo de infância João Batista Júnior e sua namorada Ana Valéria pelo apoio moral.

Ao meu amigo Antônio Eduardo Lobo que passou a ser um irmão, uma parte da minha família.

Ao meu amigo Efren Lopes por seu apoio moral.

Aos membros da banca examinadora pelas opiniões valiosas e preciosas sugestões.

Obrigado a todos!

“A mente que se abre a uma nova idéia jamais voltará ao seu tamanho original.”
(Albert Einstein)

Resumo

Este trabalho descreve uma metodologia para a geração automática de código para sistemas embarcados, a partir de uma rede de Petri, com objetivo de minimizar o tempo gasto na codificação do programa e automatizar completamente o processo de transformação. A abordagem proposta utiliza uma Especificação Executável baseada em Redes de Petri, onde nesta pode ser verificado algumas de suas propriedades como *deadlock*, vivacidade, limitação, alcançabilidade, dentre outras. As Redes de Petri contêm transições que podem ter códigos anotados na linguagem de programação C e NXC, que executarão partes específicas do sistema que estará sendo modelado. Adicionalmente, os códigos anotados na linguagem de programação C serão verificados por um *Bounded Model Checker*, que testará propriedades específicas do código, como limite de *arrays*, divisão por zero, segurança de ponteiros e outras. Esta especificação serve de base para exibir ao usuário as funcionalidades do sistema que será modelado, proporcionando ao usuário uma visão das características específicas do sistema. Desta forma, contribuindo com desenvolvedores e engenheiros de software na geração de protótipos que constituem uma especificação executável, facilitando a avaliação de diferentes modelos e ajudando a reduzir as diferenças de interpretação na construção de *software*. Este trabalho também apresenta uma ferramenta denominada de PNTCG (*Petri Net Tool for Code Generation*) desenvolvida com base nesta metodologia e um estudo de caso baseado em um protótipo de automatização de embalagens de produto, no qual, é utilizado uma esteira e um braço robô para demonstrar a utilização e aplicação da metodologia proposta.

Palavras-chave: Redes de Petri, Bounded Model Checker, Compiladores, Prototipação.

Abstract

This work describes a methodology for automatic code generation for embedded systems, from a Petri net, in order to minimize the time spent in coding the program and fully automate the process of transformation. The proposed approach adopts the executable specification based on Petri Nets, where can be verified some of their properties such as deadlock, liveness, boundedness, reachability among others. Petri nets contain transitions that may have annotated codes in the C and NXC programming language, that perform specific parts of the system that is being modeled. Additionally, the annotated codes in the C programming language will be verified by a *Bounded Model Checker*, that it will test specific properties in the code, as a index of arrays, division by zero, safety pointers and other. This specification provides the basis for the user to display the functionality of the system that will be modeled, giving the user an overview of the specific features of the system. Thus, this methodology contributes to developers and software engineers in the generation of prototypes that represent an executable specification, facilitating the evaluation of different models and helping to reduce differences of interpretation in the construction of software. This work also presents a tool called PNTCG (*Petri Net Tool for Code Generation*) developed based on this methodology and a case study based on a prototype automation product packaging, which uses a conveyor belt and a robot arm to demonstrate the use and application of this methodology.

Keywords: Petri Net, Bounded Model Checker, Compilers, Prototyping.

Lista de Figuras

2.1	Exemplo de Rede de Petri	9
4.1	Arquitetura	25
4.2	Interface da Ferramenta PNTCG	25
4.3	Código .pnt gerado pelo parser	27
4.4	Checagem das propriedades da Rede de Petri pelo INA	28
4.5	Exemplo de Rede de Petri	28
4.6	Transições com código em detalhes	29
4.7	Resultado da verificação dos códigos nas transições pelo ESBMC	30
4.8	Código .cxb gerado pela Interface Gráfica	31
4.9	Código léxico do Compilador	32
4.10	Trecho do código sintático	33
4.11	Interface gráfica do Trace	34
4.12	Arquivo com o trace salvo	35
4.13	Representação da Estrutura de Repetição <i>WHILE</i>	35
4.14	Representação da Estrutura de Dados <i>IF</i>	36
4.15	Código gerado pelo método	38
4.16	Exemplo do <i>template</i> utilizado nesse trabalho	38
5.1	A esteira e o Braço Robô	40
5.2	Movimento do robô a esquerda levando a bola vermelha	40
5.3	Movimento do robô a direita levando a bola azul	41
5.4	Redes de Petri do problema Esteira - Braço Robô	42
5.5	Códigos anotados nos lugares e transições	43
5.6	Resultado da checagem das propriedades da redes de Petri	44
A.1	Parte do Algoritmo A	70
A.2	Algoritmo B	71
A.3	Algoritmo C	72

Sumário

Agradecimentos	v
Resumo	vii
Abstract	viii
Lista de Figuras	ix
1 Introdução	1
1.1 Contexto	3
1.2 Definição do Problema	4
1.3 Motivação	5
1.4 Objetivos	6
1.4.1 Objetivo Geral	6
1.4.2 Objetivos Específicos	6
1.5 Organização do trabalho	7
2 Conceitos e Definições	8
2.1 Redes de Petri	8
2.1.1 Redes Elementares	10
2.1.2 Extensões às Redes de Petri	10
2.1.3 Propriedades da Redes de Petri baseado no <i>Integrated Net Analyzer</i>	11
2.2 Model Checking	16
2.3 Prototipação de <i>Software</i>	17
2.4 Sistemas Embarcados	18
2.5 Compiladores	19
3 Trabalhos Correlatos	21
4 Método Proposto	24

4.1	Checagem das propriedades de Rede de Petri	24
4.1.1	Interface Gráfica	24
4.2	Verificação formal de código anotado na linguagem C	27
4.3	Compilador - Análise Léxica e Sintática	29
4.3.1	Linguagem PNTCG	30
4.3.2	Análise Léxica	32
4.3.3	Análise Sintática	33
4.4	Geração de Código	33
4.4.1	Especificação de um <i>trace</i>	33
4.4.2	Geração de código com base na Rede de Petri	35
4.4.3	Geração do código do <i>WHILE</i>	36
4.4.4	Geração do código da estrutura <i>IF</i>	36
4.4.5	Geração do código da Rede de Petri demonstrada nesse Método Proposto	37
5	Resultados Experimentais	39
5.1	Estudo de caso: O problema: robô e esteira.	39
5.1.1	Redes de Petri	41
5.1.2	Checagem de propriedades da Redes de Petri	44
6	Conclusões e Trabalhos Futuros	45
6.1	Considerações Finais	45
6.2	Trabalhos Futuros	46
	Referências Bibliográficas	48
A	Apêndice	51

Capítulo 1

Introdução

Dia após dia, nossas vidas se tornam mais dependentes de sistemas embarcados. Isso inclui não apenas sistemas críticos de segurança, isto é, automóveis, ferrovias, aviões, foguetes espaciais, dispositivos médicos, robôs e esteiras industriais, mas também de automação residencial, consoles de video game, eletrodomésticos, *drives* de discos, impressoras de rede, caixas eletrônicos, telefones celulares, e assim por diante. Devido a esta diversidade de aplicações, os projetos de sistemas embarcados podem estar sujeitos a diferentes restrições, como por exemplo, tempo de projeto, tamanho, peso, consumo de energia, confiabilidade e custo [Barreto, 2005].

Nestes diferentes ambientes, aplicações de softwares necessitam ser desenvolvidas rapidamente e atender um alto nível de qualidade. Os métodos formais desempenham um importante papel para mensurar a previsibilidade e dependência no projeto de aplicações crítica. Como exemplo, podemos citar as redes de Petri para modelagem dos respectivos sistemas, objetivando a verificação das propriedades dos mesmos, assim como, o seu devido comportamento.

O uso de métodos formais no desenvolvimento de software apresenta várias vantagens, por exemplo, programas (ou protótipos) podem ser gerados automaticamente e formalmente a partir das suas especificações. Tais protótipos, algumas vezes chamados

de especificações executáveis, servem de base para exibir ao usuário as funcionalidades do futuro sistema. Pode-se provar também que programas satisfazem determinadas propriedades, e que um programa é uma realização da sua especificação Moura [1995] apud Freitas [2010].

A especificação executável permite evitar inconsistências, erros e garantir a completude do sistema. Esse processo assegura uma interpretação não ambígua para a especificação do sistema, valida a funcionalidade do sistema antes do início da sua implementação, cria modelos de desempenho do sistema e avalia a performance do sistema [Synopsys, 2003].

Segundo [Budde et al., 1992], protótipos adequados fornecem aos usuários e gestores uma idéia tangível das soluções dos problemas. Para os desenvolvedores, os protótipos que constituem uma especificação executável que facilita avaliação de diferentes modelos e ajuda a reduzir as diferenças de interpretação na construção de softwares [Alcoforado, 2007].

Este trabalho gera uma especificação executável baseada em um modelo formal conhecido como redes de Petri, no qual, é possível checar propriedades específicas e características de uma rede de Petri, por exemplo, se a rede é segura, limitada, ordinária, pura, possui *deadlock*, entre outras. Nesta, contém transições e lugares que possuem códigos anotados na linguagem C ou na linguagem Not Exactly C (NXC), sendo que o código C passará por uma verificação formal usando um *Bounded Model Checker* para comprovar o comportamento e assegurar as propriedades definidas no código e, com base nestes itens é efetuada a geração do código.

A técnica de verificação formal conhecida como “checagem de modelos” (do inglês *model checking*) é uma técnica para verificação de sistemas concorrentes de estados finitos [Clarke et al., 1999]. Normalmente o *model checking* usa uma busca exaustiva no espaço dos estados do sistema, para determinar se alguma propriedade é verdadeira ou não, onde este procedimento será sempre finalizado com uma resposta positiva ou negativa [Clarke et al., 1999]. O *model checking* é uma técnica que pode ser aplicada

para comprovar o comportamento e especificações de softwares.

Neste contexto, existe um tipo especial de *model checker* denominado de *Bounded Model Checker* (BMC) baseado em *Boolean Satisfiability Problem* (SAT). A idéia básica de um BMC é verificar (a negação de) uma dada profundidade do modelo (ou código)[Rocha et al., 2010]. Neste trabalho é utilizado *Bounded Model Checker* ESBMC.

Este trabalho também apresenta uma ferramenta denominada de PNTCG (*Petri Net Tool for Code Generation*), e um estudo de caso de um protótipo de automatização de embalagens de produto, no qual, é utilizado uma esteira e um braço robô para demonstrar a utilização e aplicação da metodologia proposta.

Este Capítulo apresenta o contexto deste trabalho, descreve o problema a ser resolvido, apresenta a motivação e objetivos, explica como o problema está planejado para ser resolvido e finalmente, mostra a estrutura do presente trabalho.

1.1 Contexto

Para resolver problemas reais, um desenvolvedor de software ou uma equipe de desenvolvedores, deve incorporar uma estratégia de desenvolvimento que abrange as camadas de processo, os métodos e ferramentas. Essa estratégia é frequentemente referida como modelo de processo ou paradigma de engenharia de software [Pressman, 2002]. Existe uma variedade de modelos de processo para engenharia de software, como: ciclo de vida clássico ou modelo cascata ou também conhecido como modelo sequencial linear, modelos de prototipagem, modelos RAD (*rapid application development*), modelos de processo de software evolucionários, modelos baseado em componentes, modelos baseado em métodos formais e técnicas de quarta geração. Neste trabalho, abordaremos o processo de prototipagem, que busca reduzir os erros introduzidos na análise de requisitos minimizando os riscos do projeto, onde o protótipo pode estabelecer como o modelo funciona previamente, e também determinar a viabilidade do produto, dentre

outros. O contexto deste projeto está baseado no desenvolvimento rápido de sistemas escritos na linguagem C e/ou e na linguagem NXC.

Embora os métodos formais sejam considerados difíceis de se utilizar na prática, estes são uma tendência atual para o desenvolvimento de sistemas de segurança crítica. Estes colaboram significativamente para a compreensão do sistema, revelando inconsistências, ambiguidades e incompletude que poderiam passar despercebidos. Através das redes de Petri, obtém-se mais confiabilidade nas modelagens dos projetos. Existem hoje várias extensões para as Redes de Petri ordinárias, tais como as Redes de Petri Coloridas, Temporais e Hierárquicas. Este trabalho foi baseado nas Redes de Petri básica, que consiste de lugares, transições e arcos dirigidos.

1.2 Definição do Problema

As redes de Petri contribuem para a representação de situações complexas, associadas com a grande flexibilidade que este tipo de rede apresenta, torna-se uma ferramenta muito útil para modelar, analisar e controlar sistemas complexos. Segundo [Palomino, 1995], se comparada com outros modelos gráficos de comportamento dinâmico, como as máquinas de estados finitas [Lee & Yannakakis, 1994], as redes de Petri oferecem muitas facilidades para expressar o comportamento de sistemas que são assíncronos e distribuídos. Dentre as principais vantagens das redes de Petri pode-se citar: (i) Modelos de redes de Petri representam uma ferramenta de modelagem hierárquica com uma matemática bem definida e fundamento prático; (ii) Redes de Petri nos permitem representar sistemas, seguindo uma abordagem *top-down*, assim como, *bottom-up* em diferentes níveis de abstração e facilitando a sua decomposição em subsistemas funcionais, com uma clara interrelação entre estes subsistemas; e (iii) Redes de Petri satisfazem as necessidades de estado parcial, decomposição de sistemas, simplicidade, sincronização, concorrência entre tarefas, assim como facilitam a sua compreensão. Entretanto, de acordo com [Palomino, 1995], as redes de Petri também possuem como

desvantagens: (i) O maior problema que apresenta as redes de Petri é o fato de que, quando é requerido uma modelagem mais detalhada, o modelo tende a ser muito grande (explosão combinatória de estados), portanto, a sua análise torna-se complicada e (ii) Redes de Petri são uma ferramenta passiva, isto é, elas servem para detectar situações contrárias ao bom funcionamento de um sistema, mas não podem gerar uma solução.

O problema considerado neste trabalho é expresso na seguinte questão: Como a partir de uma Rede de Petri com códigos anotados na linguagem C e NXC nas transições e lugares, é possível gerar automaticamente código para executá-lo em ambiente de sistemas embarcados?

1.3 Motivação

Quando abordamos a prototipação de software objetivamos reduzir os erros provenientes da análise de requisitos, através de uma verificação antecipada da especificação do software que está em desenvolvimento. Assim, quando temos um protótipo executável é possível testar os estados do sistema, bem antes, do início da fase de implementação.

Com o interesse de maximizar a redução de erros inseridos na fase de análise dos requisitos funcionais de software, devemos ir além da prototipação e utilizar métodos formais, que são estruturados por linguagens e ferramentas para especificar e verificar sistemas, baseados em fundamentos lógicos matemáticos [Clarke & Wing, 1996]. A utilização de métodos formais busca principalmente aumentar a boa compreensão do software, revelando ambiguidades, inconsistências e falhas que podem, caso contrário, não serem detectadas. Além disso, essas especificações facilitam a modularização e o reuso no desenvolvimento do software [Rangel, 2006].

A utilização conjunta da prototipação de software com métodos formais enfatizam uma especificação categórica do problema, e expõe o usuário a um sistema utilizável o mais rápido possível, de modo que usuários e engenheiros de softwares sejam capazes de executar e validar as especificações dos requisitos funcionais do sistema. Assim,

os benefícios da especificação formal e da abordagem de prototipação para o desenvolvimento de software são combinados, e uma descrição de sistema bem estruturada, executável e sem ambiguidade pode ser desenvolvida.

1.4 Objetivos

1.4.1 Objetivo Geral

O principal objetivo deste trabalho é *demonstrar um método para a geração de protótipos rápidos e confiáveis com o uso de Redes de Petri, onde estes possam ter suas propriedades verificadas e analisadas.*

1.4.2 Objetivos Específicos

- Demonstrar uma metodologia para geração automática de código (na linguagem C ou NXC) a partir do formalismo de uma Rede de Petri;
- Demonstrar um método para se efetuar a verificação das propriedades de Redes de Petri;
- Propor um compilador para Redes de Petri com código anotado (na linguagem C ou NXC) nos lugares e transições;
- Provar que o código na linguagem C pode ser checado com um *Bounded Model Checker*;
- Validar o método aplicando em um estudo de caso, visando garantir que o mesmo gere protótipos rápidos e confiáveis.

1.5 Organização do trabalho

O Capítulo 2, apresenta os conceitos e definições de redes de Petri e propriedades, Model Checking, Prototipação de *Software*, Sistemas Embarcados e Compiladores. No Capítulo 3, é analisado os principais trabalhos correlatos. No Capítulo 4, é explicitado o método proposto com a arquitetura da solução do problema descrito neste trabalho. No Capítulo 5, é apresentado e analisado os resultados experimentais. E por fim no Capítulo 6, é visto as considerações finais e os trabalhos futuros nas conclusões.

Capítulo 2

Conceitos e Definições

Este Capítulo é dividido em cinco seções: Redes de Petri, Model Checking, Prototipação de Software, Sistemas Embarcados e Compiladores.

2.1 Redes de Petri

Redes de Petri são famílias de técnicas formais para a modelagem de sistemas [Murata, 1989]. As redes de Petri modelam ações e estados através de quatro tipos de entidades: lugares, transições, arcos e *tokens*. Os lugares são usados para representar estados ou condições, enquanto as transições representam ações ou eventos. Os arcos estabelecem a relação de dependência entre lugares (estados/condições), transições (ações/eventos) e vice-versa. Os *tokens* são marcadores que identificam quais lugares estão “ativos” no sistema, ou seja, a marcação dos *tokens* em uma Rede de Petri caracteriza o conjunto de estados internos do sistema. Cada uma dessas entidades possui uma identidade gráfica: os lugares são representados por circunferências; as transições, por retângulos, os arcos, por segmentos orientados, e os *tokens*, por círculos pretos. A rede resultante desses grafos é denominada redes de Petri.

A simulação de um sistema modelado por uma Rede de Petri é representada pelo consumo e criação de *tokens* por parte das transições. Os *tokens* são consumidos e

gerados pelas transições a cada mudança no sistema. O processo de consumo e geração dos *tokens* representa a dinâmica das mudanças de estados do sistema, a simulação gráfica desse processo é denominada *TokenGame* [Yakovlev, 2002].

Formalmente, as redes de Petri (*PN-Petri Nets*) (Figura 2.1) podem ser construídas a partir da seguinte definição [Murata, 1989]:

Definição - Redes de Petri

Uma Rede de Petri é uma quintupla, $PN = (P, T, F, W, M_0)$, onde:

$P = \{p_1, p_2, p_3, p_4, \dots, p_m\}$, um conjunto finito de lugares,

$T = \{t_1, t_2, t_3, t_4, \dots, t_n\}$, um conjunto finito de transições,

$F \subseteq (P \times T) \cup (T \times P)$, um conjunto de arcos,

$W: F \rightarrow \{1, 2, 3, \dots\}$, uma função de atribuição de pesos dos arcos,

$M_0: P \rightarrow \{0, 1, 2, 3, \dots\}$, uma função de mapeamento da marcação inicial da rede, $P \cap T = \emptyset$, $P \cup T \neq \emptyset$.

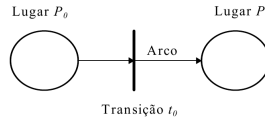


Figura 2.1. Exemplo de Rede de Petri

As redes de Petri podem ainda serem definidas como um formalismo capaz de especificar sistemas, agregando instrumentos de verificação formal [Murata, 1989]. A capacidade de simulação, permite que as redes de Petri possam ser usadas como linguagem de especificação, gerando especificações executáveis.

Com relação as transições em um modelo de redes de Petri, pode-se ter códigos anotados nas transições, isto é, ter a possibilidade de especificar uma ou mais tarefas para que seja executada quando uma transição for disparada. Como por exemplo, no trabalho de [Barreto, 2005], propôs uma modelagem de tarefas de um sistema usando uma transição anotada numa Rede de Petri Temporal (TPN), que é uma TPN com

código associado com transições.

2.1.1 Redes Elementares

Segundo [Brauer et al., 1987] e [Reisig & Rozenberg, 1998], as redes de Petri elementares constituem a versão atual, proposta para sintetizar as diversas variações que surgiram sobre o modelo proposto por Petri em 1962, conhecidas na literatura por redes clássicas ou Condição/Evento, porém, ainda, conservando suas características originais.

- Sequenciamento: é a rede que representa a execução de uma ação, desde que uma determinada condição seja satisfeita;
- Distribuição: é a rede elementar utilizada na criação de processos paralelos a partir de um processo pai. Os processos filhos são criados através da distribuição dos tokens encontrados no processo (lugar) pai;
- Junção: é a rede que modela a sincronização entre atividades concorrentes;
- Escolha Não-Determinística: é uma rede que ao se disparar uma transição, inabilita-se a outra. O fator não-determinístico dessa rede gera uma situação chamada de conflito;

2.1.2 Extensões às Redes de Petri

Nesta seção, são vistas algumas extensões propostas com a finalidade de aumentar a aplicação das redes de Petri. Segundo [Francês, 2003], podemos listar as redes de Petri coloridas, hierárquicas e redes de Petri Estocásticas.

- **Redes de Petri Coloridas**

As redes de Petri coloridas têm por objetivo reduzir o tamanho do modelo, permitindo que os *tokens* sejam individualizados, através de cores atribuídas a eles; assim,

diferentes processos ou recursos podem ser representados em uma mesma rede. As cores não significam apenas cores ou padrões. Elas podem representar tipos de dados complexos, usando a nomenclatura de colorida apenas para referenciar a possibilidade de distinção entre os *tokens*.

- **Redes Hierárquicas**

Um dos problemas apresentados nas redes de Petri originais é o fato que, à medida que o tamanho do sistema cresce, vai se tornando cada vez mais difícil manter a clareza do modelo. Na representação hierárquica, dois componentes são fundamentais para viabilizar uma representação em mais alto nível: a superpágina e a subpágina. A primeira representa um agrupamento de componentes (transições, lugares e arcos), visando gerar um modelo mais compacto e inteligível. Já as subpáginas, são o detalhamento de uma superpágina, de forma a esclarecer alguns detalhes omitidos na representação em alto nível.

- **Redes de Petri Estocásticas**

O tempo também pode ser usado de maneira probabilística, ou seja, o disparo das transições está associado a distribuições de probabilidade. Essas redes são denominadas Redes de Petri estocásticas, pois, seus comportamentos podem ser descritos por processos estocásticos.

2.1.3 Propriedades da Redes de Petri baseado no *Integrated Net Analyzer*

Nesta Seção é apresentada a ferramenta INA (*Integrated Net Analyser*), assim como, as suas principais siglas para análise das propriedades das redes de Petri.

O *Integrated Net Analyser* - INA [Roch & Starke, 1999] é uma ferramenta que contribui na análise de redes de Petri em diferentes tipos de investigação relacionada

a disparos na Rede e análise de propriedades gerais. Tipicamente, propriedades que podem ser verificadas através da análise da limitação dos lugares, vivacidades da transições e alcançabilidade das marcações ou estados. Segue as principais propriedades analisadas pela INA:

- Ordinária (**ORD** - *Ordinary*) - uma Rede de Petri diz ser ordinária, se a multiplicidade de cada arco é igual a um;
- Homogênea (**HOM** - *Homogenous*) - uma Rede de Petri se diz ser homogênea, se para qualquer lugar P , todos os arcos que começam em P tem a mesma multiplicidade;
- Multiplicidade Sem Bloqueio (**NBM** - *non-blocking multiplicity*) - se para cada lugar P , o mínimo de multiplicidades de arcos finais em P não é menos do que o máximo de multiplicidade de arcos que começam em P ;
- Pura (**PUR** - *pure*) - se existe uma transição na rede, para qual o pré-lugar, é também um pós-lugar. Neste caso, a rede não é pura, ou seja, não é livre de *loop*. Tais redes não possuem vivacidade sobre as regras de disparo seguro;
- Conservativa (**CSV** - *conservative*) - uma rede dita ser conservativa, se todas as transições adicionam exatamente muitos *tokens* aos seus pós-lugares como eles são subtraídos de seus pré-lugares. Em uma rede conservativa, o número total de *tokens* não é assim alterado pelos disparos em nenhuma transição. Logo, a quantidade de *tokens* é uma invariante. A rede é sub-conservativa, se todas as transições adicionam no máximo os mesmos *tokens* aos seus pós-lugares, como eles são subtraídos pelos seus pré-lugares. O total de número de *tokens* não pode ser aumentado;
- Livre de conflito estático (**SCF** - *static conflict free*) - se duas transições têm um comum pré-lugar, eles são um conflito estático sobre os *tokens* destes pré-lugares. Então a rede não é livre de conflito estático;

- Conectado (**CON** - *Connected*) - uma rede é dita ser conectada, se para cada nó na rede, existir um caminho não direcionado para cada outro nó. A direção dos arcos é assim ignorada;
- Fortemente Conectado (**SC** - *strongly connected*) - uma rede é conectada, se é checado em cada nó, também tem um caminho direcionado para cada outro nó, ou seja, a direção dos arcos é também considerada;
- Transição sem pré lugar (**Ft0** - *transition without pre-place*) - uma rede tem Ft0 transições, se existem transições sem um pré-lugar; $Ft = \emptyset$;
- Transição sem pós lugar (**tF0** - *transition without post-place*) - uma rede tem tF0 transições, se existem transições pós-lugar; $tF = \emptyset$;
- Lugar sem pré transição (**Fp0** - *place without pre-transition*) - uma rede tem Fp0 lugares se existem lugares sem pré-transições; $Fp = \emptyset$;
- lugar sem pós transição (**pF0** - *place without post-transition*) - uma rede tem pF0 lugares se existem lugares sem pós-transições; $pF = \emptyset$;
- Grafo marcado (**MG** - *marked graph*) - uma rede é um grafo de sincronização (ou marcado), se para cada lugar tem exatamente uma pré-transição e uma pós-transição. Aqui, múltiplos arcos são ignorados;
- Máquina de estado (**SM** - *state machine*) - uma rede é uma máquina de estado, se cada transição tem exatamente um pré-lugar e um pós-lugar. Aqui, multiplicidades de arcos são ignorados;
- Escolha-livre (**FC** - *free choice*) - uma rede tem escolha livre, se cada lugar compartilhado, é somente o pré-lugar de sua pós-transição;
- Escolha-livre estendida (**EFC** - *extended free choice*) - uma rede tem escolha livre estendida, se após transições de cada lugar compartilhado tem os mesmos pré-lugares;

- Extensão simples (**ES** - *extended simple*) - uma rede é extensão simples, se as seguintes condições são asseguradas: se dois lugares tem uma pós-transição em comum, então, para um deles, sua pós transição é também pós transição de um outro lugar; ou seja, um de seus lugares pode também ter outra pós transição;
- Propriedade de *Deadlock* (**DTP** - *deadlock-trap-property*) - um *deadlock* é um conjunto não vazio de lugares que não podem ser marcados novamente, uma vez que é limpo, por que toda transição no qual deve disparar o *token* para o lugar, neste conjunto tem um pré-lugar neste conjunto (então não pode disparar);
- Cobertura de Máquina de Estado (**SMC** - *state machine coverable*) - se um conjunto de componentes é dado, tal que, o correspondente de sub-redes defina a máquina de estado e sua união é igual ao conjunto de todos os lugares na rede;
- Máquina de estado Decomponível (**SMD** - *state machine decomposable*) - se um conjunto de componentes é dado, tal que, as correspondentes sub-redes definem ser máquinas de estados fortemente conectadas, e as suas uniões são iguais ao conjunto de todos os lugares na rede;
- Máquina de estado Alocáveis (**SMA** - *state machine allocatable*) - um componente é dito ser selecionado em um pré- lugar de alocação, se todas as transições que tem um pré-lugar no componente, um pré-lugar é atribuído;
- Abrangida por lugares invariantes (**CPI** - *covered by place invariants*) - uma rede é abrangida por lugares invariantes, se existe um P invariante no qual atribui um valor positivo para cada lugar;
- Abrangência por transições invariantes (**CTI** - *covered by transition invariants*) - uma rede é abrangida por transições invariantes, se existe um T invariante na qual, atribui um valor positivo para cada transição;

- Limitada (**B** - *Bounded*) - uma rede é limitada, se existir um número k , tal que, em qualquer marcação alcançável nunca existam mais que k tokens em um lugar;
- Limitada estruturalmente (**SB** - *structurally bounded*) - uma rede é limitada estruturalmente, se ela é limitada em cada marcação inicial;
- Reversível (**REV** - *reversible*) - uma rede é reversível se o estado inicial pode ser alcançado a partir de cada estado alcançável;
- Estados Alcançáveis mortos (**SMA** - *dead state reachable*) - um estado morto é alcançável, se um estado é alcançável, no qual, nenhuma transição pode disparar mais;
- Estados mal alcançáveis (**BSt** - *bad state reachable*) - se um estado satisfaz, um então chamado, predicado “mal”, ele não é mais desenvolvido, quando computado o estado do grafo;
- Transições mortas em marcação inicial (**DTr** - *dead transition at initial marking*) - este atributo indica se a rede tem transições mortas na marcação inicial;
- Livre de conflito dinâmico (**DCF** - *dynamically conflict free*) - uma rede é dita livre de conflito dinâmico, se nenhum estado é alcançável no qual, duas transições são habilitadas, e uma delas pode ser desabilitada pelo disparo da outra;
- Viva (**L** - *live*) - uma rede é viva se todas as suas transições são vivas na marcação inicial, ou seja, nenhum estado é alcançável no qual, a transição é morta;
- Viva quando ignora transições mortas (**LV** - *live when ignoring dead transitions*) - uma rede é viva ao ignorar transições mortas, se todas as suas transições, que ainda, não estão mortas na marcação inicial, estiverem vivas. As transições ignoradas podem ser consideradas como fatos não especificados;
- Viva e Segura (**L&S** - *live and safe*) - uma rede é viva e segura, se ela é viva em qualquer marcação alcançável, não mais que um token em um lugar;

- Fracamente viva (**WL** - *weakly live*) - uma rede colorida é fracamente colorida se tudo nas suas transições é fracamente viva, ou seja, para cada transição existe uma cor no qual, a transição na marcação inicial e
- Coletivamente viva (**CL** - *collectively live*) - uma rede colorida é coletivamente viva, se todas as suas transições são coletivamente vivas. Uma transição é coletivamente viva, se para cada estado alcançável existe uma cor, de tal forma que, em um estado alcançável a partir dessa marcação, a transição possa disparar nessa cor.

2.2 Model Checking

O uso de métodos formais colaboram significativamente para a compreensão do sistema, revelando inconsistências, ambiguidades e incompletude que poderiam passar despercebidos [Clarke & Wing, 1996].

A verificação de um sistema é um aspecto fundamental para a aplicação de métodos formais. Esta verificação consiste em fazer um modelo do sistema, que contenha suas propriedades mais relevantes. Segundo [Clarke et al., 1999], aplicação de testes em um modelo consiste de tarefas, como: modelagem, especificação e verificação.

O *C Bounded Model Checker* (CBMC) é um *model checker* que utiliza a técnica *model checking*, é usado para a verificação formal de programas em C.

Segundo [Rocha et al., 2010], no contexto de *model checking*, existe um tipo especial de *model checking* denominado de *bounded model checking* (BMC), baseado em *Boolean Satisfiability Problem* (SAT). A idéia básica do BMC é verificar (a negação de) uma dada propriedade em uma dada profundidade: dado um sistema de transições M , uma propriedade ϕ , e um limite (bound) k , o BMC desenrola o sistema k , vezes e traduz ele em uma condição de verificação (CV) ψ , tal que, ψ é satisfeita se e somente se ϕ tiver um contra-exemplo [Clarke & Kroening, 2006] de profundidade menor ou

igual a k .

ESBMC é um *bounded model checker* para softwares embarcados em ANSI-C baseado em *SMT solvers*, no qual permite: (i) verificar *software single* e *multithread* (com variáveis compartilhadas e alocadas); (ii) verificação aritmética de *underflow* e *overflow*, segurança de ponteiros, limites de *arrays*, atomicidade, *deadlock*, *data race* e assertivas providas pelo usuário; (iii) verificar programas que fazem o uso de *bit-level*, ponteiros, *structs*, *unions* e aritmética de ponto fixo. ESBMC utiliza uma versão modificada do CBMC no *front-end* para analisar o código ANSI-C e para gerar as CVs através de execução simbólica [Rocha et al., 2010].

As propriedades verificadas pelo ESBMC incluem segurança de ponteiros, limites de vetores, assertivas providas pelo usuário, divisão por zero, *overflow* e *underflow*, alocação de memória dinâmica e outras.

2.3 Prototipação de *Software*

Segundo [Rangel, 2003], apesar dos substanciais avanços dos métodos e ferramentas da Engenharia de *Software*, nos últimos anos, a análise de requisitos ainda, continua sendo um problema chave no desenvolvimento de softwares complexos. Um dos principais fatores que contribuíram para perpetuação deste problema, é a carência de validações logo no início do ciclo do desenvolvimento de *software*.

As abordagens de desenvolvimento de software que incorporam prototipação ganham respaldo, pois têm provado serem capazes de dinamicamente responder às mudanças nos requisitos dos usuários [Floyd, 1984].

Segundo [Pressman, 2002], em geral, qualquer aplicação que cria mostradores visuais dinâmicos, interage pesadamente com o usuário, ou exige algoritmos, ou processamento combinatório que deve ser desenvolvido de modo evolucionário, é candidata à prototipagem. Para que a prototipagem de *software* seja efetiva, um protótipo deve ser desenvolvido rapidamente, de forma que o cliente possa avaliá-lo e recomendar

mudanças.

Segundo [Pressman, 2002], para realizar a prototipagem rápida, três classes genéricas de métodos e ferramentas estão disponíveis:

- **Técnicas de quarta geração** (*fourth generation techniques, 4GT*) incluem um amplo conjunto de linguagens de relatório e de consulta a base de dados, geradores de programa e aplicações, e outras linguagens não procedimentais de muito alto nível. Como o 4GT permite ao engenheiro de *software* gerar código executável rapidamente, elas são ideais para a prototipagem rápida;
- **Componentes de *software* reusáveis**. Outra abordagem para a prototipação rápida é montar, em vez de construir o protótipo, usa um conjunto de componentes de *software* existentes;
- **Ambientes de especificação formal e prototipagem**. Nas últimas décadas, várias linguagens e ferramentas de especificação formais foram desenvolvidas para substituir técnicas de especificação em linguagem natural. Hoje, desenvolvedores dessas linguagens formais estão no processo de desenvolvimento de ambientes interativos.

2.4 Sistemas Embarcados

Os Sistemas Embarcados (ou *Embedded System*), pela sua natureza especialista, podem ter inúmeras aplicações [Deboni & Borba, 2007]. Segundo [Taurion, 2005], em praticamente todas as atividades humanas podemos identificar a presença de *softwares* embarcados, embora a grande maioria, passe despercebida por nós. Os exemplos são muitos, e em nosso dia-a-dia, usamos nas tarefas cotidianas, como nos celulares, nos sistemas embarcados nos automóveis (sistemas de freios, por exemplo), nas catracas eletrônicas e nos elevadores inteligentes.

Segundo [Ball, 1996], um sistema é classificado como embarcado, quando este é dedicado a uma única tarefa e interage continuamente com o ambiente a sua volta por meio de sensores e atuadores. A denominação "embarcado" vem do fato, de que estes sistemas são projetados geralmente para serem independentes de uma fonte de energia fixa como uma tomada ou gerador. As principais características de classificação deste sistema são, a sua capacidade computacional e sua independência de operação. Outros aspectos relevantes dependem dos tipos de sistemas, modos de funcionamento e itens desejados em aplicações embarcadas.

Com a rápida evolução tecnológica, mais e mais equipamentos e dispositivos estarão utilizando sistemas de *software* embarcado, sistemas estes que, estarão também se tornando cada vez sofisticados e complexos.

2.5 Compiladores

Segundo [Aho et al., 2008], linguagens de programação são notações para se descrever computações para pessoas e máquinas. O mundo conforme o conhecemos, depende de linguagens de programação, pois todo o *software* executado em todos os computadores foi escrito em alguma linguagem de programação. Mas, antes que possa ser executado, um programa primeiro precisa ser traduzido para um formato que lhe permita ser executado por um computador. Os sistemas de *software* que fazem essa tradução, são denominados compiladores.

Neste trabalho foi dado ênfase principalmente, na fase de análise sintática e léxica do compilador. Segundo [Aho et al., 2008], a sintaxe de uma linguagem de programação descreve a forma apropriada dos seus programas, enquanto a semântica da linguagem define o que seus programas significam, ou seja, o que cada programa faz quando é executado.

A análise sintática (ou *parsing*), é o processo para determinar como uma cadeia de terminais pode ser gerada por uma gramática. Logo, é o processo de analisar uma

sequência de entrada (lida de um arquivo de computador ou do teclado, por exemplo), para determinar sua estrutura gramatical segundo uma determinada gramática formal. Conforme [Aho et al., 2008], a análise léxica é o processo de analisar uma sequência de caracteres de entrada, compreendendo um único *token*, é chamada de *lexema*. Assim podemos dizer que, o analisador léxico isola do analisador sintático, a representação do lexema dos *tokens*. De outro modo podemos dizer que, é a forma de verificar determinado alfabeto.

Capítulo 3

Trabalhos Correlatos

Nesta seção são apresentados e discutidos os trabalhos propostos na literatura que estão relacionados com os tópicos que formam a base para o desenvolvimento deste trabalho.

A linguagem Pencil, segundo [Conway et al., 2002], foi projetada para ser simples, intuitiva, flexível, formal, poderosa e altamente modular. Possui uma modularidade e a integração Java, que permite aos programadores trabalharem para combinar módulos e verificação formal de forma integrada. A sintaxe simples (similar ao Java e C++) e o mecanismo de tradução automática, ajuda a reduzir erros de implementação. Pencil herda o modelo matemático de Redes de Petri de tal forma que o comportamento da aplicação pode ser descrito simples e economicamente. Esta combinação de características faz com que o pencil seja uma boa ponte entre modelagem e implementação. Este trabalho, especifica uma Petri Net somente num arquivo texto para depois validar a Petri Net especificada. Um dos diferenciais é que a abordagem proposta permite especificar uma Rede de Petri através de uma interface gráfica, e que se pode anotar código não somente nas transições, mas também nos lugares.

No trabalho de [Sibertin-Blanc, 2001], é demonstrado uma abordagem chamada de *CoOperative Objects (COO)*, que tem por objetivo ser uma ponte entre o formalismo e o projeto detalhado. Suporta ferramenta SYROCO (um compilador de objetos coo-

perativos) estendido. As descrições de Redes de Petri habilita e traduz modelos COO em classes C++. No *CoOperative Objects(COO)*, destaca um exemplo do problema dos filósofos no aspecto estático e dinâmico. O autor transforma conceitos de redes de Petri em Orientação Objeto. Ele fala que a definição de uma Rede de Petri vem com a definição de classes de objetos, usando uma linguagem de Programação Orientada a Objeto, e tokens são instâncias dessas classes. Para processar tokens, cada transição é provida com um pedaço de código: quando uma transição ocorre, este código é aplicado com os tokens envolvidos. Um objeto deve ser provido com uma estrutura de dados acessível pelas transições da redes de Petri. Neste trabalho, não se utiliza Orientação Objeto e nem C++, apenas baseia-se na ideia de código anotado (nas linguagens C e NXC) na transição e lugares numa Rede de Petri, adicionalmente podendo ser checado por um verificador formal.

No trabalho [Dezani, 2006] descreve-se um programa de geração automática de código para o microcontrolador 8051 da Intel, a partir de uma Rede de Petri, com o objetivo de minimizar o tempo gasto na codificação do programa e automatizar completamente este processo de transformação. Definiu-se o uso das redes de Petri Lugar/Transição como modelo de entrada, pois, mesmo tendo um modelo mais compacto, a Rede de Petri Colorida, quando transformada em código *Assembly* é, consideravelmente maior que o código *Assembly* gerado para a Rede de Petri Lugar/Transição. Mostra também que, o código gerado pelo programa corresponde, exatamente, ao modelo da rede e pode ser executado pela arquitetura-alvo. Contudo, nesta abordagem proposta não é gerado código *Assembly*, todavia, foi gerado códigos nas linguagens C e NxC e similarmente ao trabalho de Dezani [2006], foi utilizado como base para a geração de código de redes de Petri, deste modo, espera-se contribuir nesta área agregando funcionalidades como a verificação formal de código.

LOOPN++, utiliza o modelo de redes de Petri de alto nível orientado a objeto e contém um tradutor, que gera um código na linguagem de programação C++ referente à Rede de Petri. O programa gerado deve ser, portanto, compilado para ser executado

[Lakos & Keen, 1994]. Assim como a abordagem proposta neste trabalho, segundo [Dezani, 2006], o LOOPN++ possui apenas um compilador e não possui um simulador, porém, ao contrário deste trabalho, o código gerado pelo LOOPN++ não tem relação de comportamento direta com o modelo de Rede de Petri.

Capítulo 4

Método Proposto

Este Capítulo descreve o método proposto que visa uma metodologia para a geração de protótipos rápidos e confiáveis com o uso de redes de Petri, onde estes, possam ter suas propriedades verificadas e analisadas. O método proposto neste trabalho é formado por quatro etapas, identificadas na Figura 4.1 por linhas tracejadas em vermelho: (i) Checagem de propriedades de Rede de Petri; (ii) Verificação formal de código anotado na linguagem C; (iii) Compilador - Análise Léxica e Sintática; e (iv) Geração de Código. Estas etapas serviram de base para a solução do problema definido.

4.1 Checagem das propriedades de Rede de Petri

4.1.1 Interface Gráfica

Nessa primeira etapa, a ferramenta baseia-se no método proposto e permite a verificação das propriedades específicas e características de uma Rede de Petri tais como: vivacidade, alcançabilidade, reversibilidade, entre outras. Para isso, foi desenvolvido uma interface gráfica (conforme a Figura 4.2) para melhor interatividade com o desenvolvedor de *software* no momento de especificar uma Rede de Petri.

A ferramenta contém os seguintes botões para a especificação da rede de Petri:

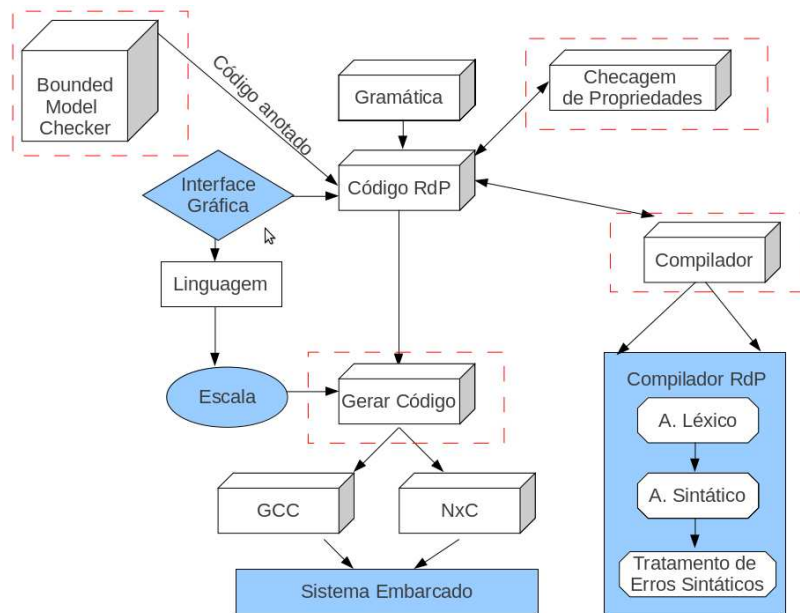


Figura 4.1. Arquitetura

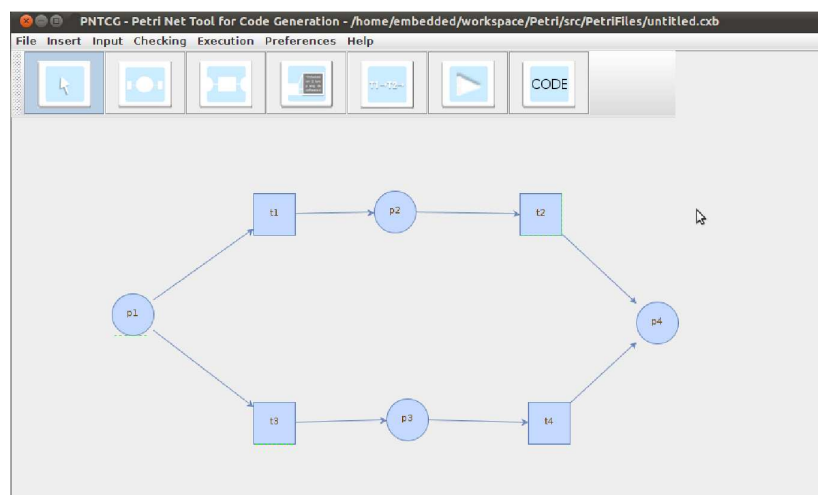


Figura 4.2. Interface da Ferramenta PNTCG

- Botão Manuseio (*hand button*): serve para manusear os objetos na interface, fazer ligações entre os objetos e ativação de outras opções;
- Botão Lugar (*place button*): serve para inserir o objeto lugar na rede de Petri;
- Botão Transição (*transition button*): serve para inserir o objeto transição na rede de Petri;
- Botão Propriedade (*property button*): serve para especificar *tokens* nos lugares e possibilitar anotar códigos nos lugares e transições;
- Botão Sequência de Disparos (*trace button*): serve para especificar a sequência de disparos das transições da rede de Petri;
- Botão Compilador (*Compiler button*): serve para efetuar a análise léxica e sintática na linguagem especificada na rede de Petri e
- Botão Código (*Code button*): serve para gerar automaticamente o código com base na rede de Petri.

A Figura 4.2 mostra uma Rede de Petri onde há quatro lugares, p1, p2, p3 e p4 e há quatro transições, t1, t2, t3 e t4. Na transição t1, tem um arco de entrada vindo de p1 e um arco de saída para p2. Na transição t2, tem um arco de entrada vindo de p2 e um arco de saída para p4. Na transição t3, tem um arco de entrada vindo de p1 e um arco de saída para p3. Na transição t4, tem um arco de entrada vindo de p3 e um arco de saída para p4. Clicando com o botão direito do mouse em cima de p1, acrescentamos um *token* a este lugar.

Após ser realizada a especificação da Rede de Petri, a mesma será salva em um arquivo com a extensão .cxb, após salva a especificação, o usuário efetua a checagem das propriedades da Rede, acionando na ferramenta o menu *Checking*, que chama a ferramenta INA para efetuar a checagem das propriedades. Este processo, aciona

um *parser*, isto é, converte o formato da PNTCG “.cxb” (os detalhes serão apresentados posteriormente) para o formato “.pnt” (formato utilizado pela ferramenta INA), conforme Figura 4.3. Com o resultado da checagem é emitido o resultado conforme apresentado na Figura 4.4. A Rede apresentada na Figura 4.2, obteve o seguinte resultado: A rede não é livre de conflito estático (SCF), é pura (PUR), é ordinária (ORD), homogênea (HOM), conservativa (CSV), é limitada estruturalmente (SB), é limitada (B), é uma máquina de estado (SM), escolha livre estendida (EFC), é extensão simples (ES), é escolha livre (FC), tem lugares sem pré-transições (Fp0), tem lugares sem pós transição (pF0), não é máquina de estado decomponível (SMD), não é máquina de estado alocáveis (SMA), é fortemente conectada (CON), não é viva (L) e não é viva e segura (L&S).

```

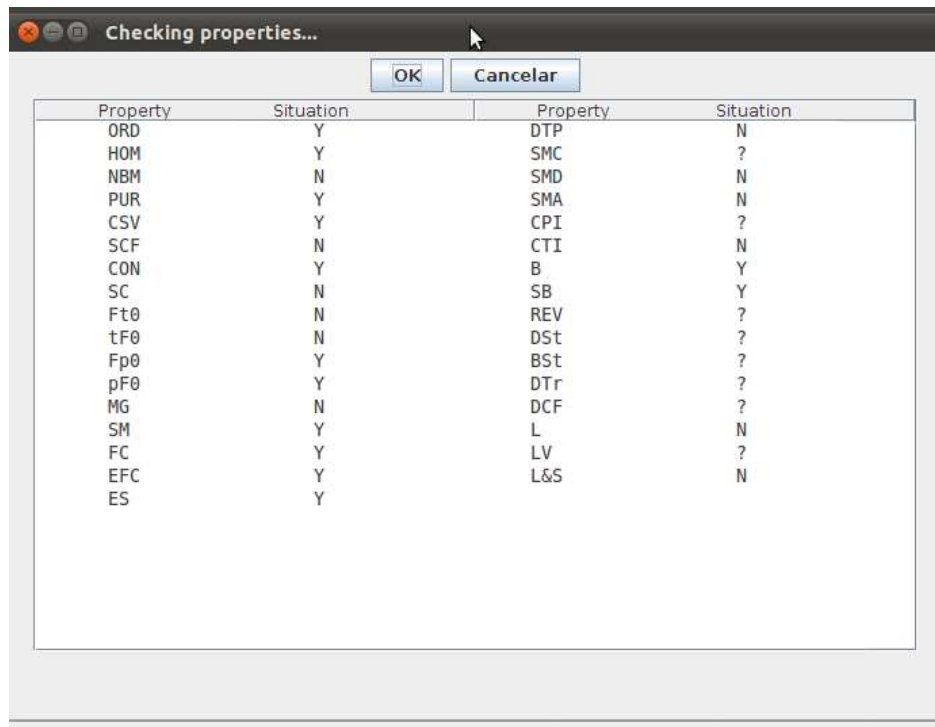
P    M    PRE,POST  NETZ  0:t
0  1  ,  0  2
1  0  0  ,  1
2  0  2  ,  3
3  0  1  3
@
place  nr.          name  capacity  time
      0:  p1          65535      0
      1:  p2          65535      0
      2:  p3          65535      0
      3:  p4          65535      0
@
trans  nr.          name  priority  time
      0:  t1           0         0
      1:  t2           0         0
      2:  t3           0         0
      3:  t4           0         0
@

```

Figura 4.3. Código .pnt gerado pelo parser

4.2 Verificação formal de código anotado na linguagem C

Nessa etapa, a verificação é efetuada nos códigos C anotados nas transições e/ou lugares da rede de Petri. Como exemplo, utilizaremos a Rede na Figura 4.5 que contém a



The screenshot shows a window titled "Checking properties..." with "OK" and "Cancelar" buttons. Below the buttons is a table with two columns: "Property" and "Situation". The table is divided into two sections by a vertical line.

Property	Situation	Property	Situation
ORD	Y	DTP	N
HOM	Y	SMC	?
NBM	N	SMD	N
PUR	Y	SMA	N
CSV	Y	CPI	?
SCF	N	CTI	N
CON	Y	B	Y
SC	N	SB	Y
Ft0	N	REV	?
tF0	N	DSt	?
Fp0	Y	BSt	?
pF0	Y	DTr	?
MG	N	DCF	?
SM	Y	L	N
FC	Y	LV	?
EFC	Y	L&S	N
ES	Y		

Figura 4.4. Checagem das propriedades da Rede de Petri pelo INA

mesma estrutura da apresentada na Figura 4.2, todavia, neste exemplo, contém códigos anotados em todas as transições.

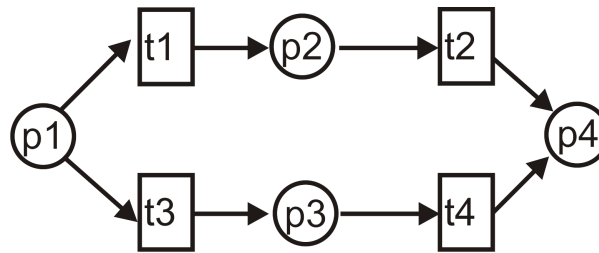


Figura 4.5. Exemplo de Rede de Petri

Logo em seguida, antes que os códigos sejam anotados nas transições, estes precisarão passar por uma checagem. O usuário entra na interface de checagem, e escolhe qual tipo de verificação que ele deseja fazer em todos os códigos anotados nas transições e/ou lugares. Após a realização da checagem, a Rede de Petri pode se apresentar conforme a Figura 4.6. Observa-se que, cada transição se apresenta com dois estados indicativos: vermelho e verde. Caso uma transição esteja na cor verde, a checagem

se realizou de forma satisfatória sem conter erros ou avisos. Caso contenha vermelho, indica um problema no código e permitirá que o usuário corrija. Uma alternativa prática a esse método está na Figura 4.7, onde mostra uma interface com duas colunas com as transições descritas na primeira coluna, onde mostram códigos anotados e a situação da verificação pelo ESBMC. Somente o código na transição t1 foi verificado com sucesso, os demais códigos nas outras transições falharam na verificação.

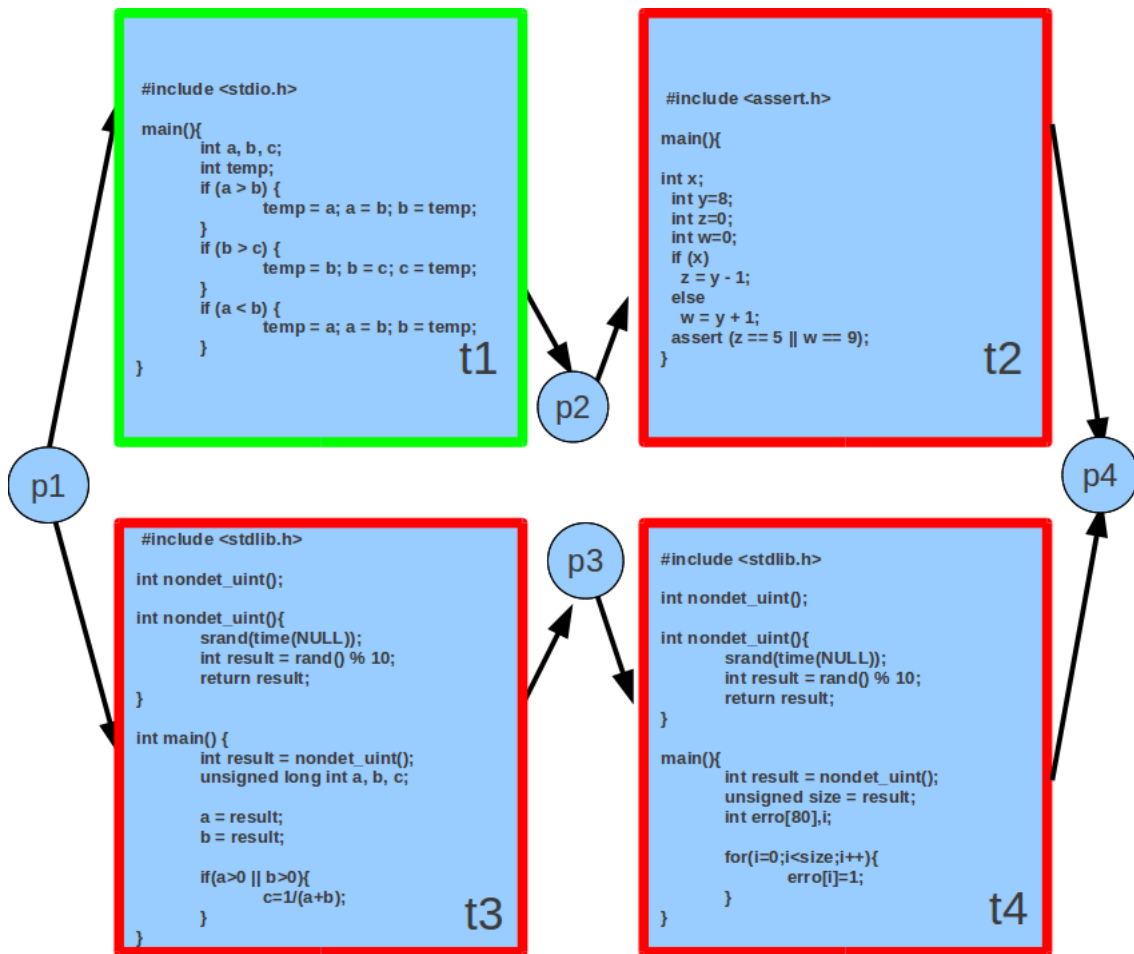
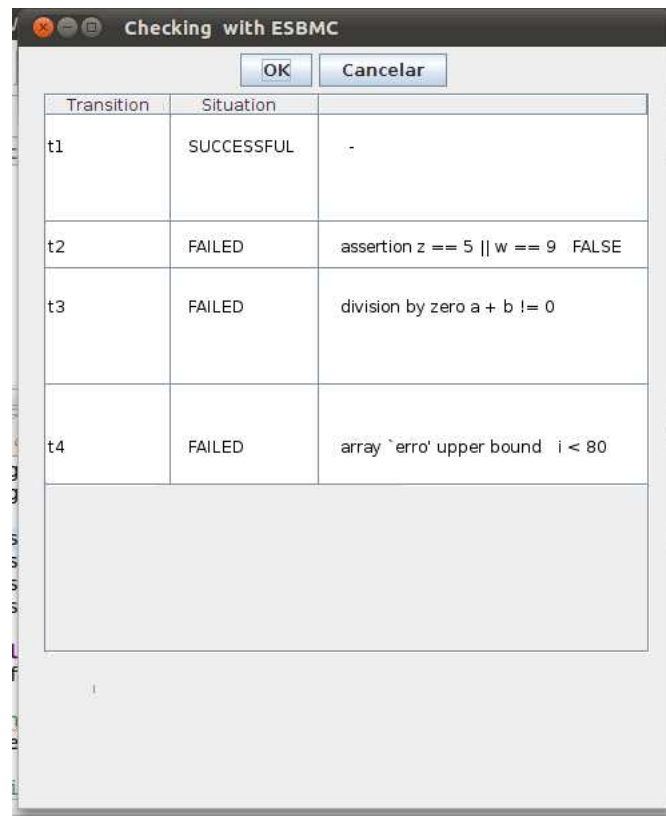


Figura 4.6. Transições com código em detalhes

4.3 Compilador - Análise Léxica e Sintática

Neste trabalho foi implementado um compilador. Este aceita, como entrada, uma Rede de Petri. Para aplicar as fases do Analisador Léxico e Analisador Sintático, as



Transition	Situation	
t1	SUCCESSFUL	-
t2	FAILED	assertion <code>z == 5 w == 9</code> FALSE
t3	FAILED	division by zero <code>a + b != 0</code>
t4	FAILED	array <code>'erro'</code> upper bound <code>i < 80</code>

Figura 4.7. Resultado da verificação dos códigos nas transições pelo ESBMC

ferramentas *FLEX* e *BISON* [Aaby et al., 1996] respectivamente com esses objetivos.

4.3.1 Linguagem PNTCG

A linguagem PNTCG foi desenvolvida baseada na definição formal de redes de Petri, ou seja, utiliza conjuntos para definir lugares, transições, arcos e *tokens*. O trabalho de [Conway et al., 2002] foi uma das bases para construção dessa linguagem.

A linguagem PNTCG pode ser representada na Figura 4.8 (os lugares e as transições no exemplo, foram especificados na interface gráfica conforme Figura 4.2).

Primeiramente, na linha 1 encontra-se a identificação do projeto precedido da palavra *net*, depois é especificado todos os lugares compostos pela palavra *place* seguido de sua identificação. Na linha 02 da Figura 4.8, a notação `p1(=1)` significa que `p1` tem uma marcação inicial (um *token*, que é o começo). Logo em seguida, nas linhas

```

01) net mp;
02) place p1(=1){
03) <%
04) %>
05) }
06) place p2{
07) <%
08) %>
09) }
10) place p3{
11) <%
12) %>
13) }
14) place p4{
15) <%
16) %>
17) }
18) transition t1{
19) in: p1;
20) out: p2;
21) <%
22) %>
23) }
24) transition t2{
25) in: p2;
26) out: p4;
27) <%
28) %>
29) }
30) transition t3{
31) in: p1;
32) out: p3;
33) <%
34) %>
35) }
36) transition t4{
37) in: p3;
38) out: p4;
39) <%
40) %>
41) }

```

Figura 4.8. Código .cxb gerado pela Interface Gráfica

03 e 04 será anotado o código que estará entre os caracteres `< %` e `% >`, e isso se aplica aos demais lugares especificados. E por fim, temos as transições, onde na linha 18, apresenta-se com a palavra *transition* seguido de sua identificação. Na linha 19, contém a palavra *in* que demonstra os lugares que são ligados pelo arco de entrada, e o *out*, que mostra os lugares que são ligados pelo arco de saída da transição. Ainda neste bloco, pode-se identificar os caracteres `< %` e `% >`, no qual, é um espaço reservado para a inserção de códigos anotados, e isso se aplica as demais transições especificadas.

4.3.2 Análise Léxica

Na Figura 4.9, temos o código a ser verificado pelo FLEX, objetivando a análise léxica.

```
%{
#include "p.tab.h" /* The tokens */
#include <string.h> /* strdup */
}%
DIGIT [0-9]
ID [a-zA-Z][a-zA-Z0-9]*
LT (0|[1-9][0-9]*)
%{
/*Regras*/
}%
%x cblock
%%
"{" { printf("%s\n", "LCURLY"); return LCURLY; }
"}" { printf("%s\n", "RCURLY"); return RCURLY; }
"(" { printf("%s\n", "LPAREN"); return LPAREN; }
")" { printf("%s\n", "RPAREN"); return RPAREN; }
"[" { printf("%s\n", "LBRACK"); return LBRACK; }
"]" { printf("%s\n", "RBRACK"); return RBRACK; }
":" { printf("%s\n", "COLON"); return COLON; }
";" { printf("%s\n", "SEMI"); return SEMI; }
"," { printf("%s\n", "COMMA"); return COMMA; }
"." { printf("%s\n", "DOT"); return DOT; }
"=" { printf("%s\n", "EQUALS"); return EQUALS; }
boolean { printf("%s\n", "BOOLEAN"); return BOOLEAN; }
byte { printf("%s\n", "BYTE"); return BYTE; }
char { printf("%s\n", "CHAR"); return CHAR; }
const { printf("%s\n", "CONST"); return CONST; }
double { printf("%s\n", "DOUBLE"); return DOUBLE; }
fire { printf("%s\n", "FIRE"); return FIRE; }
float { printf("%s\n", "FLOAT"); return FLOAT; }
immediate { printf("%s\n", "IMMEDIATE"); return IMMEDIATE; }
in { printf("%s\n", "IN"); return IN; }
int { printf("%s\n", "INT"); return INT; }
long { printf("%s\n", "LONG"); return LONG; }
net { printf("%s\n", "NET"); return NET; }
onCall { printf("%s\n", "ONCALL"); return ONCALL; }
out { printf("%s\n", "OUT"); return OUT; }
package { printf("%s\n", "PACKAGE"); return PACKAGE; }
place { printf("%s\n", "PLACE"); return PLACE; }
public { printf("%s\n", "PUBLIC"); return PUBLIC; }
ret { printf("%s\n", "RETURN"); return RETURN; }
short { printf("%s\n", "SHORT"); return SHORT; }
transition { printf("%s\n", "TRANSITION"); return TRANSITION; }
{ID} { printf("%s\n", "IDENTIFIER"); return IDENTIFIER; }
{LT} { printf("%s\n", "LITERAL"); return LITERAL; }
[ \n\t]+ /* Ignore espaços e tabulações */
("%>".*) /* Comentários de uma linha */
"<%" {BEGIN(cblock);
printf("%s\n", "CBLOCK"); return CBLOCK; }
<cblock>"%"+"^>\n)* /* Desconsidera qualquer
coisa que não seja um traço , */

<cblock>"%"+">" { BEGIN(INITIAL); }
. printf("Símbolo %s inexistente\n", yytext); }
%%
```

Figura 4.9. Código léxico do Compilador

4.3.3 Análise Sintática

Na Figura 4.10, temos o código a ser verificado pelo BISON, o analisador sintático. Esse código representa trecho fundamental e diferencial da gramática criada neste trabalho, pois, é mostrado uma área para o código (*Cblock*) ser anotado em um lugar. Na referida figura mostra que *PlaceDef* é composto de *LCURLY* (parêntese aberto), seguido por *PLACEAttrList* (um conjunto de definições do *Place* (lugar)), juntamente com o *Cblock* (onde se pode especificar código anotado) e *RCURLY* (parêntese fechado).

```

PlaceDef: LCURLY PlaceAttrListx CBlock RCURLY
;
PlaceAttrListx: PlaceAttrList | /* aceita regra vazia agora */
;
PlaceAttrList : InputList
               | OutputList
               | PlaceAttrList2
               | PlaceAttrList3
;
PlaceAttrList2 : InputList PlaceAttrList
;
PlaceAttrList3 : OutputList PlaceAttrList
;
InputList : IN COLON InArcList SEMI
;
OutputList : OUT COLON OutArcList SEMI
;

```

Figura 4.10. Trecho do código sintático

4.4 Geração de Código

Esta etapa detalha a geração automática de código usando a especificação executável de redes de Petri modelada e verificada nas etapas anteriores.

4.4.1 Especificação de um *trace*

Uma escala (ou *trace*), é uma sequência de disparos de transições de uma Rede de Petri. Através dessa informação, o usuário pode especificar o caminho que o *token* deve percorrer na rede modelada. Para produzir o *trace* que será armazenado em um arquivo “.txt”, o usuário utilizará a *interface* gráfica através do botão chamado *trace*

onde ao clicá-lo, acionará uma outra interface para especificar o trace como mostrado na Figura 4.11. De forma automática, será inserido no campo da esquerda todas as transições contidas na Rede de Petri especificada anteriormente. Sendo assim, o usuário pode escolher quais transições serão disparadas. As transições escolhidas estarão no campo da direita. Estas informações, serão armazenadas em um arquivo com a extensão .xyt contendo o nome do projeto. No exemplo mostrado neste capítulo, após gerado o *trace*, será consultado o arquivo de texto que o contém juntamente com as posições dos objetos da interface gráfica como mostrado na Figura 4.12. O trace é especificado da seguinte forma: ele deve vir depois da última palavra (na última linha), precedido de uma palavra “tcr”, onde cada transição virá precedida dos caracteres “- >” (traço seguido com o sinal de maior), com exceção da primeira transição.

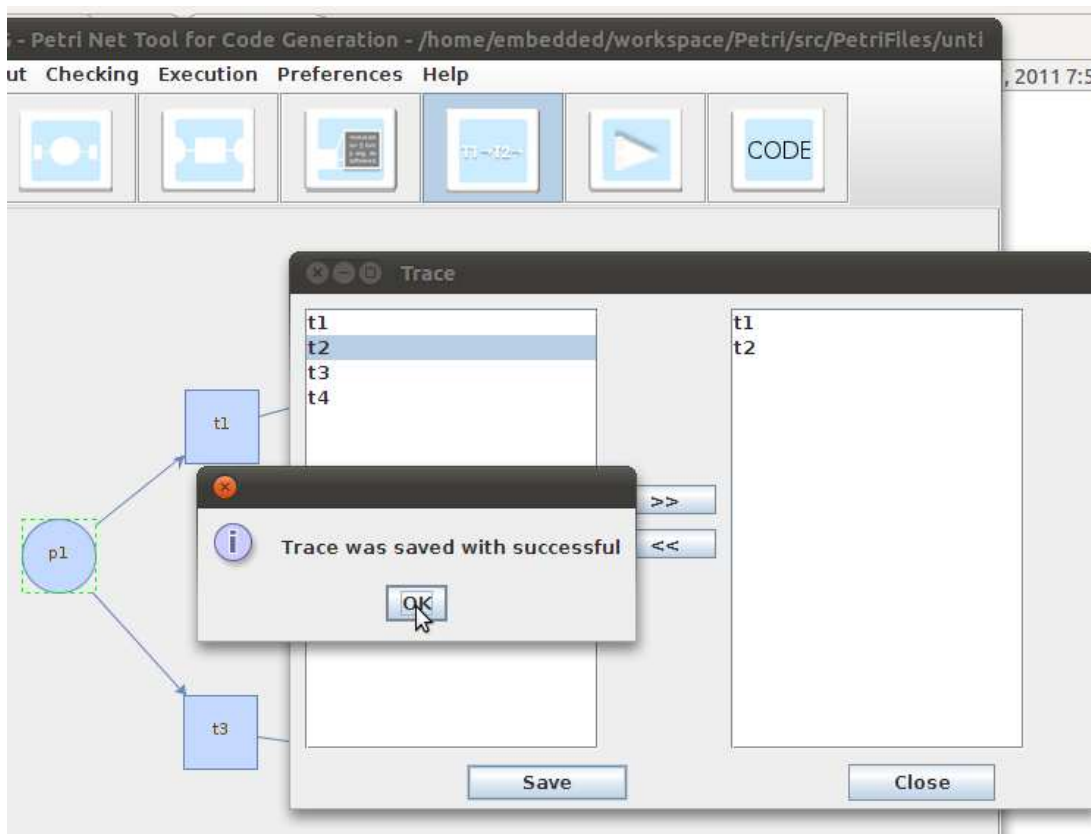


Figura 4.11. Interface gráfica do Trace

```

mp.xyt ✕
pos p1 130 290
pos p2 374 166
pos p3 380 430
pos p4 606 293
pos t1 241 202
pos t2 500 200
pos t3 240 410
pos t4 500 410
tcr t1->t2

```

Figura 4.12. Arquivo com o trace salvo

4.4.2 Geração de código com base na Rede de Petri

Na geração de código, um tratamento específico para as condições *WHILE* (Figura 4.13) e o *IF* (Figura 4.14) foram aplicados.

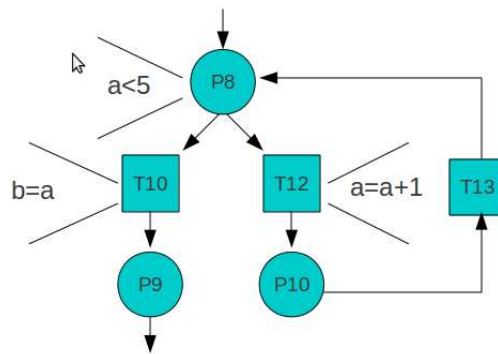


Figura 4.13. Representação da Estrutura de Repetição *WHILE*

É utilizado o algoritmo de busca em profundidade (ou DFS - *Depth-First Search*) [Cormen et al., 2001], através de um algoritmo recursivo principal, que iremos chamar de "A"(ver Apêndice A.1). Este algoritmo percorre todos os vértices (lugar e transição) e marca-os como visitados a partir da marcação inicial.

Este algoritmo ao encontrar um lugar, por exemplo p8, que contenha código, é feito uma nova busca em profundidade através do algoritmo B (ver Apêndice A.2), para verificar se esse lugar p8 é o vértice inicial de uma estrutura de dados *WHILE* ou de uma estrutura de dados *IF*, o algoritmo recursivo B percorre todos os vertices a partir de p8 e compara se cada um é igual ao vertice original p8. Caso encontre p8

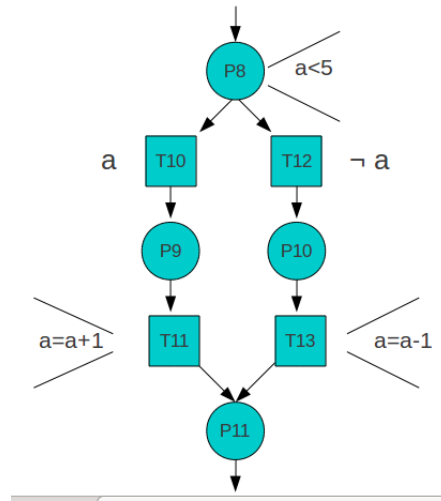


Figura 4.14. Representação da Estrutura de Dados *IF*

novamente, significa que, este é um vertice inicial de um *WHILE* (grafo cíclico), caso contrário, significa que $p8$ é o vértice inicial de um *IF*.

Como regra, os códigos com as condições do *IF* e do *WHILE* e também do fim do *IF* (*endif*) são anotados nos lugares e os demais códigos são anotados nas transições.

4.4.3 Geração do código do *WHILE*

No caso da estrutura do *WHILE*, o algoritmo recursivo A (ver Apêndice A.1) imprime o código de $p8$ (Figura 4.13) como sendo o condicional do *WHILE*, depois chama a si mesmo novamente para percorrer e imprimir o código de todos os vértices da direita que $p8$ aponta, até chegar novamente a $p8$ (já que é um grafo cíclico), como sendo o bloco de repetição do *WHILE*.

Depois da impressão do *WHILE*, o algoritmo recursivo A, é chamado para continuar a impressão do restante do grafo, a partir dos vértices da esquerda que $p8$ aponta.

4.4.4 Geração do código da estrutura *IF*

Na aplicação do algoritmo de busca em profundidade, se for identificada uma estrutura de dados *IF*, é chamado um novo algoritmo recursivo para percorrer todos os vértices

que p8 aponta e verifica qual o vértice (lugar) final do *IF*. Para identificar esse vértice final do *IF*, um algoritmo C (ver Apêndice A.3) percorre todos os vértices da esquerda de p8 (Figura 4.14) e os adiciona a uma lista de vértices visitados. Depois, esse algoritmo percorre todos os vértices da direita de p8 e verifica se algum deles consta na lista de vértices visitados. O vértice não visitado e que contém o código anotado *endif*, é o vértice final do *IF* e é guardado em uma variável.

O algoritmo principal A imprime o código de p8 como sendo o condicional do *IF*, depois chama-o de forma recursiva, percorrendo e imprimindo o código de todos os vértices da esquerda que p8 (Figura 4.14) aponta, até o vértice final do *IF* como sendo o bloco que será executado, caso o condicional do *IF* retorne verdadeiro. Depois, o algoritmo principal chama a si mesmo para percorrer e imprimir o código de todos os vértices da direita que p8 aponta até o vértice final, como sendo o bloco que será executado caso o condicional do *IF* seja falso.

Depois da impressão do *IF*, o algoritmo recursivo A é chamado para continuar a impressão do restante do grafo, a partir do vértice final do *IF*.

4.4.5 Geração do código da Rede de Petri demonstrada nesse Método Proposto

A geração do código da Rede de Petri, é efetuada por meio da gravação dos resultados obtidos com a aplicação dos algoritmos propostos nesta Seção, incluindo a estruturas *WHILE* e *IF*. É importante ressaltar, que a estrutura para a organização do código gerado é efetuada com base na aplicação de um *template* (conforme Figura 4.16), objetivando a organização dos códigos, deste modo como resultado final, obteremos o código gerado a partir da Rede de Petri, conforme mostrado na Figura 4.15.

Como item final nesta etapa de geração de código, o código é compilado, visando a sua validação, no que diz respeito, aos itens analisados pelos seus respectivos compiladores, ou seja, caso o código seja na linguagem C, é utilizado o compilador GCC, de

outro modo é usado um compilador NXC.

```
// bibliotecas
#include <stdio.h>
#include <assert.h>

//funções de cada código, se houver.

//funções que representam o "main" de cada código
t1(){
//código t1.c
    int a, b, c;
    int temp;
    if (a > b) {
        temp = a; a = b; b = temp;
    }
    if (b > c) {
        temp = b; b = c; c = temp;
    }
    if (a < b) {
        temp = a; a = b; b = temp;
    }
}

t2(){
    //código t2.c
    int x;
    int y=8;
    int z=0;
    int w=0;
    if (x)
        z = y - 1;
    else
        w = y + 1;
    assert (z == 7 || w == 9); //código corrigido pelo usuario.
}

main(){
    //Chamada da função que representam o "main" de cada código
    t1();
    t2();
}
```

Figura 4.15. Código gerado pelo método

```
<Bibliotecas dos códigos>

<Funções de cada código>//seguem o seguinte formato, <nome_código>_<função>

<Funções que representam o "main" de cada código>

main(){

    <Chamada da função que representam o "main" de cada código>

}
```

Figura 4.16. Exemplo do *template* utilizado nesse trabalho

Capítulo 5

Resultados Experimentais

Neste capítulo são descritos os testes realizados com o método proposto para a geração automática de código. Para realizar os testes, foi utilizado um microcomputador com processador AMD Athlon II X2 M300, 2.0 GHz e 4 GB de memória RAM.

5.1 Estudo de caso: O problema: robô e esteira.

O objetivo deste estudo de caso, é modelar uma linha de produção formada por uma esteira e um braço robô, usando redes de Petri contido no trabalho de [Pinto et al., 2009]. Na linha de produção (Figura 5.1), a esteira leva as bolas ao robô, este processo é modelado usando redes de Petri. A esteira leva as bolas de duas cores (vermelhas e azuis). Em uma posição da esteira, há um sensor de luz que detecta se a bola chegou até essa posição e, quando a referida detecção ocorrer, a esteira para de funcionar. Com a esteira parada, um braço robô (posicionado acima da bola detectada), abaixa, pega a bola, e sobe alguns centímetros para não colidir com a esteira. Como a bola foi retirada, o sensor não a detecta mais, então, a esteira volta ao funcionamento, até que outra bola esteja na posição de retirada.

No braço robô, há outro sensor que determina a cor da bola, que o mesmo está segurando. Quando a bola é vermelha (Figura 5.2), o braço se desloca para esquerda,

onde está uma caixa específica para as bolas vermelhas, e larga a bola. Mas, se a bola é azul (Figura 5.3), o braço se desloca para a direita, onde está um caixa específica para as bolas azuis e larga a bola. Depois disso, o braço volta para a posição central, ficando pronto para pegar outra bola. A Figura 5.1 abaixo, mostra um exemplo, de implementação dessa especificação usando o *Kit Lego Mindstorms NXT*¹.

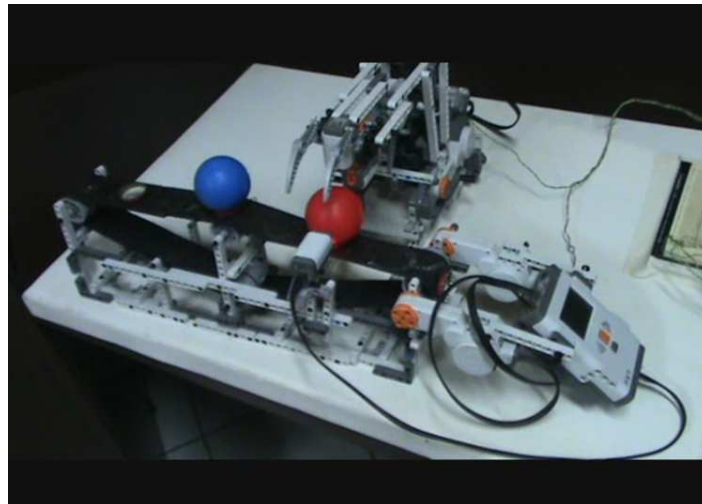


Figura 5.1. A esteira e o Braço Robô



Figura 5.2. Movimento do robô a esquerda levando a bola vermelha

¹<http://mindstorms.lego.com/>

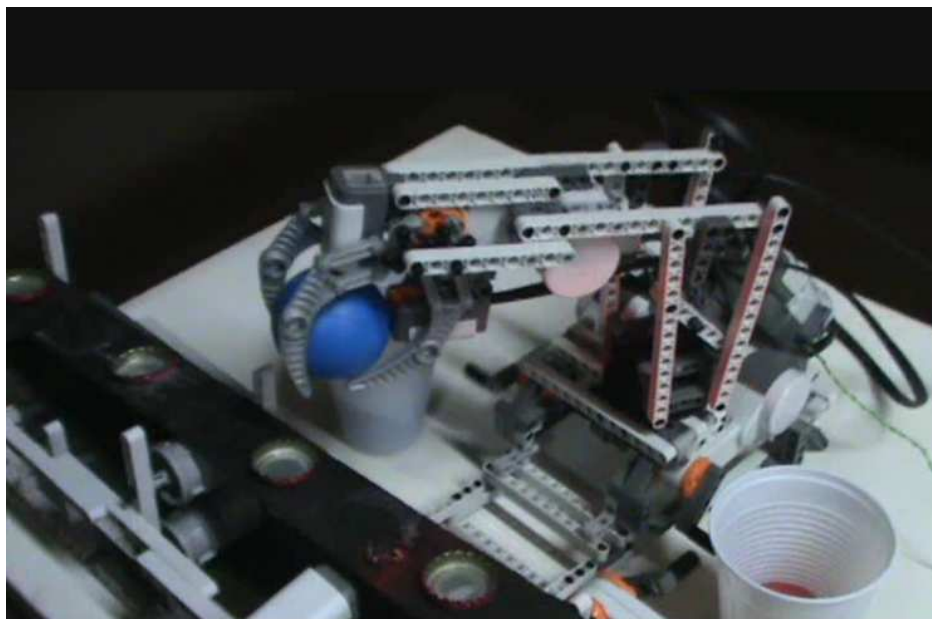


Figura 5.3. Movimento do robô a direita levando a bola azul

5.1.1 Redes de Petri

Especificamos o funcionamento da Rede de Petri da Figura 5.4. O estado p_0 , começa com um *token* que significa que a esteira está desligada. No estado p_2 , também fica representado que a esteira está desligada. No estado p_3 , é checado com o sensor se há bola em sua frente. Caso sim, a transição t_4 dispara um código para parar a esteira, onde o robô entra em ação. Caso contrário, a transição t_3 dispara um código para movimentar a esteira. No estado p_5 , a esteira permanece desligada e o braço robótico também. E pode também permanecer desligada no lugar p_7 , de acordo com o código anotado no lugar p_6 . Neste lugar, pode também ser o início de um estado de um processo repetitivo, onde o estado p_8 , através de um sensor, checa se tem alguma bola no sensor, caso negativo, o processo se repete, voltando para o estado p_6 . Caso positivo, no lugar p_9 , é um estado inicial de um *loop*, onde a transição t_{12} fará com que o braço robótico erga-se para não colidir com a esteira. Depois, fazemos uma nova checagem no lugar p_{10} . Este questiona a cor da bola. Caso seja azul, o lugar p_{12} é um estado inicial de um *loop* que fará movimento do braço robótico para sua

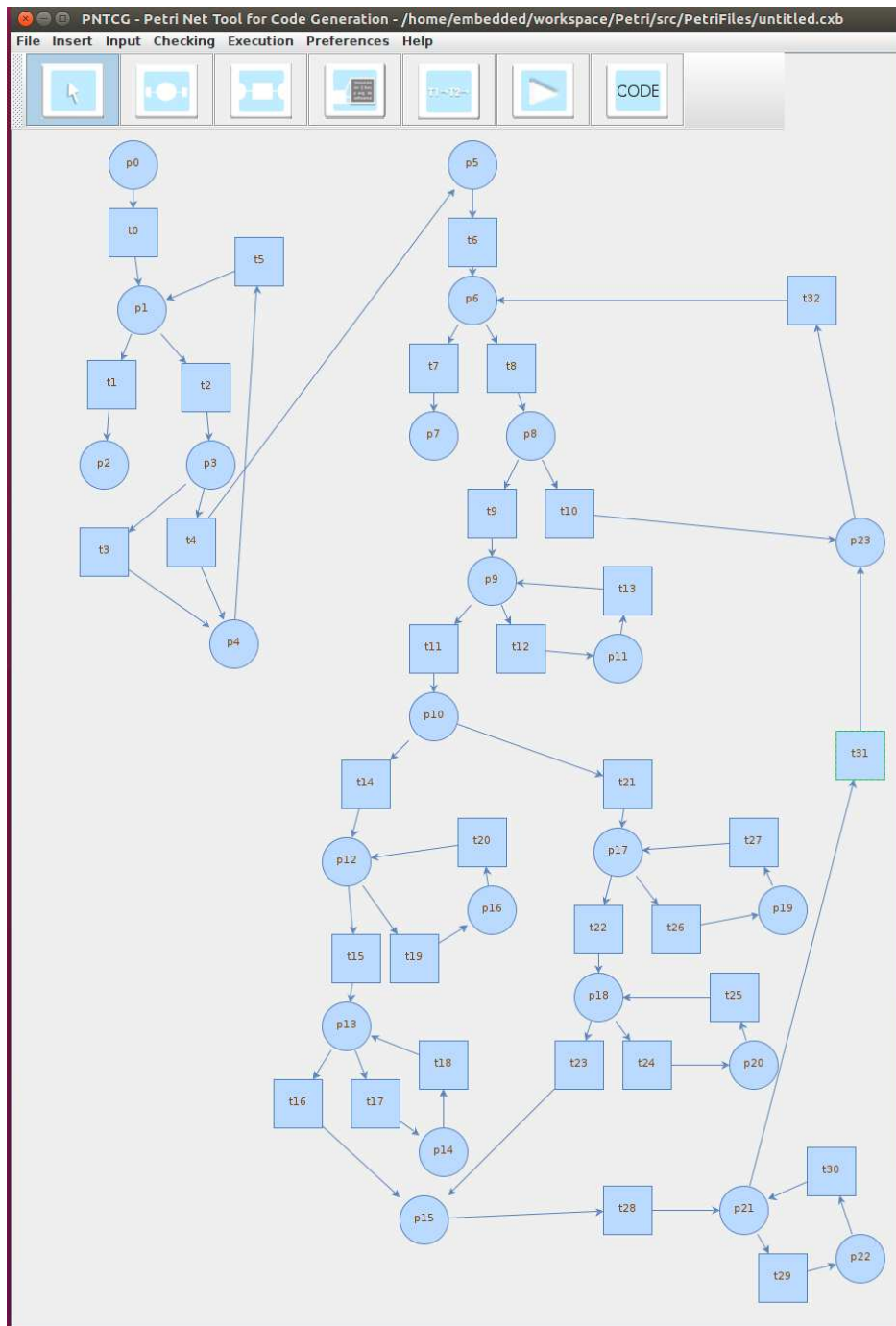


Figura 5.4. Redes de Petri do problema Esteira - Braço Robô

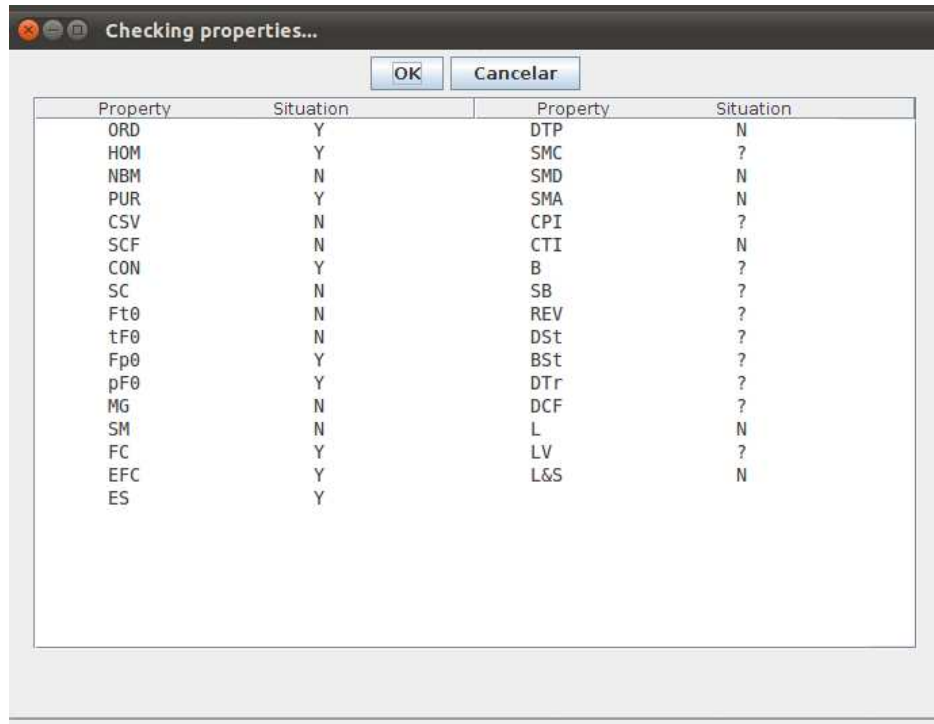
direita, através da transição t19, até se localizar abaixo do copo/caixa específica de bolas azuis. Chegando na transição t15, tem um código anotado que faz com que a garra do braço robótico abra, liberando a bola para cair no copo/caixa. No lugar p13, é um estado inicial do movimento de volta para a esquerda do braço do robô, onde

o código na transição t17 executa isso. Caso a bola seja vermelha, fará o processo contrário realizando com a bola azul, o lugar p17 é um estado inicial de um *loop*, que fará movimentos no braço robótico para sua esquerda, através da transição t26, até se localizar abaixo do copo/caixa específica de bolas vermelhas. Chegando na transição t22, tem um código anotado que faz com que a garra do braço robótico abra, liberando a bola para cair na caixa. No lugar p16, é um estado inicial do movimento de volta para a direita do braço do robô, onde o código na transição t24 executa isso. Passando pela entrega das bolas aos seus devidos lugares, temos que posicionar o braço do robô novamente, afim de que ele pegue outra bola disponível novamente. Com o lugar p21, é o estado de início do movimento de baixar a garra para que fique na altura na esteira. Isso é executado com o código anotado na transição t29. Na Figura 5.5, contém uma tabela que mostra os códigos que estão anotados em cada lugar e transição.

Lugar	Código anotado	Trasição	Código anotado
p1	true	t0	#define THRESHOLD 40 task main() {
p2	}	t2	SetSensorLight(IN_3);
p3	Sensor(IN_3) < THRESHOLD	t3	OnRev(OUT_AB, 30);
p6	true	t4	Off(OUT_AB);
p7	}	t6	#define THRESHOLD 50 #define THRESHOLD2 10 task main() {
p8	Sensor(IN_3) > THRESHOLD2	t8	SetSensorLight(IN_3);
p9	i < 8	t9	RotateMotor(OUT_A, 75, 60); int i=0;
p10	Sensor(IN_3) < THRESHOLD	t12	RotateMotor(OUT_B, 75, 360); i=i+1;
p12	j < 15	t14	int j=0;
p13	j < 15	t15	RotateMotor(OUT_A, 75, -60); J=0;
p17	j < 15	t17	RotateMotor(OUT_C, 75, -360); j=j+1;
p18	j < 15	t19	RotateMotor(OUT_C, 75, 360); j=j+1;
p21	i < 8	t21	int j=0;
		t22	RotateMotor(OUT_A, 75, -60); J=0;
		t24	RotateMotor(OUT_C, 75, 360); j=j+1;
		t26	RotateMotor(OUT_C, 75, -360); j=j+1;
		t28	l=0;
		t29	RotateMotor(OUT_B, 75, -360); i=i+1 ;

Figura 5.5. Códigos anotados nos lugares e transições

5.1.2 Checagem de propriedades da Redes de Petri



Property	Situation	Property	Situation
ORD	Y	DTP	N
HOM	Y	SMC	?
NBM	N	SMD	N
PUR	Y	SMA	N
CSV	N	CPI	?
SCF	N	CTI	N
CON	Y	B	?
SC	N	SB	?
Ft0	N	REV	?
tF0	N	DSt	?
Fp0	Y	BSt	?
pF0	Y	DTr	?
MG	N	DCF	?
SM	N	L	N
FC	Y	LV	?
EFC	Y	L&S	N
ES	Y		

Figura 5.6. Resultado da checagem das propriedades da redes de Petri

Através de um *parser*, onde converte o arquivo texto, gerado pela interface gráfica, para o formato `.pnt` da ferramenta INA, deste modo, efetuamos a análise com a ferramenta INA e obtivemos os seguintes resultados (figura 5.6): a rede não é estaticamente livre de conflito, é pura (PUR), é homogênea (HOM), é ordinária (ORD), é conectada (CON), não é conservativa (CSV), não é uma máquina de estado (SM), existem lugar(es) sem pré transição(Fp0), há lugar(es) sem pós transição (pF0), é de escolha-livre (FC), é escolha livre estendida (EFC), é extensível simples (ES), a rede não é uma SMD e nem SMA, não é fortemente conectada (SC), a rede não é abrangida por semipositivos T-Invariantes (CTI), não é viva e segura (L&S), propriedade de *deadlock*(DTP) não é válida.

Com base na aplicação destas etapas, o método finalmente gera o código NXC final (ver Apêndice A.4 e A.3), pronto para ser embarcado ao projeto modelado.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Considerações Finais

Este trabalho apresentou um método que utiliza o formalismo redes de Petri para gerar códigos automáticos através de uma prototipagem.

Uma das formas de validar a solução de um problema proposto pelo usuário de software é através das especificações formais, porém, estas podem capturar erroneamente os requisitos do usuário, apresentando um solução que não corresponde a solução do problema proposto.

Uma outra maneira, seria a geração de protótipos, isto é, validar por execução, sendo possível detectar falhas, uma vez que, somente durante a interação real do usuário com o sistema, que os detalhes realmente são percebidos. Desde cedo, através da prototipagem, é possível receber valiosos *feedbacks* dos usuários, evitando que os erros sejam propagados, reduzindo os custos e o tempo de desenvolvimento. Além disso, podem melhorar a qualidade do software e dos requisitos fornecidos aos desenvolvedores.

Para os desenvolvedores os protótipos constituem uma especificação executável que facilita avaliação de diferentes modelos e ajuda a reduzir as diferenças de interpretação na construção de softwares [Alcoforado, 2007]. Neste trabalho, foi criada a

ferramenta PNTCG que permitiu uma especificação executável baseada em um modelo formal utilizando Redes de Petri.

Por meio da integração com um *Bounded Model Checker*, códigos escritos na linguagem C puderam ser anotados na ferramenta PNTCG e propriedades puderam ser verificadas formalmente para comprovar o comportamento e assegurar as propriedades definidas no código.

Por meio também da integração com a ferramenta INA, possibilitou análise de redes de Petri modelada em diferentes tipos de investigação relacionada a disparos na rede e análise de propriedades gerais. Propriedades que podem ser verificadas por meio da análise da limitação dos lugares, vivacidades da transições e alcançabilidade das marcações ou estados. Outras propriedades puderam ser verificadas como, se a rede é ordinária, homogênea, conservativa, é fortemente conectada, se é uma máquina de estado, limitada, reversível etc.

A aplicação do método no estudo de caso aqui proposto também demonstrou gerar código utilizando o conceito de busca em profundidade para percorrer o grafo da rede de Petri e diferenciar um comportamento de uma estrutura de repetição com uma estrutura de seleção. Portanto, a metodologia abordada é viável e contribui significativamente no processo de desenvolvimento de códigos na linguagem NXC e C a partir do formalismo de uma rede de Petri.

6.2 Trabalhos Futuros

. Nessa seção, explicitamos os principais itens para estender a metodologia apresentada:

- Desenvolver um *Bounded Model Checker* simples para a linguagem NXC;
- Especificar Redes de Petri Coloridas e outras redes;
- Utilizar outras linguagens com seus respectivos compiladores e possíveis Model

Checkers, por exemplo, a linguagem Java, com o Java Pathfinder¹.

¹<http://babelfish.arc.nasa.gov/trac/jpf>

Referências Bibliográficas

- Aaby, A. A.; Anthony, C. & Aaby, A. (1996). Compiler construction using flex and bison.
- Aho, A. V.; Lam, M. S.; Sethi, R. & Ullman, J. D. (2008). *Compiladores: princípios, técnicas e ferramentas*. Pearson Addison-Wesley.
- Alcoforado, M. G. (2007). Comunicação intermediada por protótipos. Master's thesis, Universidade Federal de Pernambuco.
- Ball, S. R. (1996). *Embedded Microprocessor Systems: Real World Design*. Butterworth-Heinemann, Newton, MA, USA.
- Barreto, R. d. (2005). *A Time Petri Net-based Methodology for Embedded Hard Real-Time Software Synthesis*. PhD thesis, Universidade Federal de Pernambuco, Centro de Informática.
- Brauer, W.; Reisig, W. & Rozenberg, G., editores (1987). *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986-Part I*, London, UK. Springer-Verlag.
- Budde, R.; Kuhlenkamp, K.; Kautz, K. & Zulighoven, H. (1992). *Prototyping: An Approach to Evolutionary System Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- Clarke, E.; Grumberg, O. & Peled, D. (1999). *Model Checking*. MIT Press.
- Clarke, E. & Kroening, D. (2006). Ansi-c bounded model checker user manual.
- Clarke, E. M. & Wing, J. M. (1996). Formal methods: State of the art and future directions. volume 28. ACM Computing Surveys.
- Conway, C.; Li, C.-H. & Pengelly, M. (2002). Pencil: A petri net specification language for java. Math Departament Macquire University. Australia.

- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. & Stein, C. (2001). *Introduction to Algorithms*. MIT Press.
- Deboni, F. L. & Borba, R. F. (2007). *Sistemas Embarcados em Segurança de Redes - OPENWRT*. Monografia apresentada ao Curso de Pós-graduação em Segurança de Redes de Computadores da Faculdade Salesiana de Vitória.
- Dezani, H. (2006). Geração automática de código para microcontroladores aplicada a um ambiente de co-projeto de hardware e software. Master's thesis, Faculdade de Engenharia de Ilha Solteira - UNESP/FEIS.
- Floyd, C. (1984). *A Systematic Look at Prototyping*, pp. 1--18. Springer-Verlag, Berlin.
- Francês, C. R. L., editor (2003). *Introdução às Redes de Petri*. Laboratório de Computação Aplicada - LACA, Universidade Federal do Pará - UFPA.
- Freitas, J. M. (2010). Metamodelação de processos e serviços. Master's thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Departamento de Informática. Dissertação apresentada para a obtenção do Grau de Mestre em Engenharia Informática.
- Lakos, C. & Keen, C. (1994). Loopn++: A new language for object-oriented petri net. In *European Simulation Multiconference*, pp. 369–374p.
- Lee, D. & Yannakakis, M. (1994). Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.*, 43:306--320.
- Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.
- Palomino, R. C. (1995). *Uma Abordagem para a Modelagem, Análise e Controle de Sistemas de Produção Utilizando Redes de Petri*. PhD thesis, Universidade Federal de Santa Catarina, Programa de Pós-Graduação em Engenharia de Produção.
- Pinto, L.; Rosa, R.; Pacheco, C.; Xavier, C.; Barreto, R.; Lucena, V.; Caxias, M. & Figueiredo, C. (2009). On the use of scrum for the management of practical projects in graduate courses. In *Frontiers in Education Conference, 2009. FIE '09. 39th IEEE*, pp. 1–6.
- Pressman, R. S. (2002). *Engenharia de Software*. Mc Graw Hill.

- Rangel, G. S. (2003). *ProTool: uma Ferramenta de Prototipação de Software para o Ambiente PROSOFT*. PhD thesis, Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação.
- Rangel, G. S. (2006). *Protool: uma ferramenta de prototipação de software para o ambiente prosoft*. Master's thesis, Universidade Federal do Rio Grande do Sul.
- Reisig, W. & Rozenberg, G., editores (1998). *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, London, UK. Springer-Verlag.
- Roch, S. & Starke, P. H. (1999). *INA Integrated Net Analyzer*. Humboldt-Universität zu Berlin Institut für Informatik Lehrstuhl für Automaten- und Systemtheorie. Version 2.2.
- Rocha, H.; Cordeiro, L.; Barreto, R. & Netto, J. (2010). Exploiting safety properties in bounded model checking for test cases generation of c programs. In *4th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, pp. 121--130. SBC.
- Sibertin-Blanc, C. (2001). Cooperative objects: principles, use and implementation. pp. 216--246.
- Synopsys (2003). *SystemC User's Guide*. Synopsys Inc. Version 2.0. Update for SystemC 2.0.1.
- Taurion, C. (2005). *Software Embarcado: a Nova Onda da Informática*. Brasport.
- Yakovlev, A. (2002). Is the die cast for the token game? In *ICATPN '02: Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, pp. 70--79, London, UK. Springer-Verlag.

Apêndice A

Apêndice

Nesse capítulo disponibilizaremos o código completo na linguagem NxC para o leitor deste trabalho

Algoritmo A.1. Código Redes de Petri com código anotado.

```

net ec1;
place p0{
<%
  %>
}
place p1{
<%
true
  %>
}
place p2{
<%

  %>
}
place p3{
<%
Sensor(IN_3) < THRESHOLD
  %>
}
place p4{
<%
  %>
}
place p5{
<%
  %>
}
place p6{
<%
true
  %>
}
place p7{
<%

```

```

    %>
}
place p8{
<%
Sensor(IN_3) > THRESHOLD2
%>
}
place p9{
<%
i<8
%>
}
place p10{
<%
Sensor(IN_3) < THRESHOLD
%>
}
place p11{
<%
%>
}
place p12{
<%
j<15
%>
}
place p13{
<%
j<15
%>
}
place p14{
<%
%>
}

```

```

place p15{
<%
endif
  %>
}
place p16{
<%
  %>
}
place p17{
<%
j<15
  %>
}
place p18{
<%
j<15
  %>
}
place p19{
<%

  %>
}
place p20{
<%

  %>
}
place p21{
<%
i<8
  %>
}
place p22{
<%

```

```

    %>
}
place p23{
<%
endif
    %>
}

transition t0{
in: p0;
out: p1;
<%
#define THRESHOLD 40
task main()
{
    %>
}

transition t1{
in: p1;
out: p2;
<%
}
    %>
}
transition t2{
in: p1;
out: p3;
<%
SetSensorLight(IN_3);
    %>
}
transition t3{
in: p3;
out: P4;

```

```

<%
    OnRev(OUT_AB, 30);
    %>
}
transition t4{
in: p3;
out: p4,P5;
<%
Off(OUT_AB);
    %>
}
transition t5{
in: p4;
out: p1;
<%

    %>
}
transition t6{
in: p5;
out: p6;
<%
#define THRESHOLD 50
#define THRESHOLD2 10
task main()
{
    %>
}
transition t7{
in: p6;
out: p7;
<%
}
    %>
}
transition t8{

```

```

in: p6;
out: p8;
<%
SetSensorLight(IN_3);
%>
}
transition t9{
in: p8;
out: p9;
<%
RotateMotor(OUT_A, 75, 60);
int i=0;
%>
}
transition t10{
in: p8;
out: p23;
<%

%>
}
transition t11{
in: p9;
out: p10;
<%

%>
}
transition t12{
in: p9;
out: p11;
<%
RotateMotor(OUT_B, 75, 360);
i=i+1 ;
%>
}

```

```

transition t13{
in: p11;
out: p9;
<%

    %>
}
transition t14{
in: p10;
out: p12;
<%
    int j=0;
    %>
}
transition t15{
in: p12;
out: p13;
<%
RotateMotor(OUT_A, 75, -60);
j=0;
    %>
}
transition t16{
in: p13;
out: p15;
<%

    %>
}
transition t17{
in: p13;
out: p14;
<%
RotateMotor(OUT_C, 75, -360);
j=j+1 ;
    %>

```

```

}
transition t18{
in: p14;
out: p13;
<%

    %>
}
transition t19{
in: p12;
out: p16;
<%
RotateMotor(OUT_C, 75, 360);
j=j+1 ;
    %>
}
transition t20{
in: p16;
out: p12;
<%

    %>
}
transition t21{
in: p10;
out: P17;
<%
int j=0;
    %>
}
transition t22{
in: p17;
out: P18;
<%
RotateMotor(OUT_A, 75, -60);
j=0;

```

```

%>
}
transition t23{
in: p18;
out: P15;
<%

%>
}
transition t24{
in: p18;
out: P20;
<%
RotateMotor(OUT_C, 75, 360);
j=j+1 ;
%>
}
transition t25{
in: p20;
out: P18;
<%

%>
}
transition t26{
in: p17;
out: P19;
<%
RotateMotor(OUT_C, 75, -360);
j=j+1 ;

%>
}

transition t27{
in: p19;

```

```

out: P17;
<%

    %>
}
transition t28{
in: p15;
out: P21;
<%
i=0;
    %>
}
transition t29{
in: p21;
out: P22;
<%
RotateMotor(OUT_B, 75, -360);
i=i+1 ;
    %>
}

transition t30{
in: p22;
out: P21;
<%

    %>
}
transition t31{
in: p21;
out: P23;
<%

    %>
}
transition t32{

```

```
in: p23;
```

```
out: P6;
```

```
<%
```

```
%>
```

```
}
```

Algoritmo A.2. Código da Descrição da Redes de Petri de Rede lugar/transição.

P M PRE,POST NETZ 0:t

0 1 , 0

1 0 5 0 , 2 1

2 0 1

3 0 2 , 3 4

4 0 3 4 , 5

5 0 4 , 6

6 0 32 6 , 7 8

7 0 7

8 0 8 , 9 10

9 0 9 13 , 11 12

10 0 11 , 14 21

11 0 12 , 13

12 0 14 20 , 15 19

13 0 18 15, 17 16

14 0 17 , 18

15 0 23 16 , 28

16 0 19 , 20

17 0 21 27 , 22 26

18 0 22 25 , 23 24

19 0 26 , 27

20 0 24 , 25

21 0 30 28 , 29 31

22 0 29 , 30

23 0 10 31 , 32

@

place	nr.	name	capacity	time
-------	-----	------	----------	------

0:	p0	65535	0
----	----	-------	---

1:	p1	65535	0
----	----	-------	---

2:	p2	65535	0
----	----	-------	---

3:	p3	65535	0
----	----	-------	---

4:	p4	65535	0
----	----	-------	---

5:	p5	65535	0
----	----	-------	---

6:	p6	65535	0
----	----	-------	---

7:	p7	65535	0
----	----	-------	---

8: p8	65535	0
9: p9	65535	0
10: p10	65535	0
11: p11	65535	0
12: p12	65535	0
13: p13	65535	0
14: p14	65535	0
15: p15	65535	0
16: p16	65535	0
17: p17	65535	0
18: p18	65535	0
19: p19	65535	0
20: p20	65535	0
21: p21	65535	0
22: p21	65535	0
23: p22	65535	0

@

trans	nr.	name	priority	time
0:	t0		0	0
1:	t1		0	0
2:	t2		0	0
3:	t3		0	0
4:	t4		0	0
5:	t5		0	0
6:	t6		0	0
7:	t7		0	0
8:	t8		0	0
9:	t9		0	0
10:	t10		0	0
11:	t11		0	0
12:	t12		0	0
13:	t13		0	0
14:	t14		0	0
15:	t15		0	0
16:	t16		0	0
17:	t17		0	0

Algoritmo A.3. Código em NxC dos movimentos da Esteira.

```
#define THRESHOLD 40

task main()
{

    while (true)
    {
        SetSensorLight(IN_3);
        if (Sensor(IN_3) < THRESHOLD) {
            OnRev(OUT_AB, 30);
            //    Wait(5);

        }
        else { Off(OUT_AB);}
    }
}
```

Algoritmo A.4. Código em NxC dos movimentos do Braço Robô.

```
#define THRESHOLD 50
#define THRESHOLD2 10
task main()
{

    while (true)
    {

        SetSensorLight(IN_3);

        if (Sensor(IN_3) > THRESHOLD2) //verifica se ha objeto na sua frente
        {

            RotateMotor(OUT_A, 75, 60);
            int i=0;
            while (i<8)
            {
                RotateMotor(OUT_B, 75, 360); //move para cima
                i=i+1 ;
            }

            if (Sensor(IN_3) < THRESHOLD)
            {

                int j=0;
                while (j<15)
                {
                    RotateMotor(OUT_C, 75, 360);
//move para direita
                    j=j+1 ;
                }

                RotateMotor(OUT_A, 75, -60);

                j=0;
                while (j<15)
```

```

        {
            RotateMotor(OUT_C, 75, -360);
//move para esquerda
            j=j+1 ;
        }
    }

    else
    {

        int j=0;
        while (j<15)
        {
            RotateMotor(OUT_C, 75, -360);
//move para esquerda
            j=j+1 ;
        }

        RotateMotor(OUT_A, 75, -60);

        j=0;
        while (j<15)

        {
            RotateMotor(OUT_C, 75, 360);
//move para direita
            j=j+1 ;
        }
    }

    i=0;
    while (i<8)
    {
        RotateMotor(OUT_B, 75, -360); //move para baixo
        i=i+1 ;
    } //while

```

```
    } //if
    else
    { //nada a fazer
    }

    } // first while

} //task
```

```

150
151 public static void imprime2(Vertex vertice, HashMap<mxCell, String> codep, HashMap<mxCell, String> codet, Vertex verticeFinal){
152     mxCell cell = null;
153     if ((vertice != verticeFinal) && (vertice!=null)&&(!vertice.isbVisitado())) {
154         vertice.setbVisitado(true);
155         cell = (mxCell) vertice.getValor();//*-
156         if ((codep.get(vertice.getValor()) != null) && (!codep.get(vertice.getValor()).equals(""))
157             && (!codep.get(vertice.getValor()).equals("endif"))) {
158             Vertex verticeDoWhile = isWhile(vertice, vertice);
159             if (verticeDoWhile != null){
160                 System.out.printf("while (" +codep.get(vertice.getValor())+" ) {"");
161                 imprime2(verticeDoWhile,codep,codet,vertice);
162                 System.out.printf("}");
163             }
164             else {
165                 Vertex verticeEndIf = null;
166                 if (vertice.getaVertice().size() > 0) {
167                     verticeEndIf = getEndIf(vertice,codep);
168                     System.out.printf("if (" +codep.get(vertice.getValor())+" ) {\n");
169                     imprime2(vertice.getaVertice().get(0),codep,codet,verticeEndIf);
170                     System.out.printf("\n}\n");
171                     if (vertice.getaVertice().size() > 1) {
172                         System.out.println("else {\n");
173                         imprime2(vertice.getaVertice().get(1),codep,codet,verticeEndIf);
174                         System.out.printf("\n}");
175                     }
176                 }
177                 vertice = verticeEndIf;
178             }
179         }
180         else {
181             if (codet.get(vertice.getValor()) != null) {
182                 System.out.println(codet.get(vertice.getValor()));
183             }
184         }
185         if (vertice != null) {
186             for(int cont = 0; cont < vertice.getaVertice().size(); cont++) {
187                 imprime2(vertice.getaVertice().get(cont),codep,codet,verticeFinal);
188             }
189         }
190     }
191 }
192 public static void imprime(Vertex vertice,HashMap<mxCell, String> codep, HashMap<mxCell, String> codet) {
193     imprime2(vertice,codep,codet, null);
194     Vertex.resetVisitados(vertice);
195 }

```

Figura A.1. Parte do Algoritmo A

```

75 public static boolean isWhile2(Vertex vertice, Vertex verticeOrigem, ArrayList<Vertex> aVerticeVisitados) {
76     boolean resultado = false;
77     if (!isVisitado(vertice, aVerticeVisitados) && !vertice.isbVisitado()) {
78         aVerticeVisitados.add(vertice);
79         if (vertice == verticeOrigem) {
80             resultado = true;
81         } else {
82             for(int cont = 0; cont < vertice.getaVertice().size(); cont++) {
83                 resultado = isWhile2(vertice.getaVertice().get(cont), verticeOrigem, aVerticeVisitados);
84                 if (resultado) {
85                     break;
86                 }
87             }
88         }
89     } else {
90         if (vertice == verticeOrigem) {
91             resultado = true;
92         }
93     }
94     return resultado;
95 }
96
97 public static Vertex isWhile(Vertex vertice, Vertex verticeOrigem) {
98     Vertex resultado = null;
99     ArrayList<Vertex> aVerticeVisitados = new ArrayList<Vertex>();
100
101     aVerticeVisitados.add(vertice);
102     for(int cont = 0; cont < vertice.getaVertice().size(); cont++) {
103         if (isWhile2(vertice.getaVertice().get(cont), verticeOrigem, aVerticeVisitados)) {
104             resultado = vertice.getaVertice().get(cont);
105             break;
106         }
107     }
108     return resultado;
109 }

```

Figura A.2. Algoritmo B

```

118 public static void setVisitados(Vertex vertice, ArrayList<Vertex> aVerticeVisitados) {
119     aVerticeVisitados.add(vertice);
120     for(int cont = 0; cont < vertice.getaVertice().size(); cont++) {
121         aVerticeVisitados.add(vertice.getaVertice().get(cont));
122     }
123 }
124
125 public static Vertex getEndIf2(Vertex vertice, ArrayList<Vertex> aVerticeVisitados, HashMap<mxCell, String> codep) {
126     Vertex resultado = null;
127
128     if (!isVisitado(vertice, aVerticeVisitados) && !vertice.isbVisitado() && (codep.get(vertice.getValor()) != null)
129         && (codep.get(vertice.getValor()).equals("endif"))) {
130         resultado = vertice;
131     } else {
132         for(int cont = 0; cont < vertice.getaVertice().size(); cont++) {
133             resultado = getEndIf2(vertice.getaVertice().get(cont), aVerticeVisitados, codep);
134             if (resultado != null) {
135                 break;
136             }
137         }
138     }
139     return resultado;
140 }
141
142 public static Vertex getEndIf(Vertex vertice, HashMap<mxCell, String> codep) {
143     Vertex resultado = null;
144     ArrayList<Vertex> aVerticeVisitados = new ArrayList<Vertex>();
145
146     setVisitados(vertice.getaVertice().get(0), aVerticeVisitados);
147     resultado = getEndIf2(vertice.getaVertice().get(0), aVerticeVisitados, codep);
148     return resultado;
149 }
150
151
152
153
154

```

Figura A.3. Algoritmo C