



ALEXANDRE LUÍS DE SOUSA GABRIEL

Licenciatura em Ciências de Engenharia Eletrotécnica e de
Computadores

EXECUÇÃO DE MODELOS DE REDES DE PETRI IOPT USANDO O CONCEITO DE CONTENDA E COMPUTAÇÃO PARALELA

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
November, 2021

EXECUÇÃO DE MODELOS DE REDES DE PETRI IOPT USANDO O CONCEITO DE CONTENDA E COMPUTAÇÃO PARALELA

ALEXANDRE LUÍS DE SOUSA GABRIEL

Licenciatura em Ciências de Engenharia Eletrotécnica e de Computadores

Orientador: Luís Filipe dos Santos Gomes

Professor Associado com Agregação, NOVA School of Science and Technology / NOVA University Lisbon

Júri

Presidente: Luís Filipe Figueira Brito Palma

*Professor Auxiliar, NOVA School of Science and Technology / NOVA University
Lisbon*

Arguente: Rogério Alexandre Botelho Campos Rebelo

UNINOVA

Orientador: Luís Filipe dos Santos Gomes

*Professor Associado com Agregação, NOVA School of Science and Technology /
NOVA University Lisbon*

Execução de modelos de redes de Petri IOPT usando o conceito de contenda e computação paralela

Copyright © Alexandre Luís de Sousa Gabriel, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

AGRADECIMENTOS

Quero agradecer ao professor Luís Gomes, pelo apoio e orientação ao longo da dissertação e do curso.

Um agradecimento à Professora Anikó Costa, por sempre se ter disponibilizado a me entregar material, e pelo apoio durante o curso.

Quero agradecer ao professor Fernando Pereira pelo apoio e disponibilidade, pela ajuda em relação ao ambiente IOPT-Tools.

Um agradecimento também a todos os professores e à FCT/UNL que me permitiram realizar este percurso.

Aprendi no primeiro ano que “ninguém faz o curso sozinho”, e desta forma quero agradecer a todos os meus amigos que me acompanharam, pelos bons momentos e pelas grandes amizades.

Finalmente, mas não menos importante, quero agradecer aos meus pais, avós e tios por sempre me acompanharem, apoiarem e me permitirem que esta etapa se realizasse.

RESUMO

O ambiente de desenvolvimento IOPT-Tools disponível publicamente em <http://gres.uninova.pt/IOPT-Tools/>, permite a edição, simulação e geração de código para execução de modelos expressos através de Redes de Petri (RdP) da classe IOPT - Input-Output Place-Transition nets, fornecendo todas as ferramentas de suporte para desenvolvimento e teste de controladores digitais. O servidor IOPT-Tools fornece também um gerador do espaço de estados associado aos modelos RdP, baseado no código de execução do modelo, também produzido pelo próprio servidor associado ao modelo em análise. Mesmo para modelos modestos, devido à dimensão do espaço de estados gerado, o gerador pode ocupar os recursos do servidor por “bastante” tempo.

Assim, esta dissertação pretende contribuir para dois objetivos, nomeadamente, explorando a identificação de contendas na geração do código de execução do modelo, melhorar o código C gerado pelo gerador automático do servidor IOPT-Tools em termos do seu tempo de execução, e posteriormente adaptar o código gerado, utilizando ferramentas de computação paralela, para utilizar eficazmente os vários *cores* do servidor IOPT-Tools na geração do espaço de estados associado.

Desta forma, tendo presente resultados obtidos em trabalhos desenvolvidos anteriormente, é apresentado um algoritmo que explora o conceito de contenda e a organização das transições aquando da geração do código C, de modo a melhorar o tempo de execução e reduzir o uso de recursos. Nas estratégias consideradas foram tidos em conta modelos RdP seguras e limitadas. O impacto da utilização do código gerado considerando o novo algoritmo também foi analisado recorrendo a computação paralela em processadores multi-core.

Palavras-chave: Redes de Petri, Classe de rede de Petri IOPT, IOPT-Tools, contenda, C Geração automática de código, computação paralela, multi-core

ABSTRACT

The environment IOPT-Tools publicly available at <http://gres.uninova.pt/IOPT-Tools/>, allows edition, simulation and code generation for the execution of models expressed through RdP of the IOPT - Input-Output Place-Transition nets class, providing all the support tools for developing and testing digital controllers. The server also provides a state space generator associated with RdP models, which is based on the model execution code, also produced by the server itself associated to the model under analysis. Even for modest models, due to the size of the generated state space, the generator can occupy server resources for "quite"long time.

Thus, this dissertation aims to contribute with two objectives, namely, by exploring the identification of conflict sets in the generation of the model execution code, to improve the C code generated by the automatic generator of the IOPT-Tools server in terms of its execution time, and subsequently to adapt the generated code, using parallel computing tools, to effectively use the various cores of the IOPT-Tools server in the generation of the associated state space.

This way, having in mind results obtained in previous works, an algorithm that explores the concept of conflict sets and the organization of transitions when generating C code, is presented, in order to improve execution time and reduce the use of resources. Safe and limited Petri net models were taken into account in the strategies considered. The impact of using the generated code considering the new algorithm was also analyzed using parallel computing on multi-core processors.

Keywords: Petri nets, IOPT Petri nets class, IOPT-Tools, conflict set, C, Automatic code generation, parallel computing, multi-core

ÍNDICE

Índice de Figuras	ix
Índice de Tabelas	xii
Índice de Listagens	xiii
Siglas	xiv
1 Introdução	1
1.1 Enquadramento do trabalho e motivação	1
1.2 Problema	2
1.3 Objetivos	2
1.4 Estrutura da dissertação	3
2 Conceitos e tecnologias de Suporte	5
2.1 Redes de Petri	5
2.1.1 História	5
2.1.2 Definição	5
2.1.3 Grafo de redes de Petri	6
2.1.4 Marcação em redes de Petri	7
2.2 Classes de Redes de Petri	7
2.2.1 Propriedades das redes de Petri	8
2.2.2 Habilitação e disparo de transições	8
2.2.3 Conflitos em redes de Petri	9
2.2.4 Contendas em redes de Petri	10
2.2.5 Grafo do Espaço de estados de redes de Petri	11
2.2.6 Classe de redes de Petri <i>Input-Output Place-Transition</i> (IOPT) . .	11
2.3 <i>Petri Net Markup Language</i> (PNML)	15
2.3.1 PNML <i>core model</i>	15
2.3.2 Petri Net Type Definitions	15
2.3.3 Mapeamento para XML	17
2.4 <i>eXtensible Stylesheet Language Transformations</i> (XSLT)	18
2.5 Processamento multi-core	20

2.5.1	Introdução	20
2.5.2	<i>Multi-threading</i> & multi-cores	21
2.6	Programação paralela com OpenMP	22
2.7	Ambientes de desenvolvimento de Software	23
2.7.1	Arduino <i>Integrated Development Environment</i>	23
2.7.2	Vitis	24
2.7.3	Visual Studio	24
2.8	Plataformas de hardware para teste	24
2.8.1	Arduino Mega 2560	24
2.8.2	Arduino DUE	24
2.8.3	ZedBoard	24
2.8.4	P8B-E/4L	25
2.8.5	Asus ROG GL553VE-77AT5PB2	25
3	IOPT-Tools e Arquitetura	26
3.1	IOPT-Tools	26
3.1.1	Introdução	26
3.1.2	Editor	28
3.1.3	Simulador	33
3.1.4	Gerador do espaço de estados	34
3.1.5	<i>quersys</i>	35
3.1.6	Gerador de código VHDL	36
3.1.7	Gerador de código C	36
3.1.8	Execução do código C gerado	36
3.1.9	Rede de Petri IOPT usando PNML	38
3.2	Ficheiros XSLT	40
3.2.1	Ficheiro <i>iopt2c_inc</i>	40
3.2.2	Ficheiro <i>iopt2c_io</i>	40
3.2.3	<i>iopt2c_func</i>	41
3.2.4	<i>iopt2c_exec_step</i>	42
3.2.5	<i>iopt2c_main</i>	42
3.3	Algoritmo PHP de geração do código C	43
4	Algoritmos Desenvolvidos	45
4.1	Introdução	45
4.2	Algoritmo de adaptação do PNML para contendas	45
4.2.1	Função <i>organize_contendas</i>	46
4.2.2	Função <i>processTransition</i>	48
4.2.3	Função <i>transitionVectorAttribution</i>	52
4.2.4	Função <i>placeContendaAssociation</i>	53
4.2.5	Função <i>compareTransitionPriority</i>	55

4.3	Algoritmo do código C gerado	56
4.3.1	Ficheiro <i>net_types.h</i>	56
4.3.2	Ficheiro <i>net_functions.c</i>	58
4.3.3	Ficheiro <i>net_exec_step.c</i>	60
4.4	Conversor de modelos PIPE para PNML	64
5	Testes de validação e análise de resultados	67
5.1	Introdução	67
5.2	Modelos	67
5.2.1	Modelo <i>test5</i>	67
5.2.2	Modelo <i>TIE10_automation</i>	72
5.2.3	Modelo <i>girault8</i>	73
5.3	Testes de validação e análise de resultados	74
5.3.1	Testes e análise do algoritmo de contendas	75
5.3.2	Testes e análise do algoritmo de computação paralela	80
5.3.3	Testes e análise do algoritmo final	83
6	Conclusões	86
6.1	Balanço geral	86
6.2	Trabalho futuro	88
	Bibliografia	89

ÍNDICE DE FIGURAS

2.1	Elementos de um Grafo de Redes de Petri.	6
2.2	Exemplo de um Grafo de Redes de Petri.	7
2.3	Exemplo de habilitação e disparo de uma Rede de Petri.	9
2.4	Exemplo de conflictos em Redes de Petri.	10
2.5	Exemplo de Contendas em Redes de Petri.	11
2.6	Exemplo de um Grafo de Espaço de Estados de uma Rede de Petri.	12
2.7	PNML <i>Core Model</i>	16
2.8	PNTD para <i>Place/Transition-Nets</i>	17
2.9	Resultado da transformação por XSLT	19
2.10	Exemplo da configuração de um processador.	20
2.11	Modelos de CPUs de memória partilhada e distribuída.	21
2.12	Comparação de arquiteturas de processadores <i>Single-Core</i> , <i>Multi-Core</i> , e <i>Multi-Threading</i>	22
2.13	Abstração da execução paralela utilizando OpenMP.	23
3.1	IOPT-Tools página principal.	27
3.2	IOPT-Tools página do editor.	28
3.3	Propriedades dos elementos do editor IOPT-Tools.	30
3.4	Propriedades dos eventos do editor IOPT-Tools.	31
3.5	Propriedades dos sinais do editor IOPT-Tools.	31
3.6	Ferramenta <i>ClipBoard</i> do editor IOPT-Tools.	32
3.7	Ferramenta de edição de expressões da plataforma IOPT-Tools.	33
3.8	Simulador de modelos da plataforma IOPT-Tools.	34
3.9	Exemplo de um grafo de espaço de estados gerado pela IOPT-Tools.	34
3.10	Página de edição e selecção de <i>queries</i> da plataforma IOPT-Tools.	35
3.11	Plataforma de execução do algoritmo em C gerado.	37
3.12	Fluxograma do algoritmo de PHP para gerar os ficheiros C.	43
4.1	Exemplo de rede de Petri.	46
4.2	Exemplo da transição de id 2 após o primeiro foreach	47
4.3	Exemplo da transição de id 4 após ser processada.	51
4.4	Exemplo da parte das transições do PNML da rede de Petri da figura 4.1.	53
4.5	Elemento <i>contendas</i> e respetivos filhos associado à rede de Petri da figura 4.1.	53

4.6	Elemento respetivo ao lugar 12 da rede de Petri da figura 4.1.	55
4.7	Exemplos da divisão das principais estruturas de dados do algoritmo gerado.	57
4.8	Função <i>createInitial_avail_NAME_ContendasMarking</i> associada à rede de Petri da figura 4.1.	58
4.9	Função <i>transfer_NAME_ContendasMarking</i> associada à rede de Petri da figura 4.1.	58
4.10	Função <i>createInitial_add_NAME_ContendasMarking</i> associada à rede de Petri da figura 4.1.	59
4.11	Função <i>add_NAME_ContendasMarking</i> associada à rede de Petri da figura 4.1.	59
4.12	Função <i>transfer_NAME_firingMarking</i> associada à rede de Petri da figura 4.1.	59
4.13	Página index da ferramenta IOPT-Tools com o formulário para ficheiros PIPE.	66
5.1	Rede de Petri do modelo <i>test5_2C</i>	68
5.2	Rede de Petri do modelo <i>test5_2C_new</i>	69
5.3	Rede de Petri do modelo <i>test5_6C</i>	70
5.4	Rede de Petri do modelo <i>test5_6C_new</i>	70
5.5	Rede de Petri do modelo <i>test5_10C</i>	71
5.6	Rede de Petri do modelo <i>test5_10C_new</i>	71
5.7	Rede de Petri do modelo <i>TIE10_automation</i>	72
5.8	Rede de Petri do modelo <i>girault8</i> na plataforma Hippo.	73
5.9	Rede de Petri do modelo <i>girault8</i> na plataforma IOPT-Tools.	74
5.10	Gráfico de Transições ignoradas por modelo até ao final de uma execução completa.	76
5.11	Gráfico de Transições totais por modelo.	76
5.12	Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendras na plataforma <i>Asus ROG Laptop</i>	77
5.13	Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendras na plataforma <i>P8B-E/4L</i>	77
5.14	Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendras na plataforma <i>Zedboard</i>	78
5.15	Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendras na plataforma <i>Arduino Mega</i>	79
5.16	Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendras na plataforma <i>Arduino DUE</i>	80
5.17	Gráfico de resultados obtidos da execução dos modelos com o algoritmo de computação paralela na plataforma <i>Asus ROG Laptop</i>	81
5.18	Gráfico de resultados obtidos da execução dos modelos com o algoritmo de computação paralela na plataforma <i>P8B-E/4L</i>	81
5.19	Gráfico de resultados obtidos do processamento de transições dos modelos com o algoritmo de computação paralela na plataforma <i>Asus ROG Laptop</i>	82

5.20	Gráfico de resultados obtidos do processamento de transições dos modelos com o algoritmo de computação paralela na plataforma <i>P8B-E/4L</i>	83
5.21	Gráfico de resultados obtidos da execução dos modelos com o algoritmo final na plataforma <i>Asus ROG Laptop</i>	84
5.22	Gráfico de resultados obtidos da execução dos modelos com o algoritmo final na plataforma <i>P8B-E/4L</i>	84
5.23	Gráfico de resultados obtidos do processamento de transições dos modelos com o algoritmo final na plataforma <i>Asus ROG Laptop</i>	85
5.24	Gráfico de resultados obtidos do processamento de transições dos modelos com o algoritmo final na plataforma <i>P8B-E/4L</i>	85

ÍNDICE DE TABELAS

2.1	Elementos PNML do relativos ao <i>PNMLCore Model</i>	17
5.1	Tabela de plataformas utilizadas para teste dos algoritmos e respectivas características.	75

ÍNDICE DE LISTAGENS

2.1	Ficheiro XML.	18
2.2	Ficheiro XSLT para efetuar a transformação.	18
3.1	Pseudo-Código do passo de execução.	38
3.2	Exemplo da implementação de um disparo de uma transição.	42
3.3	Exemplo da implementação da criação do ficheiro <code>net_types.h</code> em PHP.	43
4.1	Pseudo código da função <i>organize_contenda</i>	47
4.2	Pseudo código da função <i>processTransition</i>	48
4.3	Pseudo código da função <i>transitionVectorAtribution</i>	52
4.4	Pseudo código da função <i>placeContendaAssociation</i>	54
4.5	Pseudo código da função <i>compareTransitionPriority</i>	55
4.6	Pseudo código do algoritmo que aproveita o conceito de conflito e contenda.	60
4.7	Exemplo das modificações feitas à verificação e disparo da transição para o algoritmo de conflitos.	61
4.8	Pseudo código do algoritmo integrado com a API <code>openMP</code>	62
4.9	Exemplo das modificações feitas à verificação e disparo da transição para o algoritmo de conflitos.	62
4.10	Pseudo código do algoritmo final C gerado.	63
4.11	Exemplo das modificações feitas à verificação e disparo da transição para o algoritmo final.	64
4.12	Pseudo-código da função <i>pipe2pnml()</i>	65

SIGLAS

CPU	Central Process Unit
CSS	Cascading Style Sheets
DDR3	Double data rate type 3
GALS	Globally asynchronous locally synchronous
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IOPT	Input Output place transition
PNML	Petri Net Markup Language
PNTD	Petri Net Type Definition
RdP	Redes de Petri
SVG	Scalable Vector Graphics
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations

INTRODUÇÃO

Neste capítulo é feito o enquadramento desta dissertação em relação aos trabalhos relacionados já desenvolvidos, descrevendo assim a motivação para o tema e os objectivos pretendidos. É apresentado também um breve resumo da estrutura da dissertação.

1.1 Enquadramento do trabalho e motivação

O formalismo de Redes de Petri, proposto inicialmente por Carl Adam Petri em 1962 na sua tese de doutoramento (Petri, 1962), é um formalismo de modelação bastante útil no desenvolvimento de controladores (Murata, 1989; Peterson, 1977).

Desde então o estudo das Redes de Petri tornou-se relevante, devido às possibilidades de análise de propriedades que estas oferecem e à facilidade de modelação de sistemas utilizando as mesmas (Murata, 1989). Os trabalhos de investigação em Redes de Petri permitiram construir um corpo de teoria e de ferramentas computacionais, facilitando a simulação e permitindo a criação de modelos de controladores baseados em Redes de Petri e geração de código pronto a ser executado. Como consequência, resultaram soluções mais eficientes para todo o processo de desenvolvimento de controladores e desenvolvimento de protótipos dos mesmos.

Várias ferramentas surgiram como apoio à modelação de controladores com base em Redes de Petri, como é o exemplo do ambiente IOPT-Tools desenvolvido ao longo dos últimos anos por investigadores, professores e alunos da Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa (L. Gomes, Rebelo, Barros, Costa & Pais, 2010), disponível publicamente em <http://gres.uninova.pt/IOPT-Tools/>.

O ambiente de desenvolvimento IOPT-Tools integra ferramentas computacionais de suporte para as várias fases de desenvolvimento de controladores e sistemas digitais, utilizando modelos de Redes de Petri da classe IOPT - Input-Output Place-Transition nets

(L. Gomes, Barros, Costa & Nunes, 2007). Uma das ferramentas com elevado impacto nas tarefas de desenvolvimento é a ferramenta responsável pela geração automática de código C associado ao modelo especificado (Campos-Rebelo, Pereira, Moutinho & Gomes, 2011; L. Gomes et al., 2010), e um gerador do espaço de estados e do grafo do mesmo, para suporte à verificação de propriedades (Pereira, Moutinho, Gomes & Campos-Rebelo, 2011), sendo que a execução do mesmo se baseia no código gerado automaticamente referido anteriormente.

Sendo assim o trabalho descrito neste documento tem como motivação o desenvolvimento e melhoramento de funções do ambiente IOPT-Tools, que será discutido em detalhe mais à frente.

1.2 Problema

Com a evolução das aplicações das redes de Petri, também a sua complexidade aumentou e a necessidade de novas ferramentas, técnicas e tecnologias de teste e verificação foram necessárias para o "debug" e verificação do funcionamento dos modelos. Como por exemplo, o grafo do espaço de estados, que permite a verificação de propriedades de um modelo expresso através de uma rede de Petri, nomeadamente a deteção de conflitos nesses modelos.

Como referido anteriormente o servidor IOPT-Tools disponibiliza um gerador do grafo do espaço de estados, podendo assim ser verificadas propriedades e conflitos, de modo a que possíveis problemas/propriedades não desejadas possam ser resolvidos a-priori no modelo.

Importa referir que, mesmo para modelos pouco complexos (poucos lugares e ou transições), o grafo do espaço de estados associado pode conter muitos nós e estados, o que a nível de computação é sinónimo de um tempo demorado para o seu cálculo e uma ocupação elevada dos recursos do servidor. Desta forma, para melhorar o desempenho do gerador de espaço de estados, será necessário melhorar o código C utilizado pelo gerador do mesmo.

1.3 Objetivos

Com esta dissertação pretende-se contribuir para dois objetivos finais que pela natureza do ambiente IOPT-Tools estão interligados:

- Primeiramente pretende-se adaptar a estrutura dos ficheiros PNML para que estes integrem explicitamente as informações sobre os conflitos e os conjuntos de conteúdos existentes. Permitindo assim uma avaliação à priori do disparo de transições, que por escassez de marcas e devido a conflitos, não iriam disparar. E utilizando esta informação, modificar o código C gerado para que aquando da execução do

modelo, ignore a verificação de transições que não irão ocorrer, permitindo assim tempos de execução menores e uma melhor eficiência do próprio algoritmo.

- Em segundo lugar pretende-se adaptar o código C gerado para estar apto para ser utilizado em hardware multi-core, ou seja, processadores multi-core ou GPUs. Pretende-se utilizar a API OpenMP para ajudar a resolver este desafio, resultando, desta forma, em um código C gerado mais eficiente e de menor latência, ao custo da utilização de um maior número de recursos.

Espera-se que estes dois objetivos tenham como resultado um código C gerado (referente ao modelo em rede de Petri IOPT a ser desenvolvido) muito mais eficiente e rápido, que por consequência, devido ao gerador do espaço de estados estar baseado neste código C gerado, também o processo de geração do grafo será mais rápido e eficiente.

1.4 Estrutura da dissertação

A estrutura deste documento está dividida em seis capítulos, nomeadamente:

- **Capítulo 1 - Introdução :** Neste capítulo são apresentados o enquadramento, a motivação assim como o problema e os objetivos da dissertação. Finalmente é apresentada a estrutura do documento.
- **Capítulo 2 - Conceitos e tecnologias de suporte :** Neste capítulo são apresentadas noções básicas das redes de Petri assim como algumas das suas classes e propriedades, focando nos principais conceitos explorados na dissertação como conflitos e contendidas. É também enfatizada a classe de redes de Petri IOPT utilizada pelo ambiente IOPT-Tools.

Ainda nesta capítulo são apresentadas as ferramentas utilizadas na geração automática de código , assim como noções básicas de processamento paralelo com foco na utilização de OpenMP.

Finalmente são apresentados os ambientes e as plataformas utilizadas nesta dissertação.

- **Capítulo 3 - IOPT-Tools e Arquitetura :** Neste capítulo é apresentada em detalhe a plataforma IOPT-Tools, focando principalmente no gerador de código C automático. Ainda quanto ao gerador de código C automático são apresentadas as estruturas dos ficheiros utilizados para a execução do mesmo assim como a arquitetura do algoritmo.
- **Capítulo 4 - Algoritmos Desenvolvidos :** Neste capítulo são apresentadas todas as contribuições desenvolvidas nesta dissertação. Os conteúdos são divididos em três partes.

A primeira sendo as modificações feitas ao nível de PHP no servidor em si de forma a modificar o PNML para guardar as informações de contendas e conflitos.

A segunda parte sendo o novo algoritmo C que é gerado e as modificações feitas ao nível dos ficheiros XSLT.

E a terceira sendo o conversor de modelos PIPE para modelos PNML de forma a utilizar modelos da plataforma Hippo (Hippo, s.d.).

- **Capítulo 5 - Testes de validação e análise de resultados :** Neste capítulo são apresentados os modelos utilizados assim como descrições e justificações da utilização dos mesmos.

São apresentados ainda neste capítulo todos os resultados obtidos assim como a discussão dos mesmos.

- **Capítulo 6 - Conclusão :** Neste capítulo são apresentadas as conclusões finais assim como sugestões de trabalho futuro no seguimento desta dissertação.

CONCEITOS E TECNOLOGIAS DE SUPORTE

Neste capítulo iremos abordar alguns conceitos e tecnologias de suporte necessárias para a compreensão do trabalho desenvolvido, com principal foco nas Redes de Petri e o ambiente de desenvolvimento IOPT-Tools.

Desta forma será fornecido um contexto inicial de informação necessário para a interpretação do que será referido nos capítulos seguintes.

2.1 Redes de Petri

2.1.1 História

As Redes de Petri foram inicialmente propostas por Carl Adam Petri, daí o seu nome, na sua tese de doutoramento (Petri, 1962). Este trabalho chamou a atenção de Anatol Holt que liderava o grupo de investigação em *Applied Data Research* e que graças à investigação desenvolvida, foram definidas as primeiras notações e teorias das Redes de Petri (Peterson, 1977).

Desde então muito trabalho de investigação foi feito em relação às Redes de Petri e às suas aplicações (Hujsa & Devillers, 2018; Jensen, 2013; Murata, 1989; Reisig, 2013).

2.1.2 Definição

As Redes de Petri são um modelo abstrato e formal de fluxo de informação (Petri, 1962), representado como um grafo bipartido, com arcos dirigidos e pesos (Peterson, 1977). Este grafo é constituído por dois nós principais, os estados ou lugares (representados por círculos) e as transições ou eventos (representados por barras ou retângulos). Estes dois tipos de elementos são ligados por arcos dirigidos com sentido unidirecional (de lugar para transição ou de transição para um lugar) (Murata, 1989; Reisig, 2012).

Definição 1 (Definição Formal de uma Rede de Petri (Murata, 1989)). Uma Rede de Petri é definida através do tuplo $PN = (P, T, F, W, M_0)$ onde:

- $P = \{p_1, p_2, p_3, \dots, p_m\}$ é o conjunto de m lugares;
- $T = \{t_1, t_2, t_3, \dots, t_n\}$ é o conjunto de n transições;
- $F \subseteq (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ conjunto de arcos que representam a relação de fluxo;
- $W : F \rightarrow \{1, 2, 3, \dots\}$ é a função de peso dos arcos;
- $M_0 : P \rightarrow \{0, 1, 2, 3, \dots\}$ é o valor da marcação inicial;
- $P \cap T = \emptyset$ e $P \cup T \neq \emptyset$.

Uma estrutura de Rede de Petri $N = (P, T, F, W)$ sem marcação inicial é denotada N .

Uma Rede de Petri com marcação inicial é denotada (N, M_0) .

2.1.3 Grafo de redes de Petri

Embora a definição nos ajude a estabelecer a base teórica de uma Rede de Petri, poderá ser difícil perceber o comportamento de um modelo expresso através da definição apresentada. Desse modo, é possível utilizar uma representação alternativa através de um grafo, para facilitar a percepção do comportamento do sistema e simplificar a sua modelação (Petri, 1962). Uma representação de uma Rede de Petri no seu grafo é constituída por três elementos geométricos, círculos ou elipses representando lugares 2.1a, barras, quadrados ou retângulos representando transições 2.1b e arcos dirigidos que representam as relações de fluxo de informação 2.1c, ou seja, as funções de entrada e saída de uma transição.

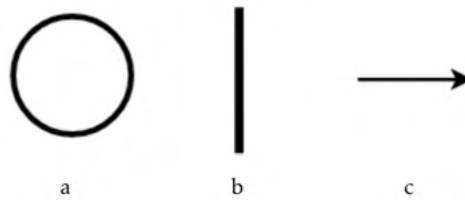


Figura 2.1: Elementos de um Grafo de Redes de Petri.

Como exemplo de um grafo de Rede de Petri temos a Figura 2.2 onde podemos identificar a estrutura da Rede de Petri como também perceber facilmente as relações entre os vários elementos. Como a estratégia de modelação numa Rede de Petri pode ser vista como baseada em eventos discretos, é relevante perceber que os lugares representam estados, ou seja, um componente passivo onde se pode acumular marcas e representar estados do modelo, e as transições representam elementos dinâmicos, que estão associadas a mudanças de estado no sistema, i.e., o consumo de um recurso ou a produção do mesmo, a transição de elementos ou a simples alteração de propriedades num ambiente.

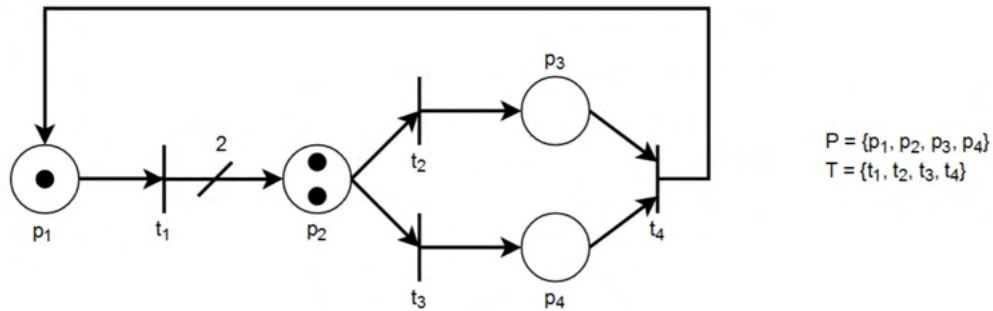


Figura 2.2: Exemplo de um Grafo de Redes de Petri.

2.1.4 Marcação em redes de Petri

Como na definição 1 uma rede de Petri tem uma marcação inicial definida como M_0 , um vetor com a dimensão do número de lugares existentes na Rede de Petri, em que cada número (não negativo) do vetor representa o número de marcas existentes no lugar correspondente da Rede de Petri. Graficamente marcas são representados como pequenos círculos negros dentro dos lugares ou números inteiros não negativos (Peterson, 1977).

Por exemplo para a Figura 2.2 a matriz M_0 de marcação inicial seria $P \rightarrow \{1, 2, 0, 0\}$.

2.2 Classes de Redes de Petri

As classes de Redes de Petri podem ser divididas em dois grandes grupos: Rede de Petri **Autónomas** e Rede de Petri **Não-Autónomas**, em que cada um desses grupos pertencem várias classes de Redes de Petri com propriedades e semânticas de execução diferentes.

A principal diferença entre os dois grupos de classes referidos pode ser associada ao facto de que as Redes de Petri Autónomas apresentam a estrutura referida nas secções anteriores, correspondentes a uma representação abstrata do comportamento de um sistema, enquanto que, as Redes de Petri Não-Autónomas possuem extensões à estrutura “convencional” de uma Rede de Petri, em particular a sua semântica de execução é alterada no sentido de considerar características físicas do sistema, nomeadamente dependências de sinais e eventos físicos ou de condições associadas a tempo (L. F. d. S. Gomes, 1997).

Recorrendo a uma classificação alternativa das classes de Redes de Petri, estas podem ser divididas em três níveis:

- Primeiro nível (baixo-nível): Os lugares têm entre zero ou uma marca, representando condições;
- Segundo nível (baixo-nível): Os lugares têm zero ou mais marcas, todas elas indiferentes entre si;
- Terceiro nível (alto-nível): Os lugares podem conter marcas específicas e com uma estrutura de dados associada, ou seja cada marca tem uma identidade e pode ser

diferenciada de todas as outras marcas e pode representar um elemento específico e por vezes único num sistema.

2.2.1 Propriedades das redes de Petri

De modo a facilitar o entendimento dos capítulos seguintes são expostas algumas propriedades relevantes das redes de Petri. Estas propriedades são atribuídas à rede dependendo da sua estrutura e da marcação inicial M_0 (Cabasino, Giua & Seatzu, 2013).

Propriedades:

Definição 2. Se um lugar só tiver entre zero ou uma marca em todas as marcações alcançáveis, diz-se um lugar seguro; se todos os lugares de uma Rede de Petri forem seguros então a Rede de Petri diz-se **segura** (Cabasino et al., 2013).

Definição 3. Se o número de marcas na Rede de Petri se mantiver constante e igual à marcação inicial M_0 após o disparo de qualquer transição, então a Rede de Petri é **conservativa** (Cabasino et al., 2013).

Definição 4. Uma marcação M_n diz-se **alcançável**, se existir uma sequência de disparos de transições que transforme a marcação inicial M_0 em M_n (Cabasino et al., 2013).

Definição 5. Uma marcação diz-se **home state** se esta for alcançável a partir de outra qualquer marcação. Se numa Rede de Petri a marcação inicial M_0 for uma marcação **home state** então a Rede de Petri diz se **reversível** (Hujisa & Devillers, 2018).

Definição 6. Uma transição diz-se **morta** quando não existe nenhuma marcação M_n que a torne habilitada a disparar. Se todas as transições forem mortas a Rede de Petri diz se **morta** (Cabasino et al., 2013; Hujisa & Devillers, 2018).

Definição 7. Uma transição diz-se **viva-nível 1** quando existe pelo menos uma marcação M_0 que a torne habilitada a disparar. Se todas as transições forem vivas-nível 1 para todas as marcações alcançáveis a Rede de Petri diz se **viva-nível 4** ou **viva** (Cabasino et al., 2013).

2.2.2 Habilitação e disparo de transições

Para a execução de uma Rede de Petri é necessário que ocorra o disparo de transições, que modelam o acontecimento de eventos. Mas para que uma transição dispare é necessário que esta esteja habilitada, e para isso é necessário que todos os lugares anteriores a essa transição, e ligados à mesma por arcos com a direção do lugar para a transição e com peso $W(p, t)$, tenham no mínimo um número de marcas igual a $W(p, t)$ (Murata, 1989; Reisig, 2013).

Para ocorrer o disparo além da transição estar habilitada, se estivermos perante uma Rede de Petri estruturalmente ou simplesmente limitada é necessário que todos os lugares posteriores à transição e que estejam ligados por um arco com direção da transição para

o lugar e um peso de $W(t, p)$ tenham no mínimo “espaço para receber” um número de marcas igual a $W(t, p)$, ou seja, se o seu número de marcas for aumentado em $W(t, p)$ estes ficam no máximo com a capacidade máxima (Murata, 1989).

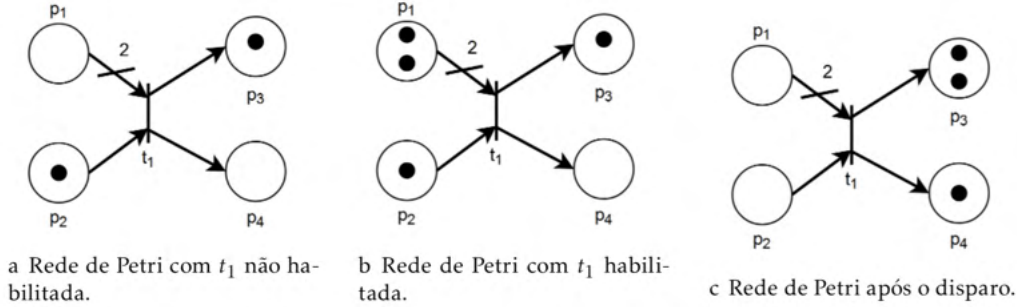


Figura 2.3: Exemplo de habilitação e disparo de uma Rede de Petri.

Considerando o exemplo da Figura 2.3 temos três instâncias de uma mesma amostra de uma Rede de Petri, em que na Figura 2.3a como explicado anteriormente, a transição t_1 não está habilitada porque embora o lugar p_2 tenha marcas suficientes para o peso do arco $W(p_2, t_1)$ o lugar p_1 não tem as duas marcas exigidos pelo arco de peso $W(p_1, t_1)$.

Em contrário na instância seguinte Figura 2.3b tanto p_1 como p_2 têm respectivamente as marcas necessárias $W(p_1, t_1)$ e $W(p_2, t_1)$, para estar habilitada. Se estivéssemos perante o caso de uma Rede de Petri limitada e a capacidade do lugar p_3 fosse de uma marca então para a transição t_1 disparar seria necessário que p_3 não tivesse a marca que tem na imagem.

Quando uma transição t dispara são retirados dos lugares anteriores à transição t e ligados por arcos à mesma transição com direção do lugar para a transição, o número de marcas correspondentes ao peso desse respectivo arco $W(p, t)$. De seguida são adicionados a todos os lugares posteriores à transição t e ligados por arcos à mesma transição com direção da transição para o lugar, o número de marcas correspondentes ao peso desse respectivo arco $W(t, p)$.

Como podemos ver na última instância da Figura 2.3c após o disparo da transição t_1 são retirados do lugar p_1 o número de marcas correspondentes ao peso do arco que liga p_1 a t_1 ou seja $W(p_1, t_1)$ e o mesmo processo acontece para p_2 . De seguida são adicionados aos lugares p_3 e p_4 o número de marcas correspondentes ao peso dos arcos que liga t_1 aos lugares p_3 e p_4 , respetivamente $W(t_1, p_3)$ e $W(t_1, p_4)$. Deste modo a marcação antes da transição t_1 disparar temos $M_x : P \rightarrow \{2, 1, 1, 0\}$ e após o disparo ficamos com $M_{x+1} : P \rightarrow \{0, 0, 2, 1\}$.

2.2.3 Conflitos em redes de Petri

Os conflitos são situações de modelação que podem ser associadas a transições, pois são estes nós da rede de Petri que lhe fornecem a dinâmica. Duas transições dizem-se em

conflito estrutural se partilharem o mesmo lugar de entrada (Itmi, 1993), ou seja, i.e. na Figura 2.4a o lugar p_1 é lugar de entrada para t_1 e t_2 .

Este conflito torna-se um **conflito efetivo**, se no caso de o lugar p_1 for limitado a uma marca, e tiver presente essa marca, então apenas uma das transições t_1 ou t_2 pode disparar, este tipo de conflito pode ser chamado de “*forwards conflict*” (Itmi, 1993; Murata, 1989).

No caso de termos lugares k -limitados podemos também segundo (Nielsen, Plotkin & Winskel, 1981) ter “*backwards conflicts*”, exemplificado na Figura 2.4b em que se p_2 for limitado a uma marca e se t_1 e t_2 tiverem habilitadas a disparar então teremos uma situação de contato pois apenas uma das transições poderá disparar.

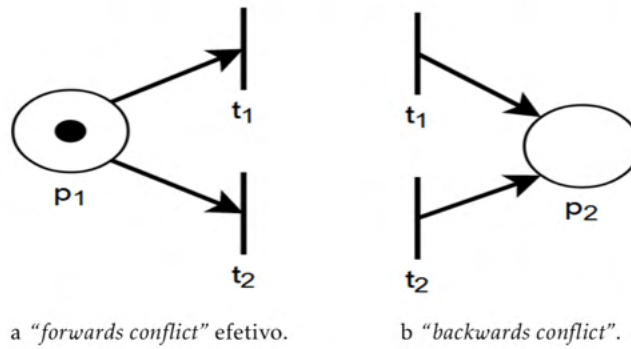


Figura 2.4: Exemplo de conflitos em Redes de Petri.

Ao modelar uma estrutura de Redes de Petri uma das maneiras de resolver conflitos é de fornecer um atributo de prioridade a cada transição de modo a selecionar qual a transição a dispara num conflito efetivo (L. Gomes et al., 2007).

2.2.4 Contendas em redes de Petri

A estrutura contenda em redes de Petri foi inicialmente definida em (Itmi, 1993), este define uma *contenda* como sendo um conjunto de transições que contêm conflitos estruturais, em que dentro dessa estrutura contenda pode ou não existir uma prioridade para cada transição (Barros, 1996).

O autor (Barros, 1996) definiu formalmente o conjunto das contendas de uma Rede de Petri como sendo:

$$CO = \{C \mid (C \in 2^T) \wedge (\forall t_1 \in C \exists t_2 \in (C - t_1) : \bullet t_1 \cap \bullet t_2 \neq \emptyset)\} \quad (2.1)$$

Em que cada transição tem um valor de prioridade atribuído, sendo que na ausência da atribuição da mesma é atribuído o valor um à prioridade de todas as transições dessa contenda, sendo que nesse caso seria escolhida aleatoriamente a transição a disparar (Lima, 2009).

Dois exemplos de contendias podem ser vistos na Figura 2.5.

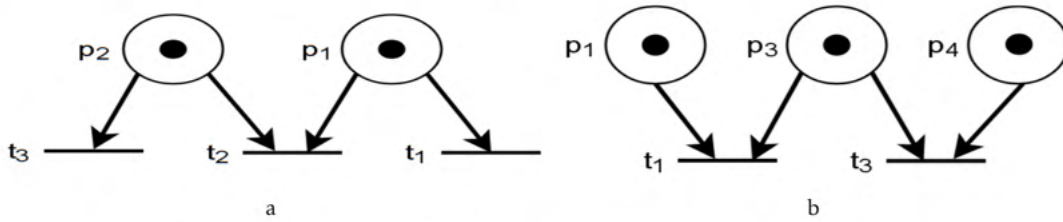


Figura 2.5: Exemplo de Contendas em Redes de Petri.

Na Figura 2.5 temos dois exemplos de contendias em que na Figura 2.5a temos um conflito direto efetivo entre t_1 e t_2 e t_2 e t_3 , mas também temos um conflito indireto entre t_1 e t_3 .

No segundo exemplo Figura 2.5b é representado um conflito direto entre t_1 e t_3 .

Quanto ao disparo de contendias baseando-me no trabalho de (Barros, 1996) as transições de cada contenda são agrupadas e são verificadas por ordem de prioridade caso exista, e dispara a transição que primeiro se encontrar na ordem apta para disparar. No caso de não existir prioridade associada, a ordem das transições é aleatória (Lima, 2009).

2.2.5 Grafo do Espaço de estados de redes de Petri

O grafo do espaço de estados de uma rede de Petri é uma ferramenta poderosa de análise das propriedades das Redes de Petri, mas com limitações aquando da análise de Rede de Petri complexas e grandes.

O grafo do espaço de estados de uma Rede de Petri tem diferenças em relação à árvore de alcançabilidade, pois um é um grafo e o outro funciona como uma árvore de estados. E o grafo do espaço de estados demonstra melhores resultados que a árvore de alcançabilidade (Ye, Zhou & Song, 2003).

O grafo de espaço de estados de uma Rede de Petri é um grafo direcional onde os nós representam marcações alcançáveis e onde os arcos representam o disparo de uma transição específica da rede, que causa uma evolução na rede da marcação de onde o arco originou para a marcação onde o arco termina.

2.2.6 Classe de redes de Petri *Input-Output Place-Transition* (IOPT)

As classes de redes de Petri *Input-Output Place-Transition* integram-se no grupo de Rede de Petri **não-autónomas**, derivando da classe *Lugar-Transição* (L. Gomes et al., 2007) do grupo de Rede de Petri **autónomas** (Reisig, 2012). Este tipo de Rede de Petri é bastante utilizada para modelar controladores (L. Gomes et al., 2014).

As redes IOPT inserem-se no grupo de redes de Petri não-autónomas, pois são adicionadas extensões de representação e propriedades à rede, que permitem representar na

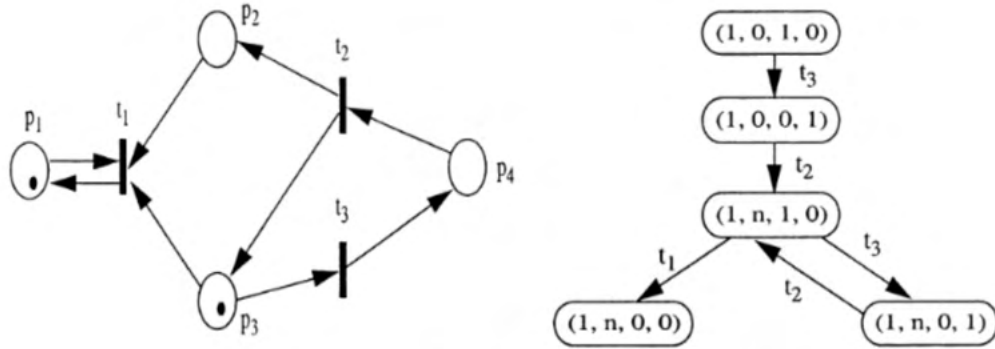


Figura 2.6: Exemplo de um Grafo de Espaço de Estados de uma Rede de Petri (Ye, Zhou & Song, 2003).

Rede de Petri uma simulação do comportamento do controlador em relação ao ambiente externo, estas extensões são representadas a partir de sinais de *Input* e sinais de *Output*. Deste modo os sinais de *Output* nos lugares assim como os sinais *Input* para as guardas das transições podem ser constituídos por expressões algébricas, variáveis e funções, podendo as transições também gerar sinais de saída (L. Gomes et al., 2014).

Na classe *IOPT* também temos presente o atributo de prioridade, a existência de arcos de teste e o atributo de peso nos arcos. Sendo que o atributo de prioridade pertence às transições e serve de resolução para conflitos. Os arcos de teste por sua vez servem como arcos de verificação, ou seja, verificam a presença ou não de marcas num lugar ligado pelo respetivo arco de teste, estes só podem ser ligados de lugares para transições e não de transições para lugares e não retiram marcas do lugar de onde provêm. O peso nos arcos foi referido nos capítulos anteriores.

Uma característica bastante importante desta classe de redes é de, na sua semântica de execução, utilizar o conceito de *maximal step* (isto significa que todas as transições habilitadas a disparar, irão disparar no mesmo passo de execução), bem como irão disparar apenas uma vez por passo de execução¹ mesmo que estivessem habilitadas a disparar múltiplas vezes.

Desta forma definimos um passo de execução como sendo o período entre dois *tics*, sendo estes, instantes específicos no tempo em que a Rede de Petri *IOPT* evolui. Os *tics* são definidos por um relógio global exterior (L. Gomes et al., 2007).

Deste modo é apresentado nos próximos parágrafos a definição formal das Redes de Petri da classe *IOPT* de acordo com (L. Gomes et al., 2007):

Definição 8 (Interface de Sistema (L. Gomes et al., 2007)). A interface de sistemas controlados através de redes *IOPT* é um tuplo $ICS = (IS, IE, OS, OE)$ e que satisfaz os seguintes requisitos:

¹Conceito de *Single Server* (L. Gomes et al., 2014)

1. IS é um conjunto finito de sinais de entrada;
2. IE é um conjunto finito de eventos de entrada;
3. OS é um conjunto finito de sinais de saída;
4. OE é um conjunto finito de eventos de saída;
5. $IS \cap IE \cap OS \cap OE = \emptyset$.

Definição 9 (Estado da entrada do sistema (L. Gomes et al., 2007)). Dado um controlador com uma interface $ICS = (IS, IE, OS, OE)$ (Def. 8, o estado da entrada do sistema é definido pelo par $SIS = (ISB, IEB)$ em que:

1. ISB é um conjunto finito de sinais de entrada: $ISB \subseteq IS \times \mathbb{N}_0$;
2. IEB é um conjunto finito de eventos de entrada: $IEB \subseteq IS \times \mathbb{B}$.

O conjunto de expressões booleanas diz-se BE e $Var(E)$ é a função que retorna o conjunto de variáveis numa expressão E .

Definição 10 (Rede $IOPT$ (L. Gomes et al., 2007)). Dada uma interface $ICS = (IS, IE, OS, OE)$ (Def. 8) de um controlador a rede $IOPT$ é um tuplo $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ que satisfaz os seguintes requisitos:

1. P é um conjunto finito de lugares;
2. T é um conjunto finito de transições (separados de P);
3. A é um conjunto de arcos, tal que $A \subseteq ((P \times T) \cup (T \times P))$;
4. TA é um conjunto de arcos de teste, tal que $TA \subseteq (P \times T)$;
5. M é uma função de marcação: $M : P \rightarrow \mathbb{N}_0$;
6. $weight : A \rightarrow \mathbb{N}_0$;
7. $weightTest : TA \rightarrow \mathbb{N}_0$;
8. $priority$ é uma função parcial de inteiros não negativos aplicados a transições: $priority : T \rightarrow \mathbb{N}_0$;
9. isg é uma função parcial de expressões booleanas, associadas a sinais de entrada, e que se aplica à guarda de transições (onde todas as variáveis são sinais de entrada): $isg : T \rightarrow BE$, onde $\forall eb \in isg(T), Var(eb) \subseteq IS$;
10. ie é um conjunto de eventos de entrada, gerados a partir de sinais de entrada, e que representam condições de habilitação para transições: $ie : T \rightarrow IE$;
11. oe é um conjunto de eventos de saída, e com uma dependência de transições de forma a serem gerados: $oe : T \rightarrow OE$;
12. osc é um conjunto de sinais de saída, ativados em função de uma série de condições associadas à marcação dos lugares (que permitem a ativação do estado dos sinais de saída em função da marcação do lugar): $osc : P \rightarrow P(RULES)$, onde $RULES \subseteq (BES \times OS \times \mathbb{N}_0)$, $BES \subseteq BE$ e $\forall e \in BES, Var(e) \in ML$ com ML sendo um conjunto de identidades para cada marcação do lugar depois da execução de um passo: cada, marcação de lugar tem um identificador associado, que é usado quando executado o código gerado

Nas seguintes definições será usado $M(p)$ para designar a marcação do lugar p numa Rede de Petri de marcação M , e $\bullet t$ para representar os lugares de entrada para uma transição t ou de um conjunto de transições S tal que:

- $\bullet t = \{p | (p, t) \in A\};$
- $\bullet S = \{p | (p, t) \in A \wedge t \in S\};$
- $\diamond t = \{p | (p, t) \in TA\};$
- $\diamond S = \{p | (p, t) \in TA \wedge t \in S\}.$

Definição 11 (condição de habilitação para disparo das transições (L. Gomes et al., 2007)). Dado uma rede $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ (Def. 10) e uma interface de um sistema $ICS = (IS, IE, OS, OE)$ (Def. 8) entre N e o estado de entrada do sistema $SIS = (ISB, IEB)$ (Def. 9), a transição t , sem conflitos estruturais, está habilitada a disparar, num certo *tic* se :

1. $\forall p \in \bullet t, M(p) \geq weight(p, t);$
2. $\forall p \in \diamond t, M(p) \geq weightTest(p, t)$
3. A guarda do sinal de entrada da transição t for verdadeira (*true*) para a ligação do sinal de entrada: $isg(t) < ISB >;$
4. $(ie(t), true) \in IEB.$

Se no caso de uma transição t estar em conflito com outras transições então além das condições acima, essa transição t só está habilitada se tiver a maior *priority* na respetiva contenda CS : $\forall t' \in CS, t' \neq t \Rightarrow priority(t') \leq priority(t).$

Definição 12 (IOPT passo execução (L. Gomes et al., 2007)). Dado uma rede $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ (Def. 10) e uma interface de um sistema $ICS = (IS, IE, OS, OE)$ (Def. 8) entre N e o estado de entrada do sistema $SIS = (ISB, IEB)$ (Def. 9), e tendo $ET \subseteq T$ como o conjunto de transições habilitadas (Def. 11), então Y é um passo de execução em N se a seguinte condição for satisfeita:

$$Y \subseteq ET \wedge \forall t_1 \in (ET \setminus Y), \exists SY \subseteq Y, (\bullet t \cap \bullet SY) \neq \emptyset \wedge \exists p \in (\bullet t_1 \cap \bullet SY),$$

$$(weight(p, t_1) + \sum_{t \in SY} weight(p, t) > M(p))$$

Definição 13 (Ocorrência do passo de execução e marcação sucessora (L. Gomes et al., 2007)). Dado uma rede $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ (Def. 10) e uma interface de um sistema $ICS = (IS, IE, OS, OE)$ (Def. 8) entre N e o estado de entrada do sistema $SIS = (ISB, IEB)$ (Def. 9), a ocorrência de um passo de execução Y (Def. 12 na Rede de Petri N retorna uma Rede de Petri $N' = (P, T, A, TA, M', weight, weightTest,$

priority, isg, ie, oe, osc) em a única diferença em de N' em relação a N é a função de marcação M' que é dada pela expressão:

$$M' = \{(p, m - \sum_{t \in Y \wedge (p,t) \in A} \text{weight}(p,t) + \sum_{t \in Y \wedge (p,t) \in A} \text{weight}(t,p)) \in (P \times \mathbb{N}_0) | (p, m) \in M\}$$

2.3 Petri Net Markup Language (PNML)

O formato *Petri Net Markup Language* (PNML) baseado em *eXtensible Markup Language* (XML) é um formato de Redes de Petri, definido em 2011 em (*Systems and software engineering – High-level Petri nets, Part 2: Transfer Format*. 2011), criado para solucionar o problema de importar e exportar Redes de Petri, sendo que o maior desafio na altura era encontrar um formato que permitisse a interoperabilidade entre ferramentas e ao mesmo tempo tivesse a capacidade de representar as várias classes de Redes de Petri, e que permitisse ser lido e editado por humanos a partir de um editor de texto (Weber & Kindler, 2003).

2.3.1 PNML core model

De forma a conseguir estabelecer um standard de definição de redes de Petri, foi necessário estabelecer quais os conceitos comuns a todas as redes de Petri.

Desta forma é então definido o PNML *core model*, que consiste num conjunto de *objects* nomeadamente *places* para definir lugares, *transitions* para definir transições e *arcs* para definir arcos. Estes *objects* contêm a informação essencial do elemento que representam definida como *node*, mas também contêm informação relativa a posicionamento gráfico e anotação na forma de *labels*.

Por sua vez o PNML *Core Model* também define a relação entre os vários *objects*, esta relação é feita através dos *arcs* que contêm a informação de quais elementos está a relacionar sob a forma dos *ids* dos respetivos *objects* (Kindler, 2006).

Na Figura 2.7 é representado o diagrama *Unified Modeling Language* (UML) que representa o PNML *Core Model*.

2.3.2 Petri Net Type Definitions

Como referido anteriormente devido às vários tipo de redes de Petri foi necessário também estabelecer um Standard de definição de novos *labels* e de novas restrições.

Como exemplo um dos tipos de redes de Petri definidos em (*Systems and software engineering – High-level Petri nets, Part 2: Transfer Format*. 2011) são as **Place/Transition-Nets**, esta definição de tipo de rede de Petri, adiciona ao, PNML *Core Model* os atributos de marcação aos lugares assim como o atributo de peso dos arcos. São também definidas ainda algumas restrições i.e. a restrição de relacionar ,através de um arco, dois lugares ou duas transições (Kindler, 2006).

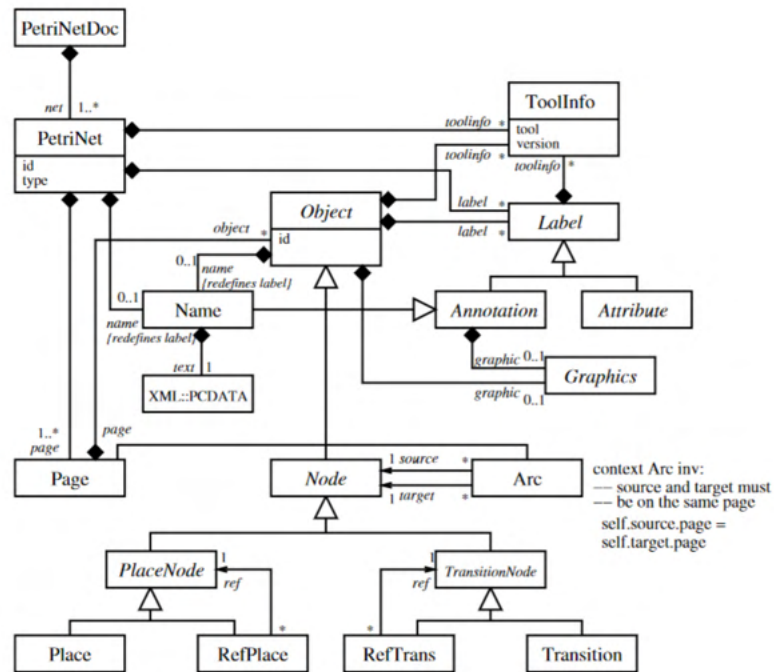
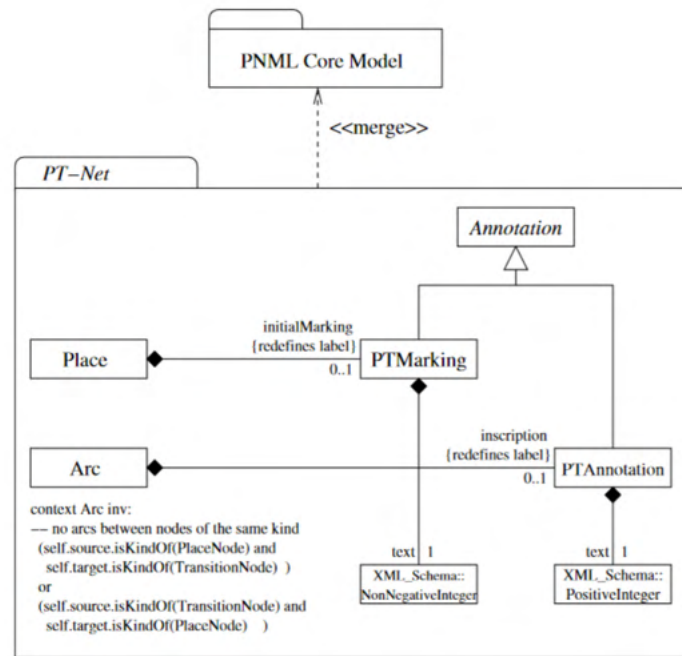


Figura 2.7: PNML Core Model (Kindler, 2006).

Na Figura 2.8 temos o diagrama UML do *Petri Net Type Definition (PNTD)* para *Place/Transition-Nets*.

Figura 2.8: PNTD para *Place/Transition-Nets* (Kindler, 2006).

2.3.3 Mapeamento para XML

Como o PNML ainda é uma extensão do XML em (*Systems and software engineering – High-level Petri nets, Part 2: Transfer Format*. 2011) foram definidos os nomes e atributos dos vários elementos do PNML representativos das classes do diagrama UML da figura 2.7. Desta forma na Tabela 2.1 temos a os elementos e os respectivos atributos do PNML para o *PNML Core Model* retirados de (Kindler, 2006).

Tabela 2.1: Elementos PNML do relativos ao *PNMLCore Model*.

Classes do UML	Elementos PNML	Atributos PNML
PetriNet Doc	<pnml>	—
PetriNet	<net>	id:ID,type:anyURL
Place	<palce>	id:ID
Transition	<transition>	id:ID
Arc	<arc>	id:ID, source:IDRef, target:IDRef
Page	<page>	id:ID
RefPlace	<referencePlace>	id:ID, ref:IDRef
RefTrans	<referenceTransition>	id:ID, ref:IDRef
ToolInfo	<toolspecific>	tool:string, version:string
Graphics	<graphics>	—

2.4 *eXtensible Stylesheet Language Transformations (XSLT)*

A linguagem XSLT, não é uma linguagem de programação, mas sim uma linguagem para transformações de XML, e é uma das partes da linguagem *eXtensible Stylesheet Language (XSL)* que é uma linguagem que descreve como elementos de um XML devem ser exibidos (Clark et al., 1999). À semelhança da relação que *Cascading Style Sheets (CSS)* tem com *HyperText Markup Language (HTML)* (Clark et al., 1999).

Desta forma um ficheiro XSLT descreve um conjunto de regras para transformar um XML em outro ficheiro (XML, HTML, texto, etc.). A transformação é efetuada através da associação de padrões do XML com *templates* definidos pelo ficheiro XSLT, ou seja, é utilizado *XPath* para identificar informação associada com *templates* definidos e é construído o novo ficheiro a partir das regras definidas pelos *templates*.

Nos seguintes ficheiros retirados de (w3schools, s.d.) temos o exemplo de um XML transformado em um HTML neste caso uma tabela.

Listagem 2.1: Ficheiro XML.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <catalog>
3   <cd>
4     <title>Empire Burlesque</title>
5     <artist>Bob Dylan</artist>
6     <country>USA</country>
7     <company>Columbia</company>
8     <price>10.90</price>
9     <year>1985</year>
10  </cd>
11  <cd>
12    <title>Hide your heart</title>
13    <artist>Bonnie Tyler</artist>
14    <country>UK</country>
15    <company>CBS Records</company>
16    <price>9.90</price>
17    <year>1988</year>
18  </cd>
19 </catalog>
```

Como podemos reparar acima temos um ficheiro XML com dois elementos que neste caso retratam CD's com as informações sobre o mesmo. É exemplificado de seguida o ficheiro XSLT que contém regras nomeadamente um *template* para transformar essa informação numa simples tabela em HTML.

Listagem 2.2: Ficheiro XSLT para efetuar a transformação.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3   <xsl:template match="/">
4     <html>
5       <body>
6         <h2>My CD Collection</h2>
7         <table border="1">
8           <tr bgcolor="#9acd32">
```

```

9      <th style="text-align:left">Title</th>
10     <th style="text-align:left">Artist</th>
11   </tr>
12   <xsl:for-each select="catalog/cd">
13     <tr>
14       <td><xsl:value-of select="title"/></td>
15       <td><xsl:value-of select="artist"/></td>
16     </tr>
17   </xsl:for-each>
18 </table>
19 </body>
20 </html>
21 </xsl:template>
22 </xsl:stylesheet>

```

The screenshot displays the result of an XSLT transformation. On the left, the XML source code is shown in a text area, and on the right, the rendered HTML output is displayed.

XML Source Code:

```

<?xml version="1.0">
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
  </cd>
  <cd>
    <title>Hide your heart</title>
    <artist>Bonnie Tyler</artist>
  </cd>
</catalog>

```

Rendered HTML Output:

My CD Collection

Title	Artist
Empire Burlesque	Bob Dylan
Hide your heart	Bonnie Tyler

Figura 2.9: Resultado da transformação por XSLT (w3schools, s.d.).

2.5 Processamento multi-core

2.5.1 Introdução

A figura 2.10 apresenta uma forma abstrata de configuração de um microprocessador segundo (Schauer, 2008):

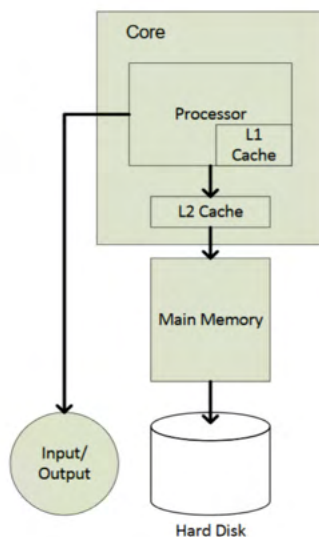


Figura 2.10: Exemplo da configuração de um processador (Schauer, 2008).

Junto ao processador temos a cache de nível um (L1), e imediatamente fora, mas ainda dentro do *core* temos a memória cache de nível dois (L2), estas por sua vez são memórias pequenas e de rápido acesso, e têm a mesma função, mas a níveis diferentes de guardar informação de rápido acesso para as funções do processador.

A memória principal por outro lado é de grandes dimensões, mas mais lenta que as memórias cache, normalmente é utilizada para guardar o ficheiro ou o elemento que está a ser trabalhado pelo processador (Schauer, 2008).

Aquando da necessidade de acesso a outras informações, então é necessário aceder ao disco rígido o que leva significativamente mais tempo.

Na Figura 2.11 podemos ver dois modelos distintos de implementação multi-core. Estas implementações são consequência da necessidade de comunicação dos *cores* entre si ou mesmo com a memória principal.

Neste caso temos dois modelos demonstrados sendo que o modelo do *bus* de comunicação Figura 2.11a é utilizado quando há necessidade de acesso à mesma memória por parte de todos os *cores*, enquanto que o modelo de conexão à Internet Figura 2.11b é utilizado em casos de memória distribuída.

Com a utilização de processadores multi-core verificou-se um aumento de banda e uma diminuição de consumo de energia (Muthana et al., 2005).

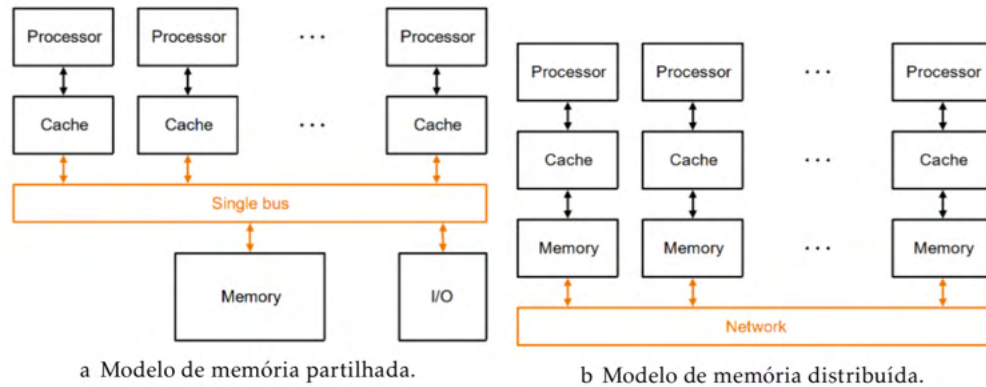


Figura 2.11: Modelos de CPUs de memória partilhada e distribuída (Schauer, 2008).

2.5.2 Multi-threading & multi-cores

Threads são unidades de utilização de processador, ou seja, se tomarmos o exemplo de uma fábrica, sendo esta o processador podemos fazer a analogia de que as *threads* são as linhas de montagem da mesma, e os *cores* os trabalhadores da fábrica, sendo que estes pela analogia podem trabalhar em várias *threads*, mas nunca em duas ao mesmo tempo. A um nível mais formal *threads* contêm um contador que aponta para a instrução que está na *stream*, pela analogia o conjunto de instruções que chegam ao core pela *thread*. Uma *thread* também contêm a informação do estado do core para aquela *thread* (Akhter & Roberts, 2006).

Na figura 2.10 podemos ver o a memória cache L1 dentro do processador, mas este é composto por mais componentes, i.e. unidades de execução, registos de controlo de interrupt, *buses*, estado da arquitetura do processador, etc.

Mas para a definição de *thread* num processador apenas é necessário que este tenha o estado da arquitetura do CPU em relação a essa *thread*. Desta maneira são definidos processadores lógicos dentro de um único processador físico multiplicando os blocos do estado do processador em relação a cada *thread* que este pode estar associado. Sendo que os recursos de execução e de processamento do processador são partilhados pelos vários blocos de estado da arquitetura. Esta técnica é conhecida como *simultaneous multi-threading* (Akhter & Roberts, 2006).

Desta forma um processador pode estar encarregue de uma ou mais *threads*, embora que só possa estar “focado” em uma de cada vez. Mas como este processo é muitas vezes veloz dá a sensação de paralelismo entre as *threads*. As mudanças entre as *threads* de um core são decididas a partir da microarquitetura do CPU.

Embora *multi-threading* forneça algumas melhorias devido ao tempo inativo dos recursos de processamento de um CPU em relação a comportamentos *monothread*, mas apenas nos dá uma sensação de processamento paralelo, quando na realidade não é o que acontece. Enquanto se estivermos a falar de um processador multi-core, como cada core terá os seus recursos de execução independentes dos outros *cores* mesmo se cada core for

monothread teremos um paralelismo de execução de instruções verdadeiro. Também existem processadores multi-core com tecnologia de *multi-thread*, criando soluções bastante eficientes (Akhter & Roberts, 2006).

Na Figura 2.12 é apresentado um resumo e comparação de algumas arquiteturas de processadores.

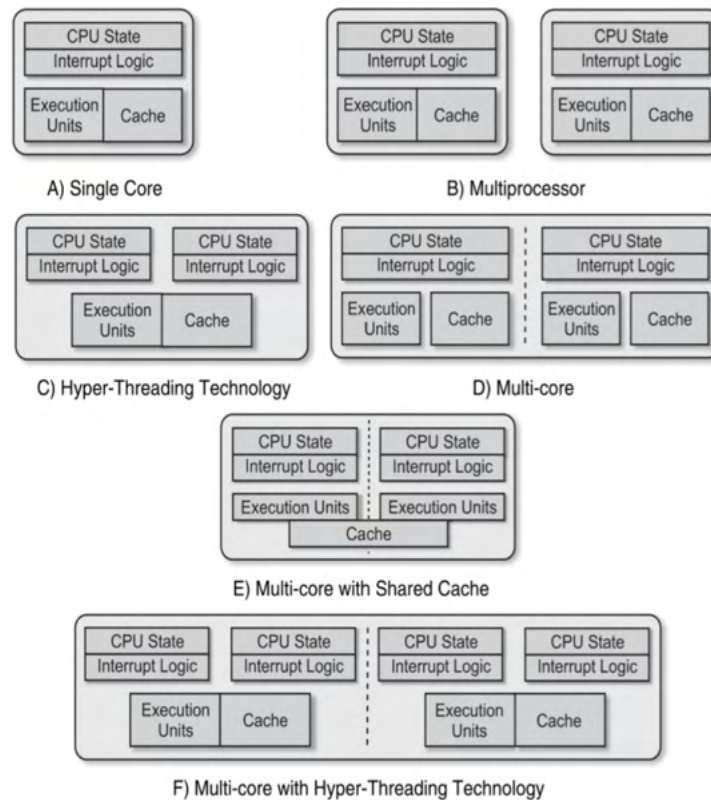


Figura 2.12: Comparação de arquiteturas de processadores *Single-Core*, *Multi-Core*, e *Multi-Threading* (Akhter & Roberts, 2006).

2.6 Programação paralela com OpenMP

A interface de programação de aplicações OpenMP foca-se no suporte de desenvolvimento de programação multi-processo, para sistemas de memória partilhada dada no exemplo da Figura 2.11a.

A OpenMP consiste num conjunto de diretivas ou pragmas, assim como bibliotecas de rotinas de execução, e finalmente um conjunto de variáveis de ambiente. Um benefício da OpenMP é o facto de esta ter compatibilidade com vários compiladores já existentes para as linguagens de C/C++ e Fortran, fornecendo assim uma independência da OpenMP em relação ao hardware que irá executar os programas (OpenMp, s.d.).

A paralelização utilizando OpenMP segue o modelo de *fork-join*, ou seja, existe uma

thread principal onde o programa corre e aquando da necessidade de paralelização, é criado um conjunto de *threads* que irão decorrer paralelamente (*fork*), e no final da zona de paralelização, as *threads* são sincronizadas de modo a voltar à *thread* inicial. Este método pode ser escalado de modo a ter vários níveis internos ao que chamamos *nesting* (Chandra et al., 2001).

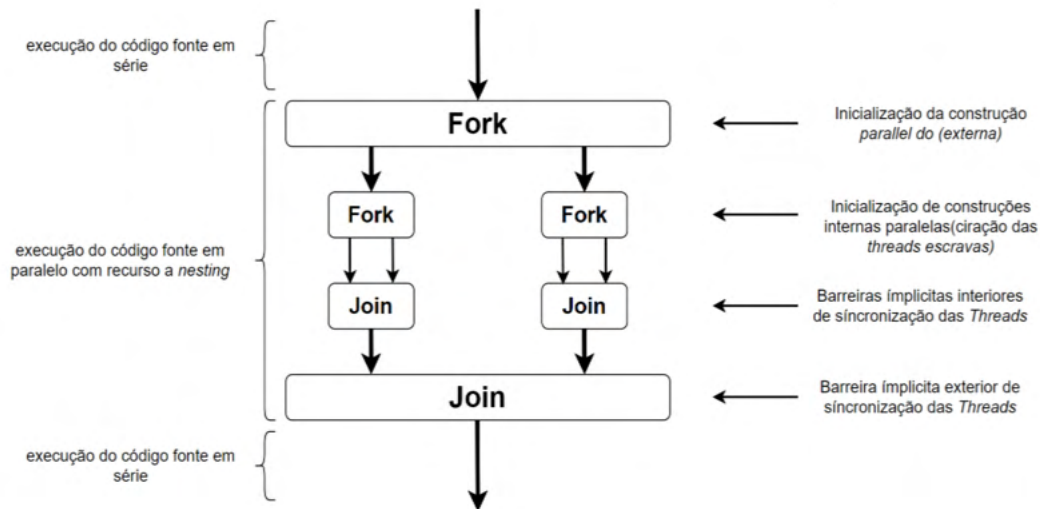


Figura 2.13: Abstração da execução paralela utilizando OpenMP (Chandra et al., 2001)

A nível do código fonte estas divisões da *thread* principal são efetuadas com base nas diretrizes referidas anteriormente, que delimitam regiões de código a ser paralelizadas. Estas regiões de códigos chamamos de construções.

2.7 Ambientes de desenvolvimento de Software

2.7.1 Arduino Integrated Development Environment

O software de desenvolvimento de código para arduinos, *Arduino Integrated Development Environment (IDE)* fornece as ferramentas necessárias para o desenvolvimento e teste de aplicações implementados em arduino.

Composto por um editor de texto e ficheiros, uma consola de texto e com capacidades de comunicação e implementação do código em arduinos, é a ferramenta ideal para desenvolver código para arduino (Arduino, s.d.).

Contém também o próprio compilador, que compila tanto código para arduino como código c/c++, e fornece um amplo repositório de bibliotecas integradas assim como ferramentas para importar bibliotecas externas (Arduino, s.d.).

2.7.2 Vitis

A plataforma Vitis desenvolvida pela empresa Xilinx fornece um ambiente de desenvolvimento de software embutido e aplicações de aceleração nos vários hardwares desenvolvidos pela mesma empresa de entre estes o *Zynq-7000* (Xinlinx, s.d.).

A plataforma oferece um conjunto de ferramentas das mais diversas desde compiladores a debuguers de C/C++ ou até python, assim como um conjunto de bibliotecas de aceleração e bibliotecas de *runtime* (Xinlinx, s.d.).

2.7.3 Visual Studio

Este IDE, desenvolvido pela Microsoft, utilizado para o desenvolvimento de software especialmente dedicado para a framework .NET, e também utilizado para C e C++ no caso desta dissertação.

Incorporado com ferramentas de debug e de release, permite o desenvolvimento rápido de código assim como um debug facilitado com acesso às variáveis em tempo real.

2.8 Plataformas de hardware para teste

2.8.1 Arduino Mega 2560

O arduino Mega é uma placa com um micro controlador ATmega2560, contém um oscilador de cristal de 16MHz, 54 pins de output e input digital e 16 inputs analógicos. Contém uma memória Flash de 256KB sendo 8KB são utilizados para o bootloader, e uma memória SRAM de 8KB.

Graças ao bootloader já pré-programado, podemos carregar código no micro controlador sem a necessidade de um programador externo. O código a ser carregado para o arduino pode ser implementado pelo ambiente Arduino IDE.

2.8.2 Arduino DUE

O arduino DUE é uma placa com um micro controlador Atmel SAM3X8E ARM Cortex-M3 CPU, contém um oscilador de cristal de 84 MHz, 54 pins de output e input digital e 16 inputs analógicos. Contém uma memória Flash de 512KB, e uma memória SRAM de 96KB.

O arduino Due também é programado utilizando o ambiente Arduino IDE.

2.8.3 ZedBoard

A ZedBoard é um Kit de desenvolvimento constituído por uma placa com o chip *Zynq-7000 All Programmable SoC* desenvolvido pela *Xilinx*, o kit contém todas as interfaces necessárias para a comunicação do chip com o exterior. Contém uma memória de 512MB DDR3, assim como uma memória 256Mb Quad-SPI Flash, e finalmente uma interface

para um cartão SD de até 4GB. Contêm também osciladores de 33.333 MHz para o sistema e 100 MHz para a parte lógica.

Quanto ao Zynq-7000 integrado, contêm dois CPU ARM Cortex-A9 integrados, assim como lógica programável. Utilizando o Ambiente Vitis somos capazes de implementar e testar sistemas neste hardware com capacidades amplas.

2.8.4 P8B-E/4L

Esta placa é onde se encontra implementado o servidor para a ferramenta IOPT-Tools. Desenvolvida pela empresa ASUS, é equipada com armazenamento SATA 6G, quatro placas GbE Intel da classe servidor e módulo ASMB5-iKVM para gestão remota out-of-band.

Quanto ao processador incorporado na placa é um Intel(R) Xeon(R) CPU E3-1240 V2 @ 3.40GHz, de quatro *cores* físicos cada um com dois *cores* virtuais, tendo um total de oito *cores*. Frequência do processador é de 1606.898 MHz, com um cache de 8192 KB.

O sistema operativo do servidor é linux, utilizando a distribuição Fedora na versão 19.

2.8.5 Asus ROG GL553VE-77AT5PB2

Este equipamento é utilizado maioritariamente para realizar testes em Visual Studio. Está equipado com um processador Intel® Core™ i7-7700HQ, de quatro *cores* físicos cada um também com dois *cores* virtuais, constituindo um total de oito *cores*, a frequência do processador é de 2.80 GHz e incorpora uma cache de 6 MB.

IOPT-TOOLS E ARQUITETURA

3.1 IOPT-Tools

3.1.1 Introdução

O ambiente de desenvolvimento IOPT-Tools, baseia-se numa interface simples e de acesso gratuito com o suporte de ferramentas de verificação e de geração de código, permitindo a conceção de sistemas e controladores embutidos distribuídos ou não. Auxiliando e facilitando todo o processo de desenvolvimento e teste de controladores e sistemas digitais a partir de um navegador de Internet (Pereira, Moutinho & Gomes, 2014).

O framework especifica os sistemas através da classe IOPT descrita acima, fornecendo recursos de entrada e saída, permitindo assim a criação de controladores prontos a implementar em software ou hardware, capazes de comunicar com um mundo exterior.

Como dito a plataforma incorpora todas as ferramentas necessárias à implementação de sistemas nomeadamente (Pereira et al., 2014):

- Editor de modelos com base na classe de Redes de Petri IOPT;
- Um ambiente de simulação dos modelos criados;
- Um gerador de espaços de estados referente aos modelos produzidos;
- Um sistema de queries para verificação de propriedades;
- Geradores automáticos de código (C, VHDL, JavaScript).

Desta forma as IOPT-Tools permitem uma conceção rápida e distribuída de sistemas, permitindo um ambiente de colaboração entre desenvolvedores, pois pode ser acedido a partir de qualquer lugar por um navegador de internet (Pereira, Moutinho, Ribeiro & Gomes, 2012). As ferramentas de verificação dos modelos e de simulação fornecem

atalhos para uma rápida detecção de falhas e correção das mesmas, mas também graças às ferramentas de geração de código temos um alívio no desenvolvimento do código de protótipos (L. Gomes, Moutinho & Pereira, 2013).

Na Figura 3.1 temos representada a página inicial do ambiente IOPT-Tools.

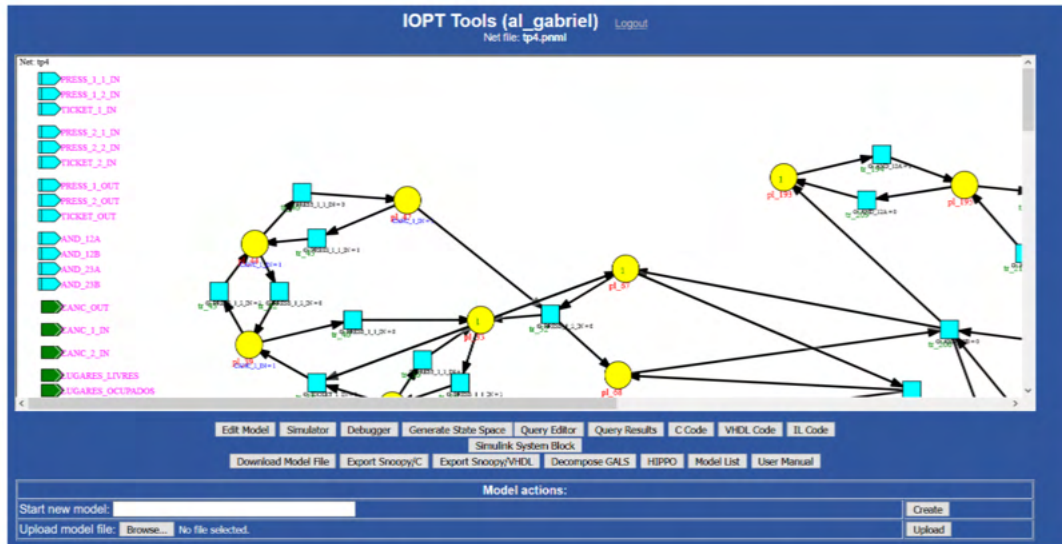


Figura 3.1: IOPT-Tools página principal (Pereira, Moutinho & Gomes, 2014).

3.1.2 Editor

Com o editor, um utilizador tem todas as ferramentas necessárias para uma edição interativa de modelos de rede de Petri IOPT, com suporte de *Scalable Vector Graphics (SVG)* (Pereira et al., 2012).

Este incorpora uma *toolbox* com as opções de colocar lugares e transições e desenhar arcos entre os mesmos, opções de seleção, manipulação dos elementos gráficos etc. No centro podemos ter acesso a uma área de edição e desenho da rede de Petri podendo construir a nossa rede ou manipular uma importada de um ficheiro PNML. E finalmente na direita temos uma área de edição de propriedades do elemento que temos selecionado no momento na área central de edição (Pereira et al., 2014) como representado na Figura 3.2.

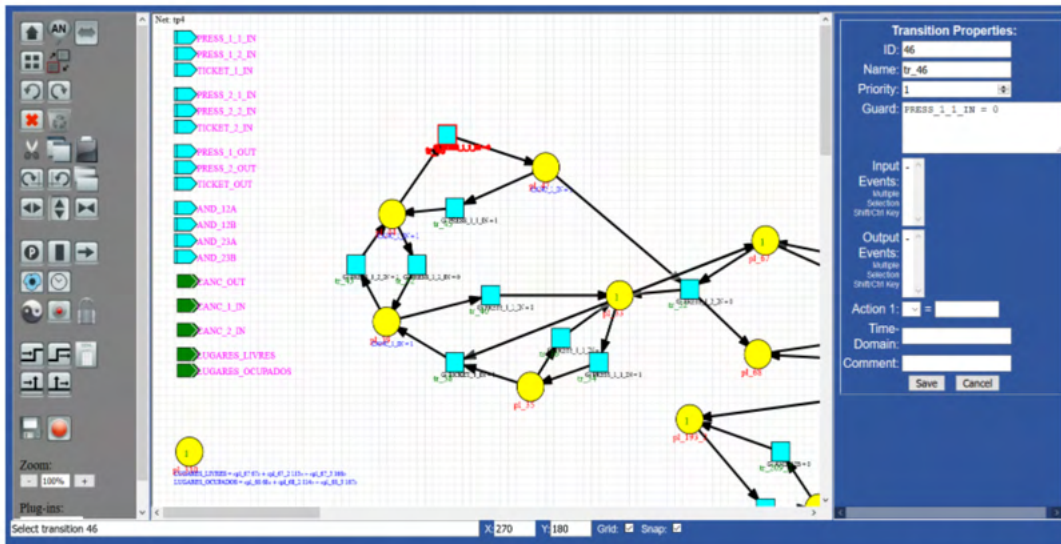


Figura 3.2: IOPT-Tools página do editor (Pereira, Moutinho & Gomes, 2014).

3.1.2.1 Propriedades dos elementos

- **Lugar:** Um lugar assim como as transições, arcos e canais têm um identificador único dado pelo editor aquando da criação do elemento e não pode ser modificado pelo utilizador. Um lugar assim como uma transição, tem uma propriedade de nome e de comentário de modo a facilitar a identificação dos elementos e a percepção das suas funções, tem também uma propriedade de *Time Domain* associado às ferramentas de *Globally asynchronous locally synchronous (GALS)*.

Em relação a propriedades únicas um elemento contém a propriedade de *Bound* e *Initial Marking*, a primeira funciona como limite máximo de marcas que o lugar pode ter e a segunda define a marcação inicial desse lugar.

Devido à natureza das redes IOPT um lugar também contém uma lista de acções que podem definir o estado dos sinais de saída. Acções são compostas por um nome

de um sinal, uma expressão e uma condição. Sendo que esta acção só é avaliada se o lugar durante a execução tiver marcação superior a zero. Se por alguma razão uma acção não for avaliada verdadeira então o valor do sinal reverte para o valor *default*.

Todas estas propriedades estão representadas na Figura 3.3b.

- **Transição:** Assim como referido acima uma transição tem as propriedades de identificador, nome, comentários e domínio temporal.

Relativamente ao disparo as transições têm quatro propriedades associadas. A propriedade de guarda é uma expressão lógica que restringe o disparo da transição e pode utilizar valores de marcação em lugares, sinais de saída e de entrada ou valores concretos. Se não houver guarda definida esta propriedade é considerada verdadeira. A propriedade de eventos de entrada também restringe o disparo da transição, ou seja, esta só dispara se todos os eventos nesta lista ocorrerem no mesmo passo de execução. Quanto à propriedade de eventos de saída, todos os eventos presentes na lista são disparados. E quanto à propriedade de acções, estas têm um comportamento semelhante às acções de lugares sendo que a diferença está no facto de que, quando uma transição não dispara os sinais de output associados a acções revertem para o último valor calculado por qualquer acção de qualquer transição.

Finalmente para a resolução de conflitos as transições têm a propriedade de prioridade, em que quanto menor o valor associado maior é a prioridade dessa transição.

Todas as propriedades de uma transição podem ser vista na Figura 3.3a.

- **Canal:** Este elemento é utilizado para a realização de modelos Globally asynchronous locally synchronous (GALS) e contém as propriedades de identificador, nome, comentário e tipo. Sendo que este último pode ter valores de *Simple async. channel*, *Acknowledge async. channel* e *Not-enable async. channel*. Estas propriedades podem ser vistas na Figura 3.3d.

Mais informação sobre implementação de GALS na IOPT-Tools podem ser consultadas em (Moutinho & Gomes, 2014).

- **Arco:** O elemento arco tem as propriedades de identificador, inscrição e o tipo de arco. Relativamente ao tipo de arco este pode ser do tipo normal de teste ou canal, sendo que este último é automaticamente seleccionado aquando da conexão do arco a um elemento canal. Quanto à propriedade inscrição esta é sinónimo do peso do arco.

As propriedades do arco podem ser vistas na Figura 3.3c.

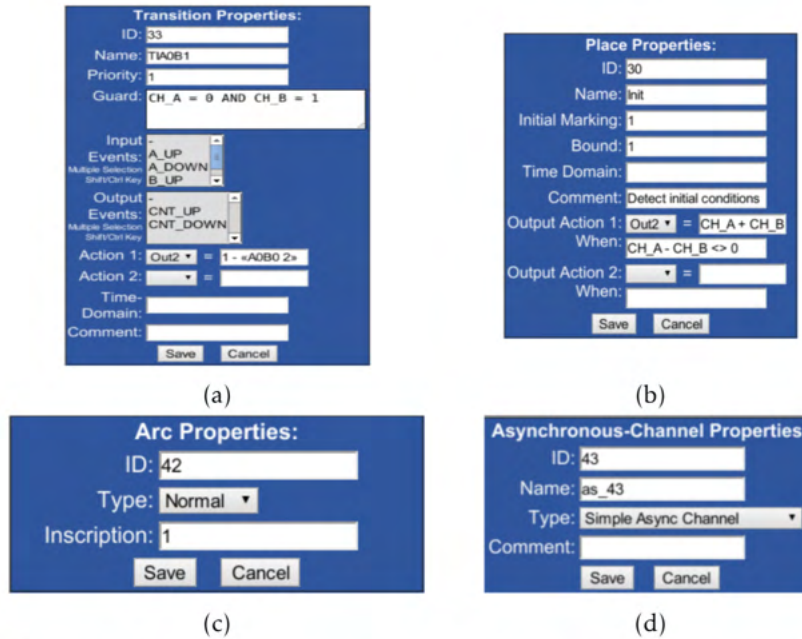


Figura 3.3: Propriedades dos elementos do editor IOPT-Tools (Pereira, Moutinho & Gomes, 2014).

- **Sinais:** Quanto aos sinais estes ao contrario dos elementos referidos acima só têm uma propriedade de referência que é o nome, que serve como identificador do sinal e só pode ser lido, sendo que para mudar o nome do sinal é necessário clicar no botão *Change ID* para mudar todas as referências que estiverem a usar o respetivo sinal.

Relativamente às restantes propriedades, a propriedade modo pode ter os valores, *input* ou *output* que especificam se o sinal é de entrada ou saída respetivamente. A propriedade tipo pode ter valores de sinal inteiro ou booleano. Contém também as propriedades valor, mínimo e máximo. As propriedades de mínimo e máximo no caso de o tipo ser booleano são respetivamente zero e um, no caso do tipo ser inteiro os valores de mínimo é zero e o de máximo pode ser definido pelo utilizador.

As propriedades dos sinais podem ser vistas na Figura 3.5.

- **Eventos:** Assim como os sinais os eventos têm as propriedades modo e identificador único, que contêm os mesmos mecanismos e valores que as propriedades modo e identificador único dos sinais.

Os eventos podem ser autónomos ou associados a um sinal. No primeiro caso estes são maioritariamente para implementações de múltiplos modelos, para casos de necessidade de propagação de eventos no mesmo passo de execução. Relativamente ao segundo tipo estes estão associados ao sinal definido na propriedade sinal.

Se no caso de ser um evento de entrada então este é verdadeiro se o sinal ao qual está associado ultrapassar o valor definido pela propriedade valor, esta ultrapassagem

do valor pode ser no sentido positivo ou negativo sendo definida pela propriedade *Edge*.

Se estivermos perante um evento de saída associado a um sinal então o evento irá influenciar o sinal associado, adicionando ou retirando o valor definido pela propriedade valor do evento à propriedade valor do sinal. O acréscimo acontece se a propriedade *Edge* do evento tiver o valor *Up* e o decréscimo quando a mesma tem o valor *Down*.

Todas as propriedades podem ser vistas na Figura 3.4.

Event Properties:
 ID: B_UP
 Mode: input
 Autonomous: ☐
 Edge: Up
 Level: 0
 Signal: CH_B
 Save Cancel Change ID
 Used by transitions tr-8:TBup1, tr-13:TBup2;

(a)

Event Properties:
 ID: CNT_UP
 Mode: output
 Autonomous: ☐
 Edge: Up
 Level: 0
 Signal: COUNTER
 Save Cancel Change ID
 Used by transitions tr-6:TAup1, tr-8:TBup1, tr-10:TAdn1, tr-12:TBdn1;

(b)

Figura 3.4: Propriedades dos eventos do editor IOPT-Tools (Pereira, Moutinho & Gomes, 2014).

Signal Properties:
 Name/ID: CH_A
 Mode: input
 Type: Boolean
 Value: 0
 Min: 0
 Max: 1
 Save Cancel Change ID
 Used in place-5:A0B1, place-30:Init, transition-31:TIA0B0, transition-32:TIA1B0, transition-33:TIA0B1, transition-34:TIA1B1; Used by input-event:A_UP, input-event:A_DOWN;

(a)

Signal Properties:
 Name/ID: COUNTER
 Mode: output
 Type: Range
 Value: 0
 Min: 0
 Max: 1023
 Wrap Limits: ☒
 Save Cancel Change ID
 Used by output-event:CNT_UP, output-event:CNT_DOWN;

(b)

Figura 3.5: Propriedades dos sinais do editor IOPT-Tools (Pereira, Moutinho & Gomes, 2014).

3.1.2.2 Ferramenta de *ClipBoard*

A ferramenta de *ClipBoard* do editor da IOPT-Tools funciona de forma independente do editor, ou seja, ao copiarmos pedaços do nosso modelo é aberto um novo separador com a ferramenta *ClipBoard* e esta não está dependente do editor ou mesmo da plataforma, sendo possível fecharmos o navegador com a IOPT-Tools e o navegador com a *ClipBoard* manter-se aberto, permitindo a transacção de segmentos de modelos entre os vários modelos que estamos a desenvolver.

Esta ferramenta pode também servir como uma ferramenta de trabalho colaborativo distribuído pois tem a capacidade de exportar e importar segmentos de modelos para a ferramenta *ClipBoard*, permitindo assim a troca dos mesmos por ferramentas de colaboração.

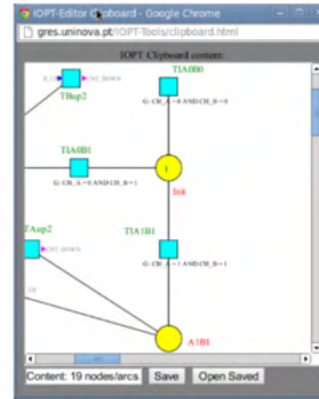


Figura 3.6: Ferramenta *ClipBoard* do editor IOPT-Tools (Pereira et al., 2014).

3.1.2.3 Ferramenta de Edição de Expressões

Esta ferramenta representada na Figura 3.7 aparece sempre que o utilizador clica numa expressão de uma propriedade para a editar, como por exemplo a propriedade guarda das transições, e serve para criar a expressão a partir das opções disponíveis na interface da ferramenta.

A ferramenta para a construção de expressões fornece os seguintes operandos:

- Os lugares presentes no modelo que está a ser editado, de forma que as marcações dos lugares sirvam como variáveis do sistema;
- Os sinais e eventos de entrada disponíveis;
- Os sinais e eventos de saída disponíveis;
- Os vectores ou tabelas disponíveis;
- Valores inteiros literais.

A ferramenta disponibiliza ainda os seguintes operadores:

- Operadores de aritmética;
- Operadores de comparação;
- Operadores de lógica;
- Operadores de Sub – Expressão (i.e NOT e negativo -).

A expressão só é aplicada se estiver correta e aí a ferramenta permite guardá-la e aplicá-la.

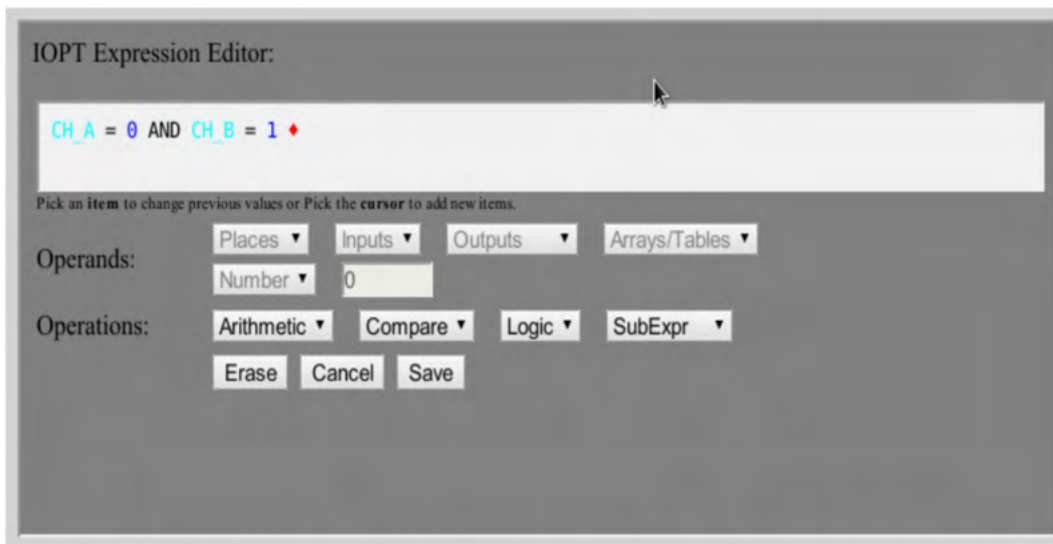


Figura 3.7: Ferramenta de edição de expressões da plataforma IOPT-Tools (Pereira, Moutinho & Gomes, 2014).

3.1.3 Simulador

A ferramenta de simulação da plataforma IOPT-Tools representada na Figura 3.8 é maioritariamente utilizada para debug, permitindo detetar o comportamento dos modelos directamente no navegador de Internet. Deste modo conseguimos simular o modelo e perceber se este está a ter o comportamento pretendido, e caso não esteja podemos voltar para o editor para resolvermos os problemas encontrados.

A ferramenta tem capacidade de execução dos modelos em tempo contínuo ou execução passo a passo, permitindo assim uma verificação mais pormenorizada do comportamento do modelo. Para um debug facilitado numa execução contínua temos a opção de colocar *breakpoints* em transições. O simulador também tem incorporado funcionalidades de história, ou seja, permite voltar atrás nos passos de execução ou mesmo, após correr um modelo, podemos navegar em toda a história de passos de execução que aconteceram. Esta história da execução do modelo pode ser guardada sob a forma de *spreadsheet* de forma a poder ser utilizada por ferramentas externas à IOPT-Tools.

À semelhança da ferramenta de edição de modelos, o simulador é constituído por uma *tollbox* à esquerda com todas as opções de simulação, e de história da simulação. À direita temos uma interface com todos os valores de variáveis do modelo, i.e. marcação dos lugares e os valores dos sinais e eventos do modelo, estes valores podem ser mudados para verificações e debug. Finalmente no meio temos a visualização da simulação, com códigos de cores para identificarmos mais facilmente o estado dos lugares, transições, sinais e eventos. Este código de cores pode ser acedido em (Pereira et al., 2014).

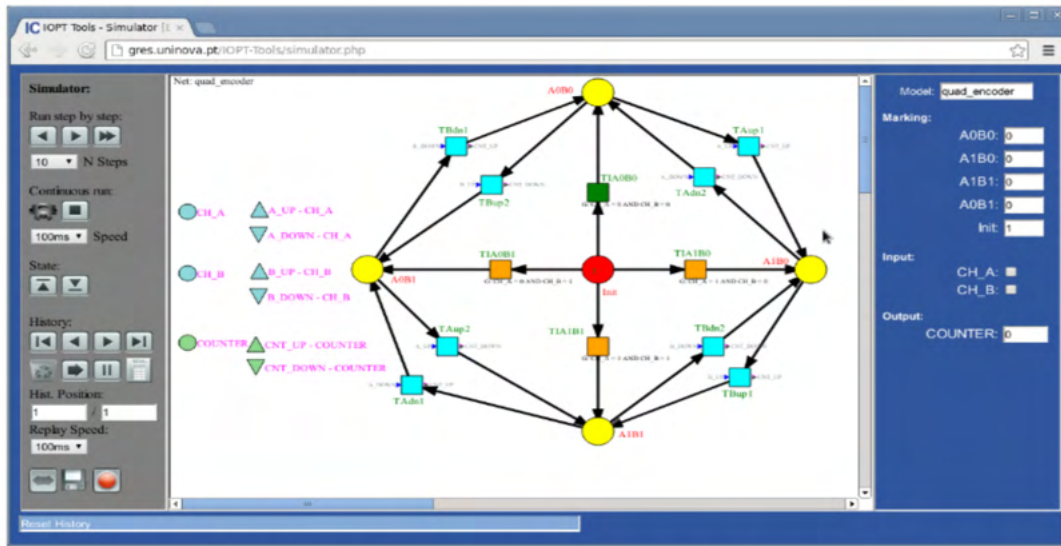


Figura 3.8: Simulador de modelos da plataforma IOPT-Tools (Pereira, Moutinho & Gomes, 2014).

3.1.4 Gerador do espaço de estados

O gerador do grafo do espaço de estados é bastante útil no desenvolvimento de controladores ou sistemas digitais, particularmente na parte de detectar propriedades ou problemas. Alguns impossíveis de identificar apenas com simulação, devido à complexidade dos modelos.

O gerador para criar o grafo utiliza a semântica de execução *Maximal-Step* (Pereira et al., 2011). Ou seja, todas as transições habilitadas a disparar num instante específico são disparadas na etapa de evolução seguinte.

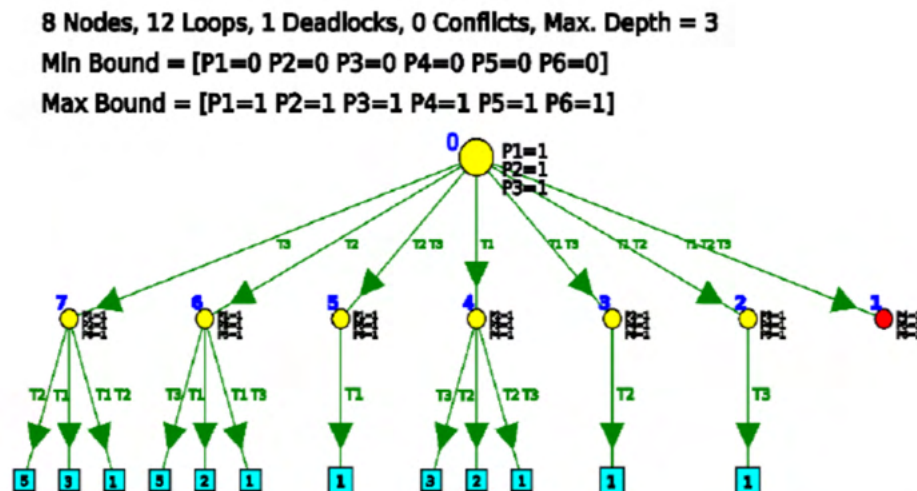


Figura 3.9: Exemplo de um grafo de espaço de estados gerado pela IOPT-Tools (Pereira, Moutinho, Gomes & Campos-Rebello, 2011).

Como o grafo do espaço de estados demonstra qual a marcação nesse estado e os valores dos sinais e eventos de saída, então o estado do grafo é um vector contendo essas informações, e os arcos entre estados contêm o nome da transição que causou a evolução de um estado para o outro. Desta forma podemos detectar directamente no grafo *dead-locks*, conflitos e estados não atingíveis ou estados do nosso modelo que não estávamos à espera que existissem.

Mas devido à complexidade de alguns modelos o grafo do espaço de estados pode ser impossível de visualizar sendo que a ferramenta apresenta apenas algumas estatísticas e resultados de *queries*. Podendo esta informação ser descarregada num formato XML.

3.1.5 *queries*

O sistema de *queries* é composto por uma página de edição e selecção de *queries* e uma página do resultado das mesmas como se pode verificar na Figura 3.10. As *queries* são um conjunto de condições aplicadas a um estado do grafo do espaço de estados explicado anteriormente. As *queries* são úteis para detetar propriedades de acessibilidade e *live-locks*. Até ao momento é possível definir até trinta *queries*, que são verificadas no final da geração do grafo de espaço de estados.

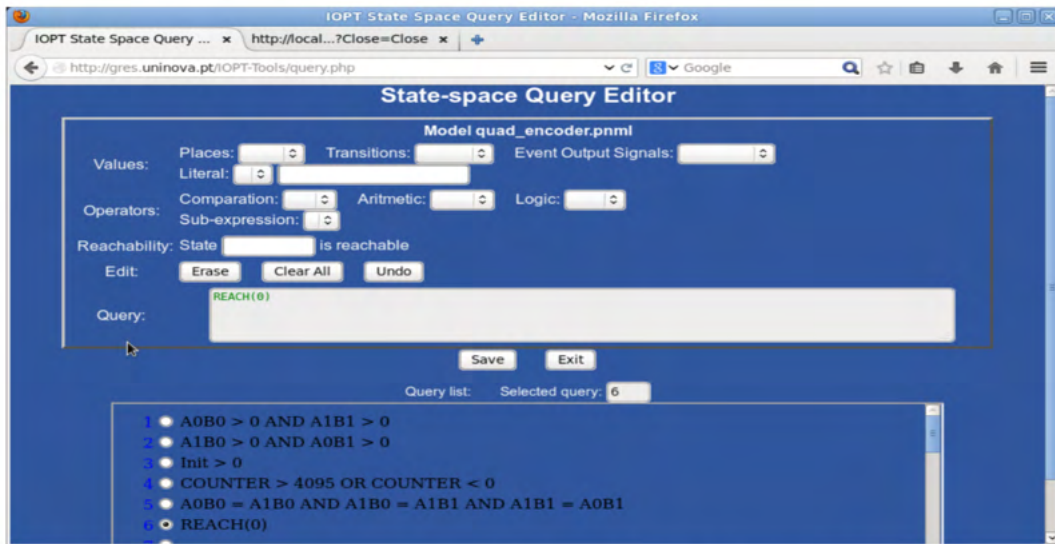


Figura 3.10: Página de edição e selecção de *queries* da plataforma IOPT-Tools (Pereira, Moutinho & Gomes, 2014).

À semelhança do editor de expressões referidos no capítulo 3.1.2.3 o editor de *queries* funciona com operadores e operandos fornecidos pela interface, e o utilizador vai construindo a *query* escolhendo uma opção da interface de cada vez.

Aquando do decorrer da geração do grafo do espaço de estados, como referido no capítulo anterior são exibidos os resultados das *queries* na interface, mas também se pode requisitar o ficheiro de resultados para melhor entendimento do funcionamento do modelo.

3.1.6 Gerador de código VHDL

O gerador de código VHDL gera um ficheiro .zip contendo dois ficheiros, nomeadamente um ficheiro contendo um componente de VHDL com o modelo implementado e outro ficheiro VHDL com a declaração do componente para ser anexada a projectos de VHDL.

O componente tem como entradas e saídas todos os sinais e eventos autónomos da rede de Petri dimensionada, sendo adicionados os sinais de *Clk*, *Reset* e *Enable*. No caso de ser necessário conectar algum sinal de entrada do componente com um sinal externo assíncrono, este precisa ser sincronizado no código com o *Edge* positivo do sinal de *clock*.

Mais informação sobre a geração de código VHDL pode ser encontrada em (Pereira & Gomes, 2013).

3.1.7 Gerador de código C

O gerador de código C recebe um único ficheiro PMNL contendo o modelo para se gerar o código. São gerados cinco ficheiros ANSI C para execução do código, nomeadamente:

- ***net_types.h***: Definição dos tipos de dados e declaração de funções;
- ***net_main.c***: Ficheiro com a função principal para correr o modelo da rede de Petri IOPT;
- ***net_io.c***: Ficheiro com as funções de interface entre o código e os sinais físicos do hardware tanto de entrada como de saída;
- ***net_functions.c***: Ficheiro com a implementação das funções necessárias para a execução do modelo, i.e disparo de transição, eventos, guardas, etc.;
- ***net_exec_step.c***: Ficheiro que contém a função que executa um passo de execução da rede de Petri IOPT;

A geração do código é feita através de código PHP utilizando a linguagem XSLT, referida anteriormente, de modo a transformar o ficheiro PNML em código ANSI C. A geração é efectuada com recursos do servidor IOPT-Tools.

O gerador usa também informação específica sobre rede de Petri e as várias classes de maneira a otimizar o código gerado, i.e. no caso de a rede de Petri IOPT ser segura, então não é necessário que no código gerado sejam reservadas variáveis para o peso dos arcos (Pais, Barros & Gomes, 2005).

3.1.8 Execução do código C gerado

O código C gerado pelo ambiente IOPT-Tools é distribuído pelos cinco ficheiros referidos anteriormente, e a execução do algoritmo pode ser dividida em três partes (L. Gomes et al., 2010):

1. Leitura de sinais e eventos de entrada;

2. Processamento interno;
3. Atribuição de valores aos sinais e eventos de saída.

Tanto a primeira parte como a terceira parte são implementadas nas funções do ficheiro *net_io.c*: enquanto que o processamento interno é efectuado nas funções dos ficheiros *net_exec_step.c*: e *net_functions.c*:. Desta forma na seguinte imagem 3.11 podemos ter uma precessão abstrata da plataforma do algoritmo de execução: Quanto à primeira e à

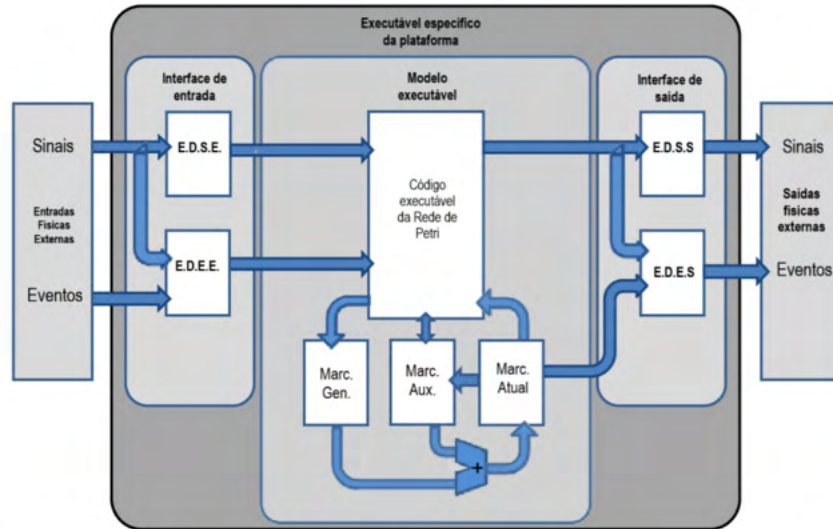


Figura 3.11: Plataforma de execução do algoritmo em C gerado (Campos-Rebello, Pereira, Moutinho & Gomes, 2011).

E.D.S.E: Estrutura de dados dos sinais de entrada; **E.D.E.E:** Estrutura de dados dos eventos de entrada; **E.D.S.S:** Estrutura de dados dos sinais de saída; **E.D.E.S:** Estrutura de dados dos eventos de saída; **Marc. Gen.:** Marcação gerada; **Marc. Aux.:** Marcação auxiliar; **Marc. Atual:** Marcação Atual.

terceira parte do algoritmo estas funcionam como interfaces, como visto na figura 3.11 acima, ou seja, têm a função de receber e traduzir, os sinais e eventos de entrada em estruturas de dados, ou vice-versa, tornando as estruturas de dados em sinais e eventos de saída. Permitindo assim uma transição de informação bidirecional entre o controlador implementado e o mundo exterior a este (Campos-Rebello et al., 2011).

Quanto à segunda parte do algoritmo esta pode ser caracterizada pela execução de um passo de execução da rede de Petri implementada e auxilia-se de três estruturas de dados, todas elas referentes a marcações e que podem ser vistas na figura 3.11:

- Marcação Gerada;
- Marcação Auxiliar;
- Marcação Atual.

No início do passo de execução no ficheiro *net_exec_step.c*, a informação da marcação atual é copiada para a marcação auxiliar, de onde as marcas serão removidas durante o

disparo de transições, estas são disparadas por ordem de prioridade resolvendo assim possíveis conflitos (L. Gomes et al., 2010). Os valores da estrutura marcação gerada no início do passo de execução são inicializados a zero, e conforme as transições são disparadas são adicionadas as informações das novas marcações aos lugares da estrutura marcação gerada. Finalmente são somados os valores das estruturas marcação gerada e marcação auxiliar e é atualizada a estrutura marcação atual com os valores resultantes da soma (L. Gomes et al., 2010). O comportamento do algoritmo relativamente ao passo de execução pode ser visto no seguinte pseudo-código da listagem 3.1 retirado de (L. Gomes et al., 2010).

Listagem 3.1: Pseudo-Código do passo de execução.

```
1  input: sinais de entrada, eventos de entrada
2  output: sinais de saída, eventos de saída
3  begin
4    E.D.S.E.  $\leftarrow \emptyset$ ;
5    E.D.E.E.  $\leftarrow \emptyset$ ;
6    Marc. Atual  $\leftarrow M_0$ ;
7    begin Passo de execução
8      E.D.S.E.  $\leftarrow$  sinais de entrada;
9      E.D.E.E.  $\leftarrow$  eventos de entrada + eventos associados a sinais de entrada;
10     E.D.S.S.  $\leftarrow \emptyset$ ;
11     E.D.E.S.  $\leftarrow \emptyset$ ;
12     Marc. Aux.  $\leftarrow$  Marc. Atual;
13     Marc. Gen.  $\leftarrow \emptyset$ ;
14     foreach transição
15       if condições de disparo da transição são verdadeiras
16         Atualização da Marc. Gen.;
17         Atualização da Marc. Aux.;
18         Atualização de E.D.E.S;
19       end
20     end
21     Marc. Atual  $\leftarrow$  Marc. Gen. + Marc. Aux.;
22     Atualização de E.D.S.S. em conformidade com a Marc. Atual;
23     return sinais de saída e eventos de saída;
24   end Passo de execução
25 end
```

3.1.9 Rede de Petri IOPT usando PNML

Relativamente ao formato PNML de uma classe de Rede de Petri IOPT este deve de conter a informação base como os lugares, transições e arcos, mas também deverá conter as informações relativas à caracterização de entradas e saídas do sistema bem como do seu comportamento.

Desta forma o ficheiro PNMT e quanto à sua gramática RELAX NG (Clark, 2001) define não só os elementos básicos, mas também a informação adicional necessária do sistema como por exemplo, informação sobre conflitos e sincronização. Mais informação pode ser encontrada em (Ribeiro, Moutinho, Pereira, Barros & Gomes, 2011) e (L. Gomes et al., 2007).

- *NetIOInput*: Contem a informação necessária sobre os sinais e eventos de entrada em que os sinais têm valores do tipo *boolean* e *range*, em relação aos eventos estes são do tipo *boolean* e estão associados a um sinal;
- *NetIOOutput*: Contem a informação necessária sobre os sinais e eventos de saída em que os sinais têm valores do tipo *boolean* e *range*, em relação aos eventos estes são do tipo *boolean* e estão associados a um sinal;
- *ConflictSet*: Identificam conflitos ou contendias e definem prioridades para a resolução dos mesmos;
- *SynchronySet*: Identifica um conjunto de transições interligadas através de um canal síncrono para que estas se comportem como uma única transição (Lima, 2009).

Aos lugares foram adicionados os seguintes elementos filho:

- *Bound*: Define a capacidade máxima de marcas que o lugar pode conter;
- *SignalOutputAction*: Permite a activação e desactivação de sinais e o disparo de eventos de saída a partir da marcação do lugar;
- *SubNet*: Relaciona-se com representações hierárquicas e determina se o lugar está associado a uma *subNet* da rede de Petri.

Quanto às transições:

- *NetIOTransGuard*: Representação das condições da transição;
- *NetIOTransInput*: Especificação das dependências da transição em relação aos eventos ou sinais de entrada;
- *NetIOTransOutput*: Especificação da geração ou mudança dos valores de sinais ou eventos de saída aquando do disparo da transição;
- *Priority*: Determina a prioridade de disparo da transição associada ao respetivo *ConflictSet*;
- *subNet*: Relaciona-se com representações hierárquicas e determina se a transição está associada a uma *subNet* da rede de Petri.

E em relação aos arcos:

- *PTArcInscription*: Especificação do peso do arco;
- *ArcType*: Define se o tipo de arco é normal ou de teste;
- *subSource*: Relaciona-se com representações hierárquicas e determina se o lugar de onde o arco inicia está associada a uma *subNet* da rede de Petri;
- *subTarget*: Relaciona-se com representações hierárquicas e determina se o lugar de onde o arco acaba está associada a uma *subNet* da rede de Petri.

3.2 Ficheiros XSLT

3.2.1 Ficheiro *iopt2c_inc*

Este ficheiro XSLT é utilizado para criar o ficheiro *net_types.h*. Entrando em detalhe, o ficheiro começa por estabelecer as regras para serem definidos valores globais constantes, i.e. o número de inputs e outputs, o número de transições e lugares e a versão, etc. De seguida define regras para a criação de estruturas, nomeadamente a estrutura da marcação do modelo, para os sinais e eventos de entrada e saída e também a estrutura para o disparo de transições. São criadas também as definições das funções que os outros ficheiros C irão utilizar. Dentro destas estão incluídas para cada transição as funções:

- *enabled*: Retorna o valor um caso a transição em questão esteja habilitada, caso contrário retorna zero;
- *events*: Retorna o valor um se todos os eventos necessários para o disparo da transição se verificarem verdadeiros, caso contrário retorna zero;
- *guards*: Retorna o valor um se a guarda da transição for verdadeira, caso contrário retorna zero;
- *remove_marks*: É utilizada para remover marcas aos lugares de entrada da transição;
- *add_marks*: É utilizada para adicionar marcas aos lugares de saída da transição;
- *generate_output_events*: É responsável pela geração de eventos de saída associados ao disparo da transição.

E finalmente são definidas regras para a definição de funções para as interfaces de entrada e de saída representadas na Figura 3.11 para comunicação com as entradas e saídas externas, assim como a função do passo de execução na listagem 3.1 e funções de acesso às estruturas definidas inicialmente.

3.2.2 Ficheiro *iopt2c_io*

Este ficheiro é utilizado para especificar as regras para a criação do ficheiro *net_io.c*. O ficheiro começa por descrever regras para a definição de variáveis constantes, e de seguida descreve cinco funções definidas no ficheiro *net_types.h* 3.2.1, nomeadamente:

- *InitializeIO*: Esta função é apenas inicializada de modo a posteriormente ser completada pelo utilizador com o código necessário para a inicialização de *inputs* e *outputs*;
- *GetInputsSignals*: Esta função é criada a partir das definições de inputs criados do PNML aquando da edição do modelo, e inicializa os valores da estrutura dos sinais de entrada e saída com valor de zero, de modo que posteriormente estes valores sejam alterados pelo utilizador;

- *PutOutputSignals*: Assim como a função *GetinputSignals*, é criada a partir das saídas existentes no PNML, e não inicializa nenhum valor nas saídas pois a ligação é considerada responsabilidade do utilizador;
- *LoopDelay*: Esta função é executada entre cada passo de execução, servindo como um delay para diminuir o consumo de energia e a utilização de recursos;
- *FinishExecution*: Embora definida e implementada esta função devolve sempre zero e serve maioritariamente para ser editada pelo utilizador de modo que este consiga definir estados da marcação para terminar a execução do modelo.

3.2.3 *iopt2c_func*

O ficheiro *iopt2c_func* é utilizado para gerar o ficheiro *net_functions.c*, e este começa por definir regras para as funções de gestão da marcação, ou seja, as funções que irão ser usadas no passo de execução para gerir as estruturas das marcações como referido na secção 3.1.8 na explicação da segunda parte do algoritmo, nomeadamente:

- *createInitial_NetMarking*: Atribuí a marcação inicial aos lugares de acordo com os dados guardados no ficheiro PNML;
- *createEmpty_NetMarking*: Atribuí o valor zero a todos os lugares da estrutura de marcação auxiliar *empty_marking*;
- *add_NetMarking*: Função auxiliar que dadas duas estruturas de marcação representativas do modelo fornecido pelo PNML, para cada valor da primeira estrutura soma-o com o respetivo valor da segunda estrutura;
- *init_OutputSignals*: Esta função inicializa a zero os valores das saídas do modelo e aloca a memória necessária para as estruturas dos eventos e sinais de saída.

De seguida o ficheiro XSLT em questão, implementa para cada transição as funções referidas na secção 3.2.1, de acordo com o ficheiro PNML de modo que cada função cumpra o seu papel. Finalmente o ficheiro define as funções que lidam com a geração de eventos e sinais de saída internos e externos, associados a transições e lugares:

- *GenerateInputSignalEvents*: Função que a partir dos sinais de entrada gera os eventos de entrada;
- *GenerateEventOutputSignals*: Função que a partir dos sinais de saída gera os eventos de saída;
- *GenerateTransitionActionOutputSignals* : Função que é responsável pela mudança dos valores dos sinais de saída dependentes do disparo de transições;
- *GeneratePlaceOutputSignals*: Função que dependendo da marcação fornecida e dos sinais de entrada atualiza os sinais de saída dependentes dos lugares e dos respetivos eventos de saída

3.2.4 *iopt2c_exec_step*

O ficheiro *iopt2c_exec_step* é utilizado para criar a o ficheiro *net_exec_step.c* que contem a função do passo de execução (*ExecutionStep*), esta função tem o comportamento descrito na secção 3.1.8.

Esta função começa por usar as funções *createEmpty_NetMarking*, *GetInputSignals*, *GenerateInputSignalEvents*, desta forma executando as partes do algoritmo da listagem 3.1 nomeadamente as linhas oito, nove e treze.

De seguida para cada transição é verificado se as condições de disparo são verdadeiras, nomeadamente se está habilitada, se a guarda da transição é verdadeira e se os eventos para o disparo da mesma são verdadeiros. Se o resultado da verificação for verdadeiro então o disparo ocorre e são removidas as marcas dos lugares de entrada, são adicionadas as marcas nos lugares de saída e são gerados os eventos associados ao disparo da transição. Todos estes processos são feitos utilizando as funções definidas pelo ficheiro *net_types.h* e implementadas no ficheiro *net_functions.c* descritas na secção 3.2.1.

Na listagem 3.2 temos um exemplo da implementação de um disparo de uma transição.

Listagem 3.2: Exemplo da implementação de um disparo de uma transição.

```
1  /* Transition tr_52 */
2  if( t_52_enabled( prev_marking, &avail_marking ) &&
3      t_52_events( &input_events ) &&
4      t_52_guards( prev_marking, inputs, place_out, ev_out ) ) {
5      tfired.t_52 = 1;
6      t_52_remove_marks( &avail_marking );
7      t_52_add_marks( &add_marking );
8      t_52_generate_output_events( &output_events );
9  }
```

De referir que na linha cinco da listagem 3.2 a estrutura de marcação *tfired* irá servir para que no final do ficheiro sejam atualizados os sinais dependentes do disparo de transições.

Por fim é executada a função para somar os valores das estruturas de dados das marcações criadas(*add_marking*, 3.2 linha sete) e existentes(*avail_marking*, 3.2 linha seis), são gerados os sinais e os eventos de saída e é actualizada a estrutura de marcação actual(*prev_marking*, 3.2 linha dois) com a soma referida anteriormente.

3.2.5 *iopt2c_main*

Finalmente o último ficheiro XSLT é utilizado para criar o ficheiro *net_main.c*. Este é o ficheiro principal que utiliza as funções e estruturas de todos os outros. O ficheiro XSLT começa por definir regras para a criação das estruturas de marcação necessárias para a execução da rede de Petri. O ficheiro *net_main.c* foi desenhado para poder ser utilizado pelo Arduino IDE, desta forma contém duas funções, a função *Setup()* e *Loop()*.

Como se pode presumir pelos nomes, a função *Setup()* é chamada uma única vez e tem a responsabilidade de inicializar a estrutura de marcação inicial (*createInitial_NetMarking*

3.2.3), inicializar os valores dos sinais de entrada e valores dos sinais de output a zero (*init_OutputSignals* 3.2.3), inicializar as ligações às saídas físicas (*InitializeIO()* 3.2.2) e finalmente recolher os primeiros valores recebidos das entradas físicas (*GetInputSignals* 3.2.2).

Quanto à função *Loop()* esta está sempre a ser chamada enquanto a função *FinishExecution()* (secção 3.2.2) não fornecer o valor zero, sendo que esta opção só acontece se o utilizador definir regras para a finalização da execução. A função *Loop()* está consecutivamente a chamar em alternância a função do passo de execução (*ExecutionStep* 3.2.4) e a função *GetInputSignals* 3.2.2 de modo a verificar e a recolher os sinais externos de entrada no sistema.

3.3 Algoritmo PHP de geração do código C

Relativamente ao código que gera os ficheiros C como referido anteriormente foi implementado em PHP no servidor das IOPT-Tools e o fluxograma da figura 3.12 representa o funcionamento do algoritmo de uma forma abstrata.

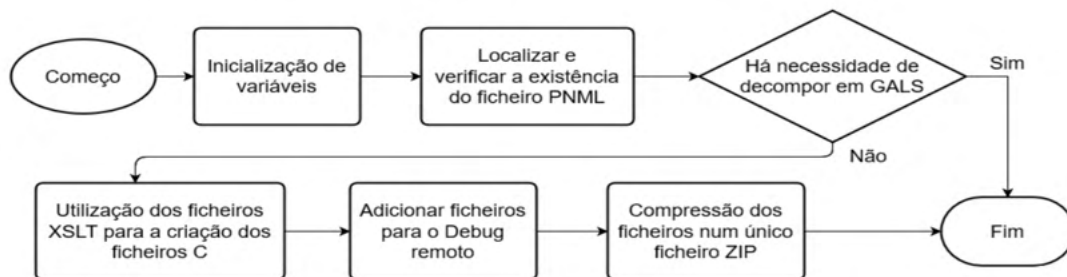


Figura 3.12: Fluxograma do algoritmo de PHP para gerar os ficheiros C.

Entrado em detalhe na parte da criação dos ficheiros C, como referido anteriormente são utilizados os ficheiros XSLT, e para cada um dos ficheiros a ser criado são executadas as seguintes linhas de código, neste caso para o exemplo do ficheiro *net_types.h*.

Listagem 3.3: Exemplo da implementação da criação do ficheiro *net_types.h* em PHP.

```

1  $xslDoc = new DOMDocument();
2  $xslDoc->load("xsl/iopt2c_inc.xsl");
3  $proc = new XSLTProcessor();
4  $proc->importStylesheet($xslDoc);
5  $proc->setParameter('', 'model_version', $mod_time);
6  $proc->transformToURI($pnmlDoc, $dir . "/" . "net_types.h" );
7  unset( $xslDoc );
8  unset( $proc );

```

Explicando o que acontece na listagem 3.3, começamos por criar um objecto *DOMDocument*, que representa um ficheiro inteiro de HTML ou XML e serve de base para a árvore de documentos (php.net, s.d.-a).

De seguida vamos buscar o documento XSLT que iremos usar para criar o documento C, e criamos um objecto do tipo *XSLTProcessor*, este irá servir para fazer a criação do

ficheiro C a partir das regras definidas no ficheiro XSLT e da informação presente no ficheiro PNML (php.net, s.d.-b).

De seguida importamos para o *XSLTProcessor* o ficheiro XSLT que pretendemos utilizar para definir as regras de transformação, definimos os parâmetros da versão atual e da data a que o ficheiro será guardado e procedemos à criação do ficheiro, colocando-o no diretório pretendido. Finalmente libertamos a memória das variáveis que contêm os objetos do *DOMDocument* e do *XSLTProcessor*.

ALGORITMOS DESENVOLVIDOS

4.1 Introdução

Neste capítulo são apresentados e explicados os algoritmos desenvolvidos que constituem a maioria do conteúdo desenvolvido de contribuição à ferramenta IOPT-tools.

O conteúdo desenvolvido está dividido em três partes, sendo elas, o algoritmo em *PHP* para adaptação do PNML da rede de Petri, para que o PNML contenha a informação das contendas e dos conflitos, de modo a facilitar aquando da geração do código C.

A segunda parte engloba todas as modificações feitas nos ficheiros XSLT nomeadamente, *iopt2c_inc*, *iopt2c_func*, *iopt2c_exec_step*, de forma que o código gerado C esteja adaptado para processamento multi-core e aproveitando o conceito de contendas para uma execução mais eficiente.

Finalmente é apresentado o conversor de modelos PIPE XML para modelos PNML, desenvolvido e utilizado durante a já dissertação, criado na ferramenta IOPT-Tools. Este conversor foi criado com intuito de facilitar a utilização de modelos da biblioteca de redes de Petri da ferramenta HIPPO (Hippo, s.d.).

4.2 Algoritmo de adaptação do PNML para contendas

Uma vez que o PNML associado a cada rede de Petri se encontra no servidor, todas as modificações do PNML foram feitas em *PHP* de modo a que o PNML fosse adaptado antes da utilização dos ficheiros XSLT.

Desta forma o algoritmo foi desenvolvido e dividido em cinco funções implementadas em *PHP*, sendo uma delas a função principal *organize_contenda* auxiliada por quatro funções, nomeadamente:

- *processTransition*

- *transitionVectorAttribution*
- *placeContendaAssociation*
- *compareTransitionPriority*

De modo a facilitar a compreensão do algoritmo será explicada a função principal *organize_contenda*, e de seguida as funções auxiliares de uma forma mais detalhada.

Para auxiliar a explicação do algoritmo será utilizada a rede de Petri da Figura 4.1.

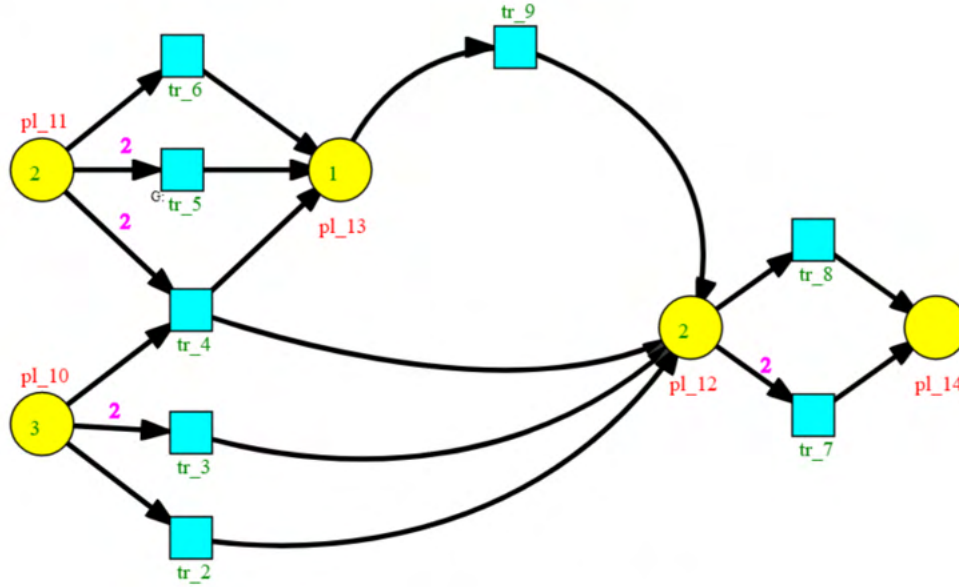


Figura 4.1: Exemplo de rede de Petri.

A rede de Petri apresentada na Figura 4.1 contém oito transições que estão divididas em três contendas.

Relativamente à prioridade das transições:

- **Prioridade 1:** tr_4, tr_7, tr_9;
- **Prioridade 2:** tr_3, tr_5, tr_8;
- **Prioridade 3:** tr_2, tr_6.

Quanto à informação da marcação inicial e o peso dos arcos, estas estão explicitas na Figura 4.1.

4.2.1 Função *organize_contendas*

Esta função é utilizada sempre que for necessário adaptar o PNML de forma a que este contenha a informação sobre a que contenda as transições pertencem, a informação dos conflitos a que cada transição está associada, a informação dos lugares de entrada para

cada contenda e finalmente a informação geral de quantas contendas existem e do número de transições pertencentes a essa contenda.

Listagem 4.1: Pseudo código da função *organize_contenda*.

```

1  input : pnml (PNML a ser modificado)
2  output: pnml (PNML depois de ser modificado)
3  begin
4      contenda ← 0;
5      foreach transition in pnml
6          Adiciona atributo contenda com valor 0 a transition;
7          Adiciona atributo verify com valor 0 a transition;
8          conflicts ← Elemento vazio com nome 'conflicts';
9          Aciciona conflicts como elemento filho a transition;
10     end
11     foreach transition in pnml
12         if atributo contenda in transition == 0
13             contenda += 1;
14             processTransition( pnml, contenda, transition, null);
15         end
16     end
17     transitionVectorAttribution( pnml, contenda);
18     placeContendaAssociation ( pnml);
19
20     return pnml;
21 end

```

A função no primeiro *foreach* (linha 5 da Listagem 4.1), adiciona a cada transição dois atributos novos chamados contenda e verify com o valor zero. É também adicionado um elemento filho a cada transição chamado *conflicts* que irá conter elementos filhos com a informação de cada conflito a que essa transição pertence.

Na Figura 4.2 temos um exemplo de como a transição *tr_2* é representada no PNML após o primeiro *foreach*.

```

<transition id="2" contenda="0" verify="0">
  <name>...
</name>
  <comment>...
</comment>
  <priority>3</priority>
  .
  .
  .
  <conflicts/>
</transition>

```

Figura 4.2: Exemplo da transição de id 2 após o primeiro *foreach*

No *foreach* seguinte (linha 10 da Listagem 4.1) é onde a parte mais pesada a nível de processamento sucede, dentro deste *foreach* para cada transição é verificado se a transição já tem um valor de contenda associado, desta forma não processamos as transições já processadas.

Para as transições ainda não processadas é então chamada a função *processTransition*, esta é uma função que se chama a si própria recorrentemente, e de forma a que sejam processadas todas as transições de uma mesma contenda. Desta forma só é incrementada a variável *contenda* quando se trata de uma transição não processada e que pertence a uma nova contenda.

Ainda na função *organize_contenda* é chamada a função *transitionVectorAttribution*, que adiciona um atributo de prioridade associado à contenda a que esta pertence, sendo utilizado na resolução de conflitos. Esta função também cria no PNML um elemento com a informação de todas as contendas e o número de transições que cada contenda contém.

Finalmente é chamada também a função *placeContendaAssociation*, que adiciona a cada elemento lugar, elementos filhos com a informação das contendas a que este lugar pertence como lugar de entrada ou de saída, ou seja, associa-se o valor das contendas associadas às transições das quais o lugar tem a função de lugar de entrada ou de saída.

4.2.2 Função *processTransition*

Esta função trata de associar à transição recebida como parâmetro de entrada o valor da contenda a que esta pertence. Esta função chama-se a si própria recorrentemente até não serem encontradas mais transições que pertençam a essa contenda.

Na Listagem 4.2 é apresentado o pseudo-código da função.

Listagem 4.2: Pseudo código da função *processTransition*.

```
1  input : pnml (PNML a ser modificado),
2          contenda_number (numero da contenda atual),
3          transitionProc (transição a ser processada),
4          placeCalled (lugar a ser processado quando foi chamada a função)
5  output:
6  begin
7      places_aux ← array vazio;
8      foreach arc in pnml
9          if atributo 'target' de arc == atributo 'id' de transitionProc
10             foreach place in pnml
11                 if atributo 'id' de place == atributo 'source' de arc
12                     Adiciona place a places_aux;
13             end
14         end
15     end
16 end
17 foreach place in place_aux
18     transition_aux ← array vazio;
19     foreach arc in pnml
20         if atributo 'source' de arc == atributo 'id' de place_aux
21             foreach transition in pnml
22                 if atributo 'id' de transition == atributo 'target' de arc
```

4.2. ALGORITMO DE ADAPTAÇÃO DO PNML PARA CONTENDAS

```
23         Adiciona transition a transition_aux;
24     end
25 end
26 end
27 end
28 foreach transition in transition_aux
29     transitionConflicts ← Elemento 'conflicts' de transition;
30
31     if atributo 'contenda' de transition == 0
32         atributo 'contenda' de transition ← contenda_value;
33         hasDependableTransitions ← 0;
34         conflict ← Elemento vazio com nome 'conflict';
35         Adiciona atributo 'place' com valor do atributo 'id'
36         de place_aux a conflict;
37         foreach transition2 in transition_aux
38             if atributo 'id' de transition != atributo 'id'
39             de transition2
40                 if compareTransitionPriority(transition,transition2) <= 0
41                     Modifica o elemento verify de transition2 para 1;
42                     conflictTransition ← Elemento com
43                     nome 'conflictTransition'
44                     e valor do atributo 'id' de transition2;
45                     Adiciona conflictTransition como elemento
46                     filho a conflict;
47                     hasDependableTransitions ← 1;
48                 end
49             end
50         end
51         if hasDependableTransitions == 1
52             Adiciona conflict como elemento filho a transitionConflicts;
53         end
54         if atributo 'id' de transition != atributo 'id' de transitionProc
55             processTransition(pnml, contenda_number, transition);
56         end
57     elseif atributo 'contenda' de transition == contenda_value
58         && placeCalled != null
59         if atributo 'id' de place_aux != atributo 'id' de placeCalled
60             foreach conflict in transitionConflicts
61                 if atributo 'place' de conflict == atributo 'id'
62                 de place_aux
63                     continue 2;
64                 end
65             end
66             hasDependableTransitions ← 0;
67             conflict ← Elemento vazio com nome 'conflict';
68             Adiciona atributo 'place' com valor atributo 'id'
69             de place_aux a conflict;
70             foreach transition2 in transition_aux
71                 if atributo 'id' de transition !=
72                 atributo 'id' de transition2
73                     if compareTransitionPriority(transition,transition2) <= 0
74                         Modifica o elemento verify de transition2 para 1;
75                         conflictTransition ← Elemento com
76                         nome 'conflictTransition' e valor
77                         do atributo 'id' de transition2;
78                         Adiciona conflictTransition como elemento
79                         filho a conflict;
80                         hasDependableTransitions ← 1;
```

```
81         end
82     end
83 end
84     if hasDependableTransitions == 1
85         Adiciona conflict como elemento filho
86         a transitionConflicts;
87     end
88 end
89 end
90 end
91 end
92 end
```

Como referido acima a função trata de associar as transições às respetivas contendas. Mas devido à natureza do PNML e à definição de contenda, é necessário encontrar os conflitos de cada transição e descobrir as independências entre transições.

Foi decidido então utilizar uma função de recorrência para encontrar todas as transições pertencentes a uma mesma contenda a partir dos conflitos entre cada uma delas.

Utilizando então a rede de Petri da figura 4.1, a função *processTransition* iria ser chamada para identificar a primeira contenda como podemos ver na linha 13 e 14 da Listagem 4.1, e devido à organização por id's das transições, a primeira a ser processada seria a transição de id 2.

Sendo o objetivo detetar os conflitos de cada transição, a função *processTransition* nas linhas 8 a 16 da Listagem 4.2, procura e guarda em *places_aux* todos os lugares de entrada dessa transição.

Para cada lugar de entrada, procura as suas respetivas transições de saída, ou seja as transições em conflito com a transição a ser processada, neste caso no exemplo prático da Figura 4.1 seriam as transições *tr_3* e *tr_4*. Estas transições encontradas não só pertencem à mesma contenda mas estão em conflito direto com a transição *tr_2*.

Para cada transição é verificado se a mesma já foi processada, através do seu atributo '*contenda*'. No caso de transição não ter sido processada é então definido o atributo '*contenda*' com o valor da contenda a ser processada. É também guardada, no elemento de transição do PNML associado à transição a ser processada, a informação de quais as transições com prioridade inferior à transição a ser processada, e se estas se encontram em conflito direto com a mesma relativamente ao lugar atual (*place_aux* linha 17 Listagem 4.2). Ou seja, para a transição *tr_2*, seria adicionado um elemento filho chamado *conflicts*, que teria elementos filhos chamados *conflict* com um atributo de nome '*place*', de forma a associar a que lugar pertenciam os conflitos diretos, e finalmente cada um desses elementos *conflict* teria elementos filhos chamados *conflictTransition* com o valor de uma transição, para cada uma das transições em conflito direto e com menor prioridade (valor do elemento *priority* maior) que a transição a ser processada.

Durante este processo todas as transições que forem encontradas que o seu disparo seja dependente de outras transições, é modificado o seu atributo *verify* para 1. Ou seja, no caso de existir conflito direto entre duas transições, a de menor prioridade, no caso de

```
<transition id="4" contenda="1" verify="0">
  <name>...
</name>
  <comment>...
</comment>
  <priority>1</priority>
  .
  .
  .
  <conflicts>
    <conflict place="10">
      <conflictTransition>2</conflictTransition>
      <conflictTransition>3</conflictTransition>
    </conflict>
    <conflict place="11">
      <conflictTransition>5</conflictTransition>
      <conflictTransition>6</conflictTransition>
    </conflict>
  </conflicts>
</transition>
```

Figura 4.3: Exemplo da transição de id 4 após ser processada.

escassez de marcas não será verificada.

De forma a facilitar o entendimento, na figura 4.3 encontra-se o exemplo de como ficaria a transição *tr_4* após ser processada tanto em relação ao lugar 10 como ao lugar 11. Neste caso foram guardadas todas as transições em conflito direto com a transição *tr_4*, pois todas elas têm uma prioridade menor ou igual em relação à transição *tr_4*. Como referido, sendo que a *tr_4* não é dependente de nenhuma transição, então manteve o seu atributo *verify* a 0.

Após a informação dos conflitos diretos ser adicionada é então chamada a função recursivamente, mas para as transições em conflito direto com a transição a ser processada.

Quanto ao código presente da linha 52 até à linha 76 da Listagem 4.2, este abrange os casos em que uma transição já foi processada em relação a um certo lugar, mas devido à recorrência foi inicializado o processamento de outra transição, e acabamos por encontrar a mesma transição, mas relativamente a um lugar de entrada diferente, ou seja, que se encontra em conflito direto com novas transições. É o caso da transição *tr_4*, que é processada primeiro em relação ao lugar 10 e de seguida em relação ao lugar 11 sendo que já contém o atributo '*contenda*' com o valor 1.

Desta forma a ordem pela qual as transições da rede de Petri da figura 4.1 seriam processadas seria pela ordem crescente, mas a transição *tr_4* seria processada duas vezes.

Esta informação relativamente à prioridade das transições em conflito direto é necessária para a modificação e melhoramento do código C gerado, em relação a este aspeto.

4.2.3 Função *transitionVectorAttribution*

Para a geração do código C adaptado para a resolução de conflitos eficiente, como iremos ver adiante, é necessário criar vectores de inteiros associados a cada contenda, e que contenham por ordem os id's das transições associadas a essa contenda.

Desta forma é necessário armazenar no PNML, um resumo de quantas contendas existem e de quantas transições estão associadas a cada contenda. É também necessário atribuir uma prioridade específica para o disparo das transições pertencentes a uma contenda, ou seja, embora cada transição tenha a prioridade definida pelo seu elemento *priority* é necessário, para a geração do código C, que transições dentro de uma mesma contenda tenham prioridades todas diferentes, sendo que essa prioridade é calculada e definida pela função *compareTransitionPriority* explicada na subsecção 4.2.5.

Listagem 4.3: Pseudo código da função *transitionVectorAttribution*.

```
1  input : pnml (PNML a ser modificado),
2          contendaNumber (numero total de contendas encontradas),
3  output:
4  begin
5      contendas ← Elemento vazio com nome 'contendas';
6      net ← Elemento 'net' de pnml;
7      while contendaNumber > 0
8          contenda ← Elemento vazio com nome 'contenda';
9          Adiciona atributo 'value' com valor de contendaNumber a contenda;
10         transition_aux ← array vazio;
11         foreach transition in pnml
12             if atributo 'contenda' de transition == contendaNumber
13                 Adiciona transition a transition_aux;
14         end
15     end
16     Ordenar vetor transition_aux por prioridade
17     utilizando compareTransitionPriority;
18     auxiliar ← 0;
19     foreach transition in transition_aux
20         Adiciona atributo 'vectorValue' com valor de auxiliar a transition;
21         auxiliar++;
22     end
23     Adiciona atributo 'totalTransitions' com valor de tamanho
24     de transition_aux a contenda;
25     Adiciona contenda como elemento filho a contendas;
26     contendaNumber--;
27 end
28 Adiciona contendas como elemento filho a net;
29 end
```

A função começa por criar o elemento chamado '*contendas*' que irá ter a informação do resumo das contendas. De seguida calcula para cada contenda quantas transições lhe estão associadas, ordenando-as por prioridade, utilizando a função *usort()* de PHP e utilizando a função *compareTransitionPriority* como função de comparação.

Após a ordenação é então adicionado a cada transição um atributo '*vectorValue*' com o valor da prioridade de disparo que lhe pertence em relação à contenda a que está

associada. Para o caso prático da figura 4.1 a parte do PNML em relação às transições ficaria da forma representada na figura 4.4.

```
<transition id="2" contenda="1" verify="1" vectorValue="3">...
</transition>
<transition id="4" contenda="1" verify="0" vectorValue="0">...
</transition>
<transition id="5" contenda="1" verify="1" vectorValue="2">...
</transition>
<transition id="6" contenda="1" verify="1" vectorValue="4">...
</transition>
<transition id="7" contenda="2" verify="0" vectorValue="0">...
</transition>
<transition id="8" contenda="2" verify="1" vectorValue="1">...
</transition>
<transition id="9" contenda="3" verify="0" vectorValue="0">...
</transition>
```

Figura 4.4: Exemplo da parte das transições do PNML da rede de Petri da figura 4.1.

Finalmente ainda para a contenda seria criado um elemento chamado *contenda* com atributos *'value'* contendo o valor da contenda e *'totalTransitions'* contendo o número total de transições associadas. O elemento *contenda* é adicionado como filho ao elemento *contendas*.

Novamente para o exemplo da figura 4.1 seria adicionado ao elemento *net* do PNML o seguinte elemento *contendas* e os respectivos filhos (figura 4.5).

```
<contendas>
  <contenda value="3" totalTransitions="1"/>
  <contenda value="2" totalTransitions="2"/>
  <contenda value="1" totalTransitions="5"/>
</contendas>
```

Figura 4.5: Elemento *contendas* e respectivos filhos associado à rede de Petri da figura 4.1.

4.2.4 Função *placeContendaAssociation*

Para a integração do código gerado C com a API openMP, é necessário dividir as estruturas de dados do código gerado pelas várias contendas, ou seja, para que cada contenda possa ser processada independentemente sem que haja problemas de acesso à memória é necessário que a própria memória já esteja separada.

Deste modo é necessário associar cada lugar às contendas a que ele se relaciona. Tomando o exemplo da figura 4.1, e escolhendo o lugar 12 para ser processado, é necessário que este contenha a informação que a contenda 2 (*tr_7* e *tr_8*) é uma contenda de saída e as contenda 3 (*tr_9*) e 1 (restantes transições) são contendas de entrada do lugar.

Listagem 4.4: Pseudo código da função *placeContendaAssociation*.

```

1  input : pnml (PNML a ser modificado)
2  output:
3  begin
4      foreach place in pnml
5          foreach arc in pnml
6              if atributo 'target' de arc == atributo 'id' de place
7                  foreach transition in pnml
8                      if atributo 'source' de arc == atributo 'id' de transition
9                          foreach Elemento contenda in place
10                             if atributo 'type' de contenda == "input" &&
11                                 contenda == atributo 'contenda' de transition
12                                     continue 2;
13                             end
14                         end
15                         contenda ← Elemento com nome 'contenda'
16                             e valor do atributo 'contenda' de transition;
17                         Adiciona atributo 'type' com valor "input" a contenda;
18                         Adiciona contenda como elemento filho a place;
19                     end
20                 end
21             elseif atributo 'source' de arc == atributo 'id' de place
22                 foreach transition in pnml
23                     if atributo 'target' de arc == atributo 'id' de transition
24                         foreach Elemento 'contenda' in place
25                             if atributo 'type' de contenda == "output" &&
26                                 contenda == atributo 'contenda' de transition
27                                     continue 2;
28                             end
29                         end
30                         contenda ← Elemento com nome 'contenda'
31                             e valor do atributo 'contenda' de transition;
32                         Adiciona atributo 'type' com valor "output" a contenda;
33                         Adiciona contenda como elemento filho a place;
34                     end
35                 end
36             end
37         end
38     end
39 end

```

Na função da Listagem 4.4 para cada lugar da rede de Petri, procura transições de entrada ou saída desse lugar, e para cada uma encontrada, cria um elemento *contenda* com o valor da *contenda* associada à transição encontrada e com um atributo '*type*' definindo se essa *contenda* é uma *contenda* de entrada ("*input*") ou saída ("*output*") **em relação** ao lugar a ser processado.

Novamente para o lugar 12 o seu elemento ficaria como representado na figura 4.6.

Frisando novamente que o atributo '*type*' designa a associação da *contenda* em relação ao lugar e não o contrário. Ou seja, as *contendas* 1 e 3 são "*input*" em relação ao lugar 12 mas a *contenda* 2 já se associa como "*output*" do lugar 12.

```
<place id="12">
  <name>
  </name>
  <comment>
  </comment>
  .
  .
  .
  <contenda type="input">1</contenda>
  <contenda type="output">2</contenda>
  <contenda type="input">3</contenda>
</place>
```

Figura 4.6: Elemento respetivo ao lugar 12 da rede de Petri da figura 4.1.

4.2.5 Função *compareTransitionPriority*

A função *compareTransitionPriority* é uma função simples utilizada para comparar a prioridade entre duas transições.

Esta prioridade é definida nas transições através do elemento *priority*, e no caso de ambas as transições a serem comparadas possuírem a mesma prioridade, então a ordem pela qual as transições foram criadas é utilizada para decidir a prioridade, esta informação é nos dada pelo atributo *id* das transições, ou seja, a transição com o atributo *id* menor, foi a transição que foi criada primeiro logo será prioritária.

Listagem 4.5: Pseudo código da função *compareTransitionPriority*.

```
1  input : transition1,
2         transition2
3  output: 0 se a prioridade das transições for igual,
4           1 se transition2 for prioritária,
5           -1 se transition1 for prioritária,
6  begin
7    if elemento priority de transition1 == elemento priority de transition2
8      if atributo id de transition1 == atributo id de transition2
9        return 0;
10     elseif atributo id de transition1 < atributo id de transition2
11       return -1;
12     else
13       return 1;
14     end
15  elseif elemento priority de transition1 < elemento priority de transition2
16    return -1;
17  else
18    return 1;
19  end
20 end
```

Como podemos verificar pela Listagem 4.5 a função devolve o valor zero se a prioridade das transições são iguais, embora que este valor, pela natureza do PNML, só é obtido se uma transição for comparada com ela própria.

4.3 Algoritmo do código C gerado

Como a geração de texto utilizando XSLT, no âmbito desta dissertação, é só utilizada para criar o código C, para a explicação do algoritmo gerado, não se torna relevante a estrutura dos ficheiros XSLT mas sim o código C gerado pelos mesmos, e a forma como a estrutura do algoritmo do código C é gerado.

Desta forma toda esta secção irá explicar como fica o código gerado após a utilização dos ficheiros XSLT, e não os próprios ficheiros XSLT.

Como referido na introdução deste capítulo, foram modificados 3 ficheiros XSLT nomeadamente *iopt2c_inc*, *iopt2c_func*, *iopt2c_exec_step* e que por sua vez como explicado na secção 3.2, geram cada um deles um ficheiro de código C específico, respetivamente pela mesma ordem *net_types.h*, *net_functions.c*, *net_exec_step.c*.

4.3.1 Ficheiro *net_types.h*

No ficheiro *net_types.h* poucas modificações foram feitas, visto que este ficheiro serve apenas para declarar estruturas de dados e funções. E as modificações foram feitas de modo a permitir a integração do algoritmo com a API openMP.

Relativamente ao processamento paralelo, existe um problema comum denominado '*race conditions*', este conceito define o problema de um programa realizar duas ou mais operações ao mesmo tempo, criando resultados inesperados. O que no caso do algoritmo desenvolvido, acontece se duas ou mais contendas estiverem a ser processadas por threads diferentes, e seja necessário acesso às estruturas de dados da marcação atual e da marcação a ser adicionada. O que muitas vezes resulta na modificação de variáveis para valores indesejados pois estão a ser adquiridas e modificadas ao mesmo tempo.

Desta forma é então necessário dividir as estruturas de dados utilizadas durante o processamento paralelo de forma a eliminar '*race conditions*' por parte das várias threads a processar as contendas.

São então divididas as seguintes estruturas de dados:

- **NAME_NetMarking:** Esta estrutura de dados é utilizada para instanciar a marcação actual e a marcação a ser adicionada. Devido à natureza do funcionamento do openMP e do algoritmo já implementado foi necessário dividir e duplicar a estrutura de dados principal de marcação em duas estruturas de dados. Uma para a marcação atual da contenda *NAME_avail_NetMarking_contendaX*, contendo os lugares de entrada para a contenda X. A outra nomeada *NAME_add_NetMaeking_contendaX*, contendo os lugares de saída da contenda X. De notar que um lugar pode ser lugar de saída de uma ou mais contendas, mas só pode ser lugar de entrada de uma e uma só contenda, pela definição de contenda;
- **NAME_TransitionFiring:** À semelhança da estrutura de dados anterior, também esta estrutura é dividida, mas como se trata de uma estrutura de dados com a

informação de cada transição, cada uma das estruturas criadas *NAME_TransitionFiring_contendaX* para cada contenda X, só contém a informação das transições que lhe estão associadas.

Para o exemplo da figura 4.1 podemos ver nas seguintes imagens da figura 4.7 como a divisão das estruturas de dados seria gerada.

```
typedef struct {
    unsigned int p_10: 10;
    unsigned int p_11: 10;
    unsigned int p_12: 10;
    unsigned int p_13: 10;
    unsigned int p_14: 10;
    unsigned int padding: 6;
} test3_contenda_NetMarking;

typedef struct {
    unsigned int p_10: 10;
    unsigned int p_11: 10;
    unsigned int padding: 6;
} test3_contenda_avail_NetMarking_contenda1;

typedef struct {
    unsigned int p_12: 10;
    unsigned int p_13: 10;
    unsigned int padding: 6;
} test3_contenda_add_NetMarking_contenda1;
typedef struct {
    unsigned int p_12: 10;
    unsigned int padding: 6;
} test3_contenda_avail_NetMarking_contenda2;

typedef struct {
    unsigned int p_14: 10;
    unsigned int padding: 6;
} test3_contenda_add_NetMarking_contenda2;
typedef struct {
    unsigned int p_13: 10; /
    unsigned int padding: 6;
} test3_contenda_avail_NetMarking_contenda3;

typedef struct {
    unsigned int p_12: 10;
    unsigned int padding: 6;
} test3_contenda_add_NetMarking_contenda3;
```

(a) Exemplo da divisão da estrutura de dados *NAME_NetMarking*.

```
typedef struct {
    unsigned int t_2 : 1;
    unsigned int t_3 : 1;
    unsigned int t_4 : 1;
    unsigned int t_5 : 1;
    unsigned int t_6 : 1;
    unsigned int t_7 : 1;
    unsigned int t_8 : 1;
    unsigned int t_9 : 1;
    unsigned int count: 8;
} test3_contenda_TransitionFiring;

typedef struct {
    unsigned int t_2 : 1;
    unsigned int t_3 : 1;
    unsigned int t_4 : 1;
    unsigned int t_5 : 1;
    unsigned int t_6 : 1;
    unsigned int count: 8;
} test3_contenda_TransitionFiring_contenda1;
typedef struct {
    unsigned int t_7 : 1;
    unsigned int t_8 : 1;
    unsigned int count: 8;
} test3_contenda_TransitionFiring_contenda2;
typedef struct {
    unsigned int t_9 : 1;
    unsigned int count: 8;
} test3_contenda_TransitionFiring_contenda3;
```

(b) Exemplo da divisão da estrutura de dados *NAME_TransitionFiring*.

Figura 4.7: Exemplos da divisão das principais estruturas de dados do algoritmo gerado.

Importante ignorar os valores a que as variáveis inteiras são inicializadas, pois esses valores são substituídos pelas funções que instanciam as estruturas de dados.

Devido ao facto de novas estruturas de dados serem criadas também novas funções para instanciar valores dentro destas estruturas foram criadas. Foram também utilizadas funções para que no final da zona de processamento paralelo, se agrupe a informação das estruturas divididas no início. Neste caso, no ficheiro *net_types.h* foram declaradas 5 novas funções.

4.3.2 Ficheiro *net_functions.c*

Em relação ao ficheiro *net_functions.c*, também muito pouco foi modificado e adicionado. Como referido anteriormente, no ficheiro *net_types.h* foram declaradas 5 funções para atribuir valores às novas estruturas de dados e para voltar a agrupar essa informação no final do passo de execução. Neste caso sendo o ficheiro *net_functions.c*, o ficheiro que define as funções auxiliares a ser usadas no passo de execução, foi então também utilizado para definir as 5 funções referidas, nomeadamente:

- *createInitial_avail_NAME_ContendasMarking*: Esta função, representada na Figura 4.8, recebe uma estrutura de lugares principal (contendo a informação de todos os lugares) e mais X estruturas de lugares de entrada de contendas, sendo X o número de contendas encontradas. E a função trata de atribuir os valores da estrutura principal aos respetivos valores das estruturas representativas de cada contenda, de frisar que nesta função apenas as estruturas de lugares de entrada ficam preenchidas com a informação correta.

```
void createInitial_avail_test3_contenda_ContendasMarking( test3_contenda_NetMarking* init_marking, ...,
test3_contenda_avail_NetMarking_contenda3* cont3_marking)
{
    cont1_marking->p_10 = init_marking->p_10; /* Place pl_10 */
    cont1_marking->p_11 = init_marking->p_11; /* Place pl_11 */
    cont2_marking->p_12 = init_marking->p_12; /* Place pl_12 */
    cont3_marking->p_13 = init_marking->p_13; /* Place pl_13 */
}
```

Figura 4.8: Função *createInitial_avail_NAME_ContendasMarking* associada à rede de Petri da figura 4.1.

- *transfer_NAME_ContendasMarking*: Esta função, representada na Figura 4.9, é utilizada após o passo de execução, ou seja após as várias estruturas de dados associadas a cada contenda tenham sido modificadas, e o objetivo da função é de voltar a agrupar na estrutura principal de lugares a informação relativa aos lugares de entrada das estruturas referentes a cada contenda existente.

```
void transfer_test3_contenda_ContendasMarking( test3_contenda_NetMarking* dest_marking,...,
test3_contenda_avail_NetMarking_contenda3* cont3_marking)
{
    dest_marking->p_10 = cont1_marking->p_10; /* Place pl_10 */
    dest_marking->p_11 = cont1_marking->p_11; /* Place pl_11 */
    dest_marking->p_12 = cont2_marking->p_12; /* Place pl_12 */
    dest_marking->p_13 = cont3_marking->p_13; /* Place pl_13 */
}
```

Figura 4.9: Função *transfer_NAME_ContendasMarking* associada à rede de Petri da figura 4.1.

- *createInitial_add_NAME_ContendasMarking*: À semelhança da função *createInitial_avail_NAME_ContendasMarking*, também esta função, representada na Figura 4.10, atribui os valores da marcação dos lugares, aos respetivos valores das estruturas de

dados de cada uma das contendas, mas no caso desta função, serão preenchidas as estruturas de dados dos lugares de saída.

```
void createInitial_add_test3_contenda_ContendasMarking( test3_contenda_NetMarking* init_marking, ... ,
test3_contenda_add_NetMarking_contenda3* cont3_marking)
{
    cont1_marking->p_12 = init_marking->p_12; /* Place pl_12 */
    cont1_marking->p_13 = init_marking->p_13; /* Place pl_13 */
    cont2_marking->p_14 = init_marking->p_14; /* Place pl_14 */
    cont3_marking->p_12 = init_marking->p_12; /* Place pl_12 */
}
```

Figura 4.10: Função *createInitial_add_NAME_ContendasMarking* associada à rede de Petri da figura 4.1.

- *add_NAME_ContendasMarking*: Relativamente aos lugares de saída das contendas, como referido anteriormente, estes podem ser lugares de saída de uma ou mais contendas, ou seja que a marcação a ser adicionada ao lugar da estrutura de dados principal, irá estar dividida pelas várias estruturas das contendas a que esse lugar está associado como lugar de saída. Desta forma esta função, representada na Figura 4.11, trata de somar todos os valores das várias cópias de cada lugar, para o respetivo valor da estrutura de lugares principal.

```
void add_test3_contenda_ContendasMarking( test3_contenda_NetMarking* dest_marking, ... ,
test3_contenda_add_NetMarking_contenda3* cont3_marking)
{
    dest_marking->p_12 += (cont1_marking->p_12 + cont3_marking->p_12); /* Place pl_12 */
    dest_marking->p_13 += (cont1_marking->p_13); /* Place pl_13 */
    dest_marking->p_14 += (cont2_marking->p_14); /* Place pl_14 */
}
```

Figura 4.11: Função *add_NAME_ContendasMarking* associada à rede de Petri da figura 4.1.

- *transfer_NAME_firingMarking*: Assim como a estrutura de lugares também a estrutura de disparos é dividida, desta forma é necessário voltar a agrupar a informação no final do passo de execução, e à semelhança da função *transfer_NAME_ContendasMarking*, também esta função, representada na Figura 4.12, reagrupa a informação dispersada nas várias estruturas de dados de disparo de transições associadas a cada contenda.

```
extern void transfer_test3_contenda_firingMarking(test3_contenda_TransitionFiring* dest_marking, ... ,
test3_contenda_TransitionFiring_contenda3* cont3_marking)
{
    dest_marking->t_2 = cont1_marking->t_2; /* transition tr_2 */
    dest_marking->t_3 = cont1_marking->t_3; /* transition tr_3 */
    dest_marking->t_4 = cont1_marking->t_4; /* transition tr_4 */
    dest_marking->t_5 = cont1_marking->t_5; /* transition tr_5 */
    dest_marking->t_6 = cont1_marking->t_6; /* transition tr_6 */
    dest_marking->t_7 = cont2_marking->t_7; /* transition tr_7 */
    dest_marking->t_8 = cont2_marking->t_8; /* transition tr_8 */
    dest_marking->t_9 = cont3_marking->t_9; /* transition tr_9 */
}
```

Figura 4.12: Função *transfer_NAME_firingMarking* associada à rede de Petri da figura 4.1.

4.3.3 Ficheiro *net_exec_step.c*

Neste ficheiro foi modificado praticamente todo o algoritmo da função do passo de execução, e as modificações resultantes no algoritmo final, podem ser divididas em dois principais algoritmos.

O primeiro que aproveita a definição de contenda e conflito, de forma a não verificar transições que já sabemos que devido à escassez de marcas não iram disparar.

E o segundo que resumidamente é a integração do algoritmo já existente com a API OpenMP, ou seja, o processamento de cada contenda é entregue a uma thread de modo que várias transições possam ver verificadas e disparadas paralelamente.

Entrando em detalhe em relação ao primeiro algoritmo, é necessário primeiro definir no contexto desta explicação, a diferença entre transições dependentes e independentes.

Desta forma, tendo duas transições em conflito direto e de prioridades diferentes, devido à escassez de marcas apenas uma das transições pode disparar, e sabendo o valor das prioridades podemos afirmar a priori, que a transição de menor prioridade está dependente do não disparo da transição de maior prioridade para poder disparar. Desta forma define-se como dependente a transição de menor prioridade e independente a transição de maior prioridade.

Voltando ao algoritmo 1, este então verifica normalmente todas as transições independentes, e envolve cada uma das transições dependentes com *if's*, que decidem se é necessário verificar a transição ou não.

Esta decisão é feita por base num vetor de inteiros com o valor de 1 ou 0, em que cada posição do vetor está associada a uma transição dependente, pela ordem de prioridades.

Na Listagem 4.6 podemos ver o pseudo-código do algoritmo referido para o caso da figura 4.1.

Listagem 4.6: Pseudo código do algoritmo que aproveita o conceito de conflito e contenda.

```
1  begin Passo Execução
2      int vector1[5] = {1,1,1,1,1};
3
4      Execução de todas as funções de inicialização
5      de estruturas e atribuição de valores às mesmas;
6
7      Execução do if da transição 4;
8      Execução do if da transição 7;
9      Execução do if da transição 9;
10     if vector[0]
11         Execução do if da transição 3;
12     end
13     if vector[1]
14         Execução do if da transição 5;
15     end
16     if vector[2]
17         Execução do if da transição 8;
18     end
```

```

19         if vector[3]
20             Execução do if da transição 2;
21         end
22         if vector[4]
23             Execução do if da transição 6;
24         end
25         Execução das funções de actualização das estruturas de dados
26         assim como de sinais e eventos.
27     end

```

Desta forma sempre que estamos perante um caso em que uma transição de maior prioridade irá impossibilitar o disparo de outras transições em conflito direto, então a transição de maior prioridade irá colocar o valor 0 no *vector1* nas posições das transições às quais vai impossibilitar o disparo.

Listagem 4.7: Exemplo das modificações feitas à verificação e disparo da transição para o algoritmo de conflitos.

```

1     if( t_4_enabled( prev_marking, &avail_marking ) &&
2         t_4_events( &input_events ) &&
3         t_4_guards( prev_marking, inputs, place_out, ev_out ) ) {
4         tfired.t_4 = 1;
5         t_4_remove_marks( &avail_marking );
6         t_4_add_marks( &add_marking );
7         t_4_generate_output_events( &output_events );
8         if(avail_marking.p_10 == 0){
9             vector[3]=0;
10            vector[0]=0;
11        }
12        if(avail_marking.p_11 == 0){
13            vector[1]=0;
14            vector[4]=0;
15        }
16    }

```

Para tornar mais claro na Listagem 4.7 é apresentado o código que na Listagem 4.6 é utilizado na linha 7, e tendo em consideração as prioridades da rede, se estivermos perante o caso em que a transição *tr_4* dispara e os lugares 10 e 11 ficam sem marcas então nenhuma das transições *tr_2*, *tr_3*, *tr_5*, *tr_6*, terá a possibilidade de disparar.

Então se para cada transição dependente, evitarmos verificar se a transição está habilitada e se pode disparar, e se apenas verificarmos se o valor respetivo às transições no *vector1* se encontra a 1 ou 0, iremos poupar algum tempo de processamento.

Em relação ao segundo algoritmo, ou à segunda parte do algoritmo principal, foi integrada a API openMP, utilizando os pragmas *sections* e *section*. O que estes pragmas permitem fazer é dividir o código em secções que serão processadas paralelamente. Neste caso foram divididas as partes do código correspondentes a cada contenda. Ou seja ainda utilizando o algoritmo anterior de verificação de transição a transição, foram divididas as várias transições por contendas, e cada secção utiliza um pragma *section* para indicar a divisão. De notar que neste algoritmo já são utilizadas as estruturas de dados divididas

para evitar '*race conditions*'. Na Listagem 4.8 podemos ficar com uma ideia de como ficaria o código utilizando este algoritmo para o exemplo da rede de Petri da figura 4.1.

Listagem 4.8: Pseudo código do algoritmo integrado com a API openMP.

```
1  begin Passo Execução
2      Execução de todas as funções de inicialização
3      de estruturas e atribuição de valores às mesmas;
4
5      Execução das funções de atribuição de valores para
6      as estruturas de dados respectivas de cada contenda;
7
8      #pragma omp parallel sections
9          #pragma omp section
10         Execução do if da transição 4;
11         Execução do if da transição 3;
12         Execução do if da transição 5;
13         Execução do if da transição 2;
14         Execução do if da transição 6;
15     #end
16     #pragma omp section
17         Execução do if da transição 7;
18         Execução do if da transição 8;
19     #end
20     #pragma omp section
21         Execução do if da transição 9;
22     #end
23 #end
24
25 Execução das funções que voltam a agrupar a informação nas
26 estruturas de dados principais;
27
28 Execução das funções de actualização das estruturas de dados
29 assim como de sinais e eventos.
30 end
```

Desta forma as transições são verificadas em série dentro de um conjunto de transições da mesma contenda, mas no geral são processadas em paralelo.

Em relação aos *ifs* de cada transição neste algoritmo, as únicas modificações aplicam-se ao uso das estruturas de dados associadas à contenda da transição a ser verificada ou disparada, como podemos verificar na Listagem 4.9 utilizando o exemplo da *tr_4* da figura 4.1.

Listagem 4.9: Exemplo das modificações feitas à verificação e disparo da transição para o algoritmo de conflitos.

```
1  if( t_4_enabled( prev_marking, &cont1_avail_marking ) &&
2      t_4_events( &input_events ) &&
3      t_4_guards( prev_marking, inputs, place_out, ev_out ) ) {
4      tfiredContenda1.t_4 = 1;
5      t_4_remove_marks( &cont1_avail_marking );
6      t_4_add_marks( &cont1_add_marking );
7      t_4_generate_output_events( &output_events );
8  }
```

Desta forma o algoritmo final pode ser descrito como a integração dos dois algoritmos referidos anteriormente, sendo este o resultado final pretendido.

O algoritmo resultante integra tanto as divisões das estruturas de dados, como as funções para realizar a divisão da informação e posteriormente agrupamento das mesmas, mas também integra a API OpenMP utilizando os pragmas de *sections*, para a divisão do processamento pelas várias contendadas.

Na Listagem 4.10 temos então o pseudo-código que resume o algoritmo gerado em C pela ferramenta IOPT-Tools, neste caso associada a rede de Petri da figura 4.1.

Listagem 4.10: Pseudo código do algoritmo final C gerado.

```

1  begin Passo Execução
2
3      int vector1[4] = {1,1,1,1};
4      int vector2[1] = {1};
5
6      Execução de todas as funções de inicialização
7      de estruturas e atribuição de valores às mesmas;
8
9      Execução das funções de atribuição de valores para
10     as estruturas de dados respectivas de cada contendada;
11
12     #pragma omp parallel sections
13         #pragma omp section
14             Execução do if da transição 4;
15             if vector1[0]
16                 Execução do if da transição 3;
17             end
18             if vector1[2]
19                 Execução do if da transição 5;
20             end
21             if vector1[3]
22                 Execução do if da transição 2;
23             end
24             if vector1[4]
25                 Execução do if da transição 6;
26             end
27         #end
28         #pragma omp section
29             Execução do if da transição 7;
30             if vector2[4]
31                 Execução do if da transição 8;
32             end
33         #end
34         #pragma omp section
35             Execução do if da transição 9;
36         #end
37     #end
38
39     Execução das funções que voltam a agrupar a informação nas
40     estruturas de dados principais;
41
42     Execução das funções de actualização das estruturas de dados
43     assim como de sinais e eventos.
44 end

```

Como podemos reparar na Listagem 4.10, foi necessário criar um vetor para cada contenda, de forma a executar independentemente, o algoritmo de aproveitamento de resolução de conflitos, para cada conjunto de transições associadas a uma contenda. Desta forma conseguimos não só reduzir tempo de processamento evitando a verificação de transições que não iram disparar, mas também pelo facto de conseguirmos ter várias transições independentes entre si a serem processadas ao mesmo tempo. Para completar também é apresentado na Listagem 4.11 o resultado final do exemplo do if de verificação e disparo da *tr_4* da rede de Petri da figura 4.1.

Listagem 4.11: Exemplo das modificações feitas à verificação e disparo da transição para o algoritmo final.

```
1      if( t_4_enabled( prev_marking, &cont1_avail_marking ) &&
2          t_4_events( &input_events ) &&
3          t_4_guards( prev_marking, inputs, place_out, ev_out ) ) {
4          tfiredContenda1.t_4 = 1;
5          t_4_remove_marks( &cont1_avail_marking );
6          t_4_add_marks( &cont1_add_marking );
7          t_4_generate_output_events( &output_events );
8          if(avail_marking.p_10 == 0){
9              vector1[2]=0;
10             vector1[0]=0;
11         }
12         if(avail_marking.p_11 == 0){
13             vector1[1]=0;
14             vector1[3]=0;
15         }
16     }
```

4.4 Conversor de modelos PIPE para PNML

A plataforma Hippo (Hippo, s.d.), é uma ferramenta online, assim como, as IOPT-Tools, que fornece apoio ao desenvolvimento e pesquisa de redes de Petri, com foco nas redes de Petri do tipo *place-transition nets*. Esta plataforma oferece uma vasta biblioteca de modelos de redes de Petri, assim como funcionalidades para o estudo das mesmas.

Uma das funcionalidades da plataforma Hippo, é a possibilidade de obter os modelos das redes de Petri no formato PIPE XML, que é uma representação PNML do PNTD das *place-transition nets*.

Desta forma, com o intuito de utilizar os modelos da biblioteca da ferramenta HIPPO, assim como contribuir para a ferramenta IOPT-Tools, foi realizado um conversor dentro das IOPT-Tools, que permite o upload de modelos PIPE e a conversão e adaptação dos mesmos para o IOPT PNML.

Inicialmente foi desenvolvido um novo ficheiro *pipe2pnml.php* que inclui apenas uma função *pipe2pnml()*. Esta função trata da conversão de um documento XML para outro

documento XML, especificamente de PIPE XML para PNML IOPT. Recebe os dois ficheiros em formato DOMDocument, e a estrutura da função, está estruturada em três partes, a conversão dos lugares, a conversão das transições e finalmente a conversão dos arcos.

De frisar que o ficheiro PNML que é recebido precisa de conter um modelo em PNML de uma rede de Petri vazia do tipo IOPT.

Na Listagem 4.12 podemos verificar o pseudo-código da função *pipe2pnml()* sendo que os três *for*'s em que a função se pode dividir são explicados no seguimento do documento.

Listagem 4.12: Pseudo-código da função *pipe2pnml()*.

```

1  input : pnmlDoc(contêm um modelo de uma rede IOPT vazia)
2          pipeDoc (contêm o modelo PIPE da rede a ser convertida)
3  output: pnmlDoc (modelo da rede PIPE convertida para IOPT)
4  begin
5      elemsID ← 2;
6      pipeXPath ← new DOMXPath(pipeDoc);
7
8      pipeArcs ← todos os arcos de pipeDoc;
9      pipePlaces ← todos os lugares de pipeDoc;
10     pipeTransitions ← todas as transições de pipeDoc;
11     IDarray ← array de associações vazio;
12
13     pnmlNet ← elemento pnml de pnmlDoc;
14
15     For associado à conversão dos lugares;
16
17     For associado à conversão das transições;
18
19     For associado à conversão dos arcos;
20
21     return pnmlDoc;
22 end

```

Como referido anteriormente a função é dividida em três partes, neste caso três *for*'s. O primeiro *for* é responsável pela conversão dos lugares, assim como guardar no vector *IDarray* uma associação entre o id antigo do lugar no modelo PIPE, ao novo id no modelo PNML, de forma a facilitar a conversão dos arcos no último *for*.

O segundo *for*, à semelhança do primeiro, trata da conversão das transições do documento PIPE para PNML, assim como a associação de id's referida no parágrafo anterior, mas neste caso para as transições.

Finalmente no terceiro e último *for*, é executada a conversão dos arcos, auxiliada pelo *IDarray* com a associação dos id's novos com os do *pipeDoc*.

As conversões são feitas elemento a elemento de forma a construir o PNML correto utilizado pela ferramenta IOPT-Tools.

Por fim foi modificado o ficheiro *index.php* de forma que este inclui-se um novo formulário para a submissão de ficheiros contendo modelos PIPE, como representado na Figura 4.13, assim como as modificações necessárias no código para a conversão do ficheiro PIPE para um ficheiro PNML, utilizando a função *pipe2pnml()*.

IOPT Tools (al_gabriel) Logout

Model actions:

Start new model:

Upload model file: No file selected.

Upload Pipe File:

Upload model pipe file: No file selected.

Important note: IOPT-Tools require the latest Browser versions: Firefox >= 10, Chrome >= 12, Safari, Opera

Figura 4.13: Página index da ferramenta IOPT-Tools com o formulário para ficheiros PIPE.

TESTES DE VALIDAÇÃO E ANÁLISE DE RESULTADOS

5.1 Introdução

Neste capítulo serão apresentados os modelos utilizados para os testes dos algoritmos desenvolvidos, assim como breves explicações do porquê da utilização dos mesmos.

São também apresentados os testes e validações do conteúdo desenvolvido, sendo que é feita uma divisão por algoritmo, ou seja, começa-se por apresentar os testes e validações ao algoritmo de aproveitamento de contendas, e de seguida os testes e validações ao algoritmo de computação paralela, finalmente são apresentados os testes e validações do algoritmo final.

5.2 Modelos

Nesta secção iremos apresentar os modelos utilizados nos testes de validações, e também algumas características dos mesmos. Foram escolhidos modelos de características diferentes podendo assim analisar uma amostra de dados mais completa.

Em todos os modelos foi adicionado um lugar adicional de modo a contabilizar o número de passos de execução, assim como controlar o final das execuções dos modelos.

5.2.1 Modelo *test5*

Nesta subsecção serão apresentados um total de seis modelos.

Os modelos inicialmente criados nomeadamente *test5_2C* e *test5_2C_new*, representados na figura 5.1 e 5.2 respectivamente, são bastante semelhantes, sendo que ambos contêm nove lugares (já contabilizando o lugar de controlo da execução do modelo), doze transições e quatro lugares de entrada.

Em ambos os modelos as transições podem ser divididas em duas contendas, que facilmente podem ser reconhecidas, em que a primeira contém as cinco transições que podemos ver à esquerda em ambas as figuras e a segunda contém sete transições que estão representadas à direita em ambas as figuras também.

A diferença entre estes dois modelos está na marcação inicial, no peso dos arcos e nas ligações dos lugares para as transições.

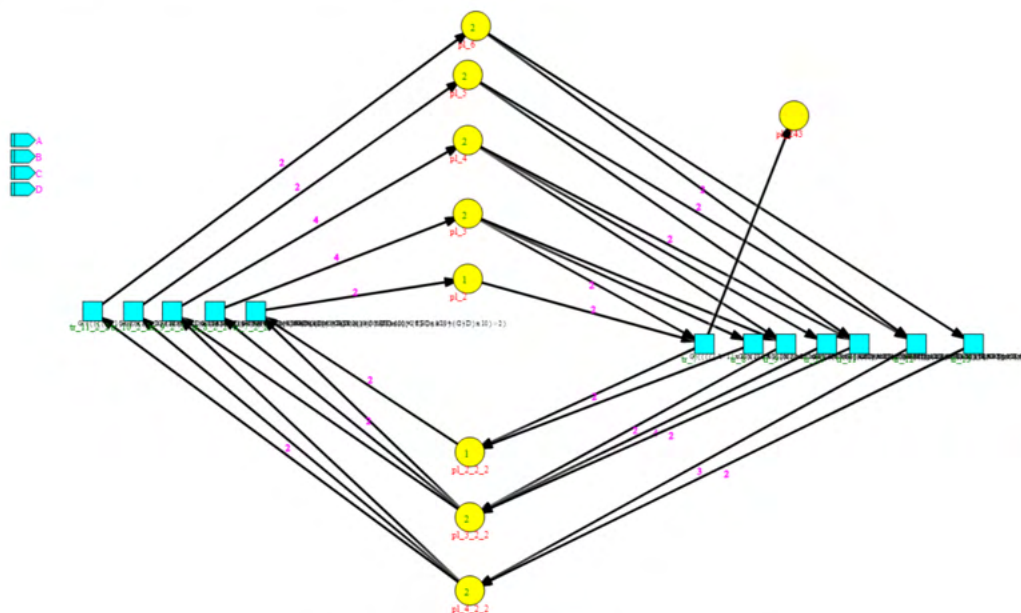


Figura 5.1: Rede de Petri do modelo *test5_2C*.

O que podemos verificar é que no modelo *test5_2C*, a marcação inicial contém valores maiores, ou seja, os lugares têm um número maior de marcas que no modelo *test5_2C_new*. Esta diferença foi criada de maneira a poder verificar resultados em modelos com um número de conflitos maior, ou seja, no modelo *test5_2C_new* devido à escassez de marcas, verificamos um número maior de conflitos directos em relação ao modelo *test5_2C*.

Em relação às diferenças no peso dos arcos e nas ligações, estas são necessárias para conseguir obter modelos com redes de Petri vivas. Deste modo podemos verificar que no modelo *test5_2C* em comparação com o modelo *test5_2C_new*, contem um menor número de ligações da contenda da esquerda para o conjunto de lugares da parte superior e diferenças nos pesos dos arcos, estas diferenças são resultantes da necessidade da rede de Petri ser viva.

Desta forma obtemos um modelo *test5_2C* com menos conflitos mas não seguro, ou seja, em que a marcação nos lugares atinge valores maiores que um. Ao contrário do modelo *test5_2C_new* que devido à escassez de marcas e ao facto de todos os lugares excepto o lugar *p_243* (lugar de controlo da execução do modelo) serem seguros, obtemos muitos mais conflitos no mesmo passo de execução.

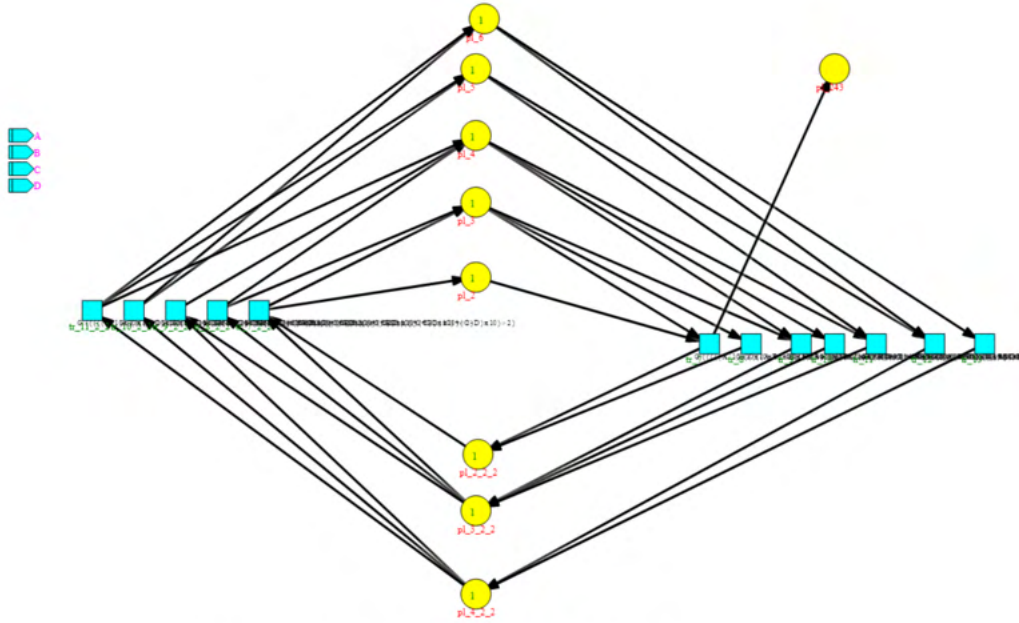
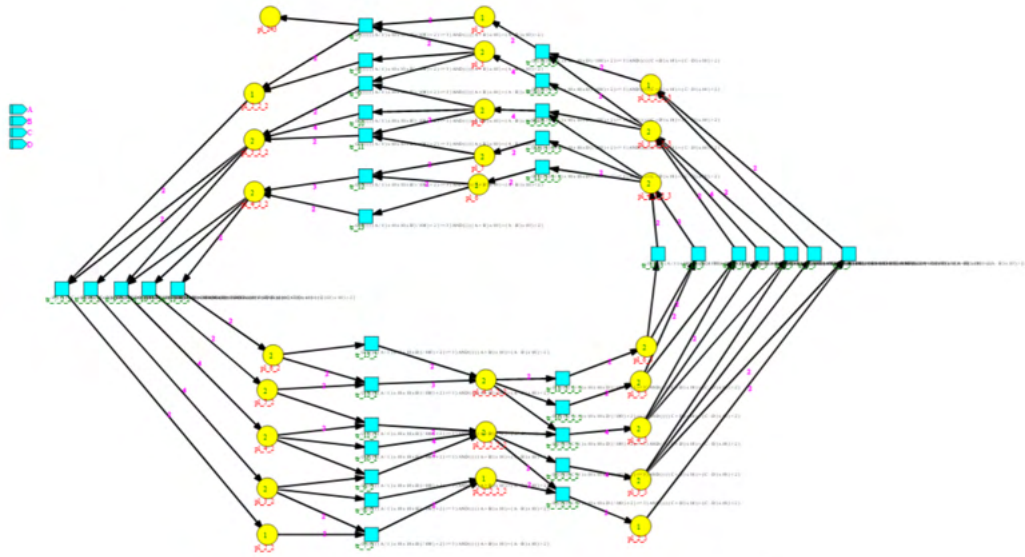
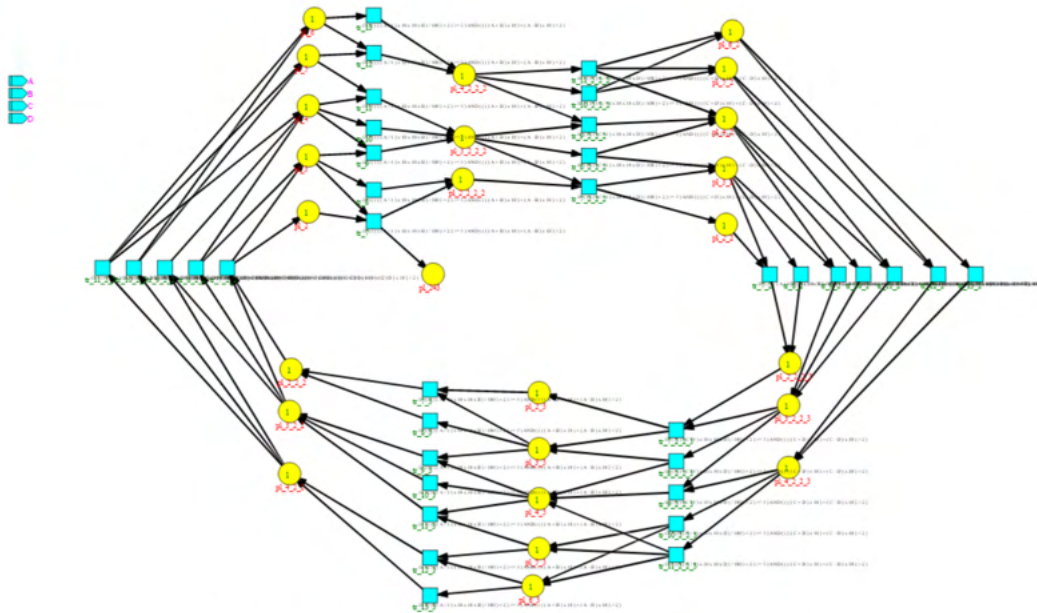


Figura 5.2: Rede de Petri do modelo *test5_2C_new*.

Estes dois modelos foram construídos de modo a obter resultados de uma amostra de modelos mais variados e conseguir uma melhor noção do impacto dos vários algoritmos em situações diferentes.

Sendo que a diferença de conflitos irá resultar em diferenças apenas no algoritmo de aproveitamento de conflitos e no algoritmo final, foi necessário construir modelos que nos permitissem verificar o impacto do algoritmo de processamento paralelo. Desta forma foram criados quatro modelos adicionais que utilizam os modelos anteriores como base ,ou seja, são utilizadas e duplicadas as transições e os lugares de modo a que o comportamento seja o mesmo mas que existam mais contendras.

Assim foram criados os modelos *test5_6C* e *test5_6C_new* representados nas figuras 5.3 e 5.4 respectivamente, que resultam da expansão e clonagem dos modelos anteriores *test5_2C* e *test5_2C_new*, de modo a obter modelos com seis contendras.

Figura 5.3: Rede de Petri do modelo *test5_6C*.Figura 5.4: Rede de Petri do modelo *test5_6C_new*.

O comportamento da rede em ambos os modelos *test5_6C* e *test5_6C_new* é similar à das suas cópias de duas contendas, mas neste caso contém cada um dos novos modelos 6 contendas.

Finalmente foram também criados, de maneira a obter uma amostra de modelos mais variada, mais dois modelos *test5_10C* e *test5_10C_new* representados nas figuras 5.5 e 5.6, que à semelhança dos modelos anteriores são uma expansão e clonagem dos dois

primeiros modelos *test5_2C* e *test5_2C_new*, mas com dez contendas.

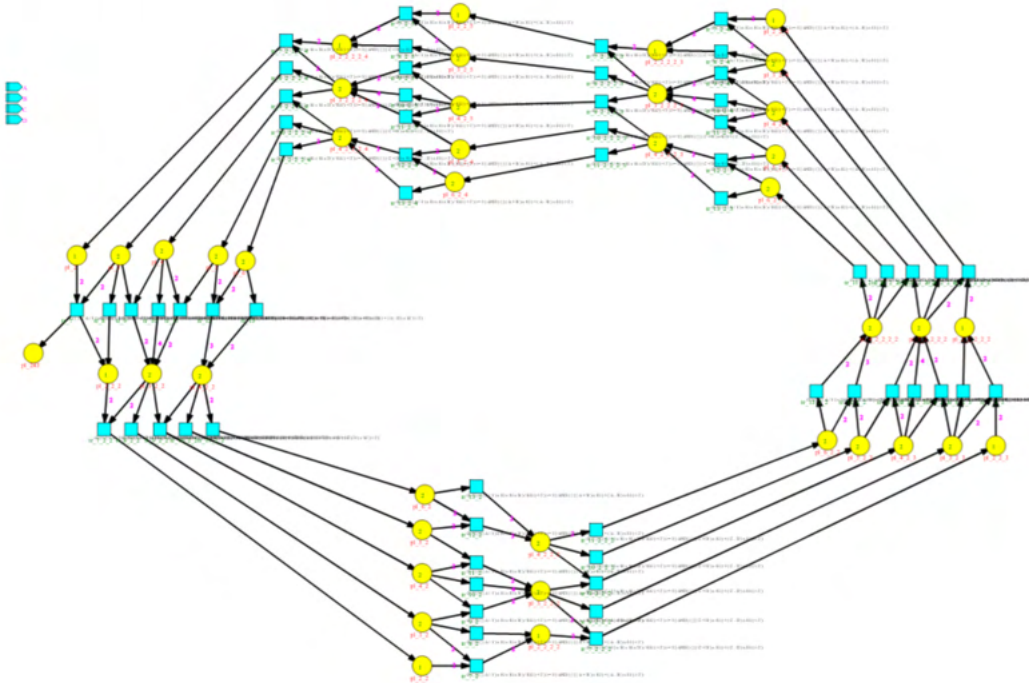


Figura 5.5: Rede de Petri do modelo *test5_10C*.

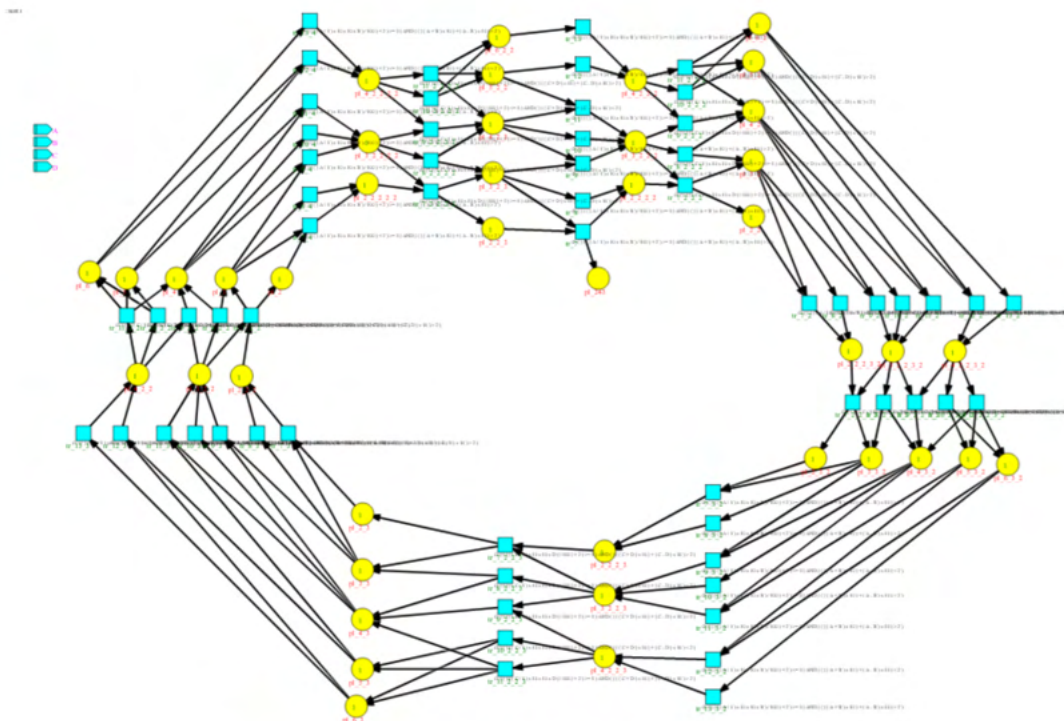


Figura 5.6: Rede de Petri do modelo *test5_10C_new*.

5.2.2 Modelo *TIE10_automation*

O modelo *TIE10_automation* representado na figura 5.7 foi fornecido pela plataforma IOPT-Tools (IOPT-Tools, s.d.), e modela um sistema de quatro tapetes rolantes e quatro robôs para movimento de cargas. O modelo é composto por trinta e dois lugares, sendo que oito têm marcas definindo assim a marcação inicial. Contêm vinte e cinco transições, nove sinais de entrada e oito sinais de saída. Os lugares de entrada representam a entrada e saída dos tapetes, e os sinais de saída representam a utilização dos tapetes e dos robôs que estão a ser utilizados.

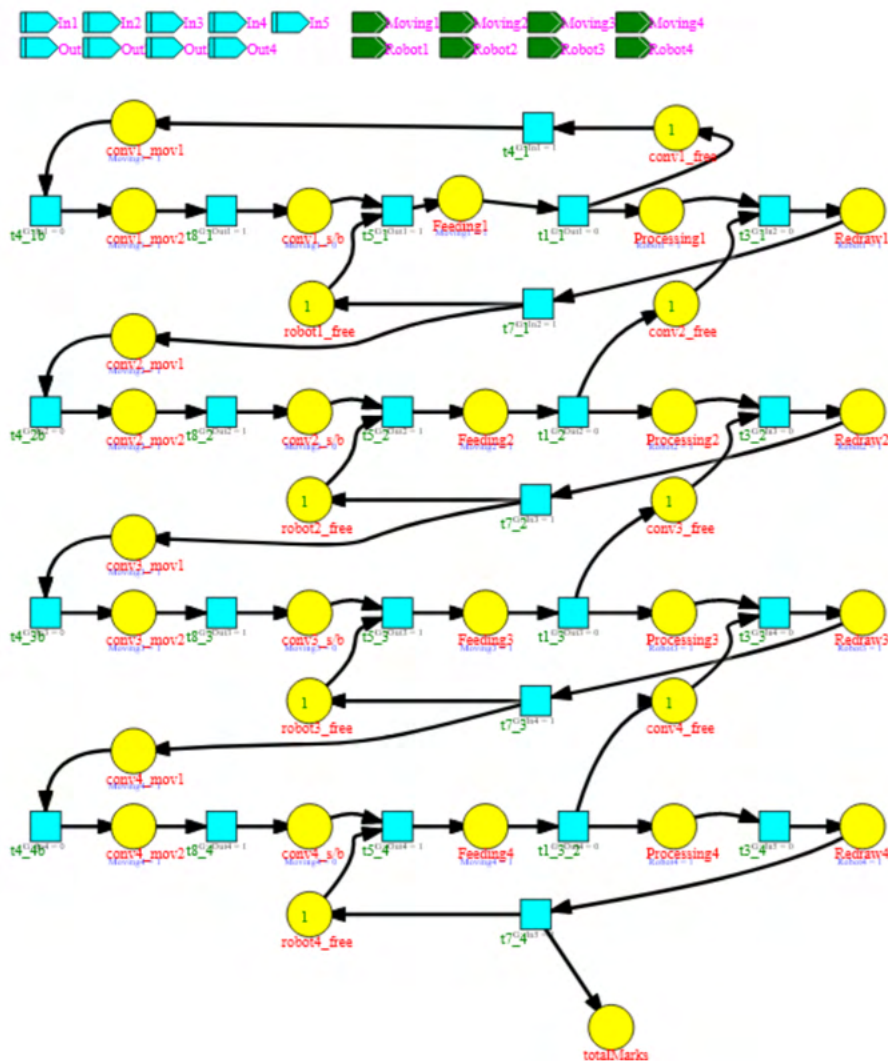


Figura 5.7: Rede de Petri do modelo *TIE10_automation*.

Como referido anteriormente, foi adicionado um lugar ao modelo original *totalMarks*, de modo a que fosse possível controlar o final da execução. A execução termina quando o lugar *totalMarks* apresenta um número específico de marcas, sendo que esse valor depende dos testes realizados. Pela natureza deste modelo é importante referir que a cada quatro passos de execução a marcação do lugar *totalMarks* aumenta em uma marca.

Este modelo contém um total de vinte e cinco contendas, sendo que nenhuma contem conflitos. Este modelo foi escolhido, pois modela um problema real e devido ao facto de não ter conflitos podemos obter resultados específicos em relação à utilização do OpenMP com um modelos de bastantes contendas.

Com este modelo espera-se conseguir dividir os resultados de forma a verificar as diferenças dos resultados entre os vários algoritmos, assim como perceber de que forma o OpenMP afecta modelos com bastantes contendas e contendas pequenas e sem conflitos.

5.2.3 Modelo *girault8*

O modelo *girault8* representado na figura 5.8 foi obtido a partir da plataforma Hippo (Hippo, s.d.), e utilizando o conversor desenvolvido na subsecção 4.4, foi possível importar o modelo para a ferramenta IOPT-Tools (IOPT-Tools, s.d.). Este modelo contém um total de oito lugares e oito transições. Assim como todos os modelos da plataforma Hippo, este modelo não contém sinais de entrada ou de saída bem como eventos de entrada e saída. No modelo já convertido nas IOPT-Tools, representado na figura 5.9 foi adicionado um lugar, de forma a controlar o final da execução do mesmo à semelhança dos modelos anteriores.

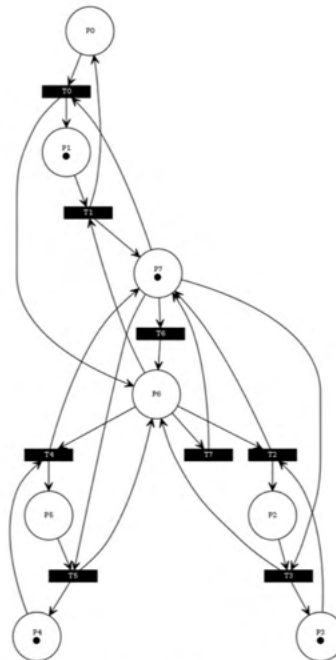


Figura 5.8: Rede de Petri do modelo *girault8* na plataforma Hippo.

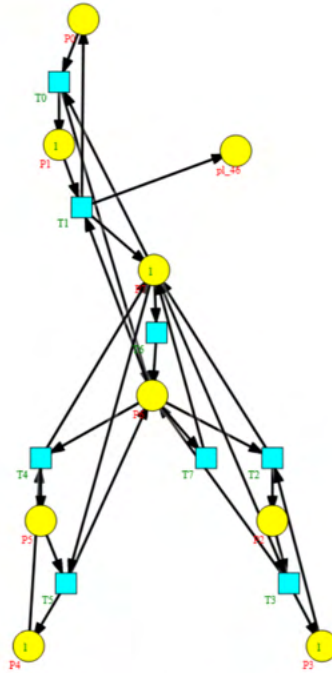


Figura 5.9: Rede de Petri do modelo *girault8* na plataforma IOPT-Tools.

As transições deste modelo podem ser divididas em duas contendas, sendo que a primeira contém as transições $T0$, $T3$, $T5$, $T6$, e a segunda contenda contém as transições $T1$, $T2$, $T4$.

Este modelo foi escolhido maioritariamente para demonstrar o conversor de modelos PIPE para PNML, e devido ao facto ser um modelo bastante genérico em podemos verificar a eficiência dos algoritmos desenvolvidos, num ambiente com um número de conflitos mais reduzido.

5.3 Testes de validação e análise de resultados

Para a obtenção de resultados a partir dos testes foram utilizadas várias plataformas, sendo que no caso dos algoritmos de processamento paralelo é necessário que as plataformas permitam a utilização de OpenMP, o que não se verifica em todas as plataformas.

Desta forma para o primeiro algoritmo foram utilizadas todas as plataformas, pois o algoritmo é executado em série. As plataformas utilizadas para o teste dos vários algoritmos são apresentadas na tabela 5.1 seguinte, assim como as suas características.

Tabela 5.1: Tabela de plataformas utilizadas para teste dos algoritmos e respectivas características.

Plataforma	CPU	Frequência	Cores	Sistema operativo
Arduino Mega	ATmega2560	16MHz	1	—
Arduino Due	AT91SAM3X8E	84 MHz	1	—
Zedboard	ARM Cortex-A9	33.3 MHz	8	—
P8B-E/4L	E3-1240 V2	1606.898 MHz	8	Fedora 19(Linux)
Asus ROG Laptop	i7-7700HQ	2.8 GHz	8	Windows 10

Da Tabela 5.1 todas as plataformas são utilizadas para testar o algoritmo de contendas, mas para o algoritmo de computação paralela, assim como para o algoritmo final, apenas são utilizadas as plataformas *P8B-E/4L* e *Asus ROG Laptop*, pois são as únicas que utilizamos para incorporar a tecnologia de OpenMP.

Todos os testes executados baseiam-se na execução dos modelos numa plataforma até um lugar específico atingir um número certo de marcas. O tempo total de execução é calculado como uma média de 900 execuções excepto no caso dos Arduinos e da ZedBoard.

5.3.1 Testes e análise do algoritmo de contendas

Para perceber melhor os resultados dos testes do algoritmo de contendas, é apresentado na Figura 5.10 o gráfico que representa o número de transições ignoradas, ou seja, o número de transições que devido à escassez de marcas não foram verificadas de modo a poupar tempo, até que em cada modelo o respectivo lugar de controlo de execução chegue às 300 marcas, ou seja, uma execução completa dos modelos.

Este cálculo foi feito utilizando uma variável auxiliar global, e utilizando o exemplo da listagem 4.7, em ambos os *if's* da linha 8 e 12 foi adicionada uma linha que incrementava a variável global de transições ignoradas em dois, pois em ambos os *if's* irão fazer com que duas transições sejam ignoradas.

Outro aspecto em ter em consideração para a validação dos resultados obtidos no seguimento deste documento, é o número total de transições de cada modelo, que pode ser visto no gráfico da figura 5.11.

Deste modo começou-se por testar o algoritmo para os vários modelos na plataforma *Asus ROG Laptop* utilizando o Visual Studio 2019 e obteve-se os seguintes resultados apresentados no gráfico da Figura 5.12. Este gráfico compara para cada modelo, os tempos de execução do algoritmo de contendas desenvolvido e o algoritmo actualmente implementado no servidor da plataforma IOPT-Tools, até que para cada modelo se atinga no lugar de controlo de execução as 300 marcas. Os tempos de execução foram calculados a partir de uma média de 900 execuções da função principal *main()* sendo a execução é contabilizada na totalidade.

Com o algoritmo desenvolvido obtemos melhores tempos de execução, mas visto que

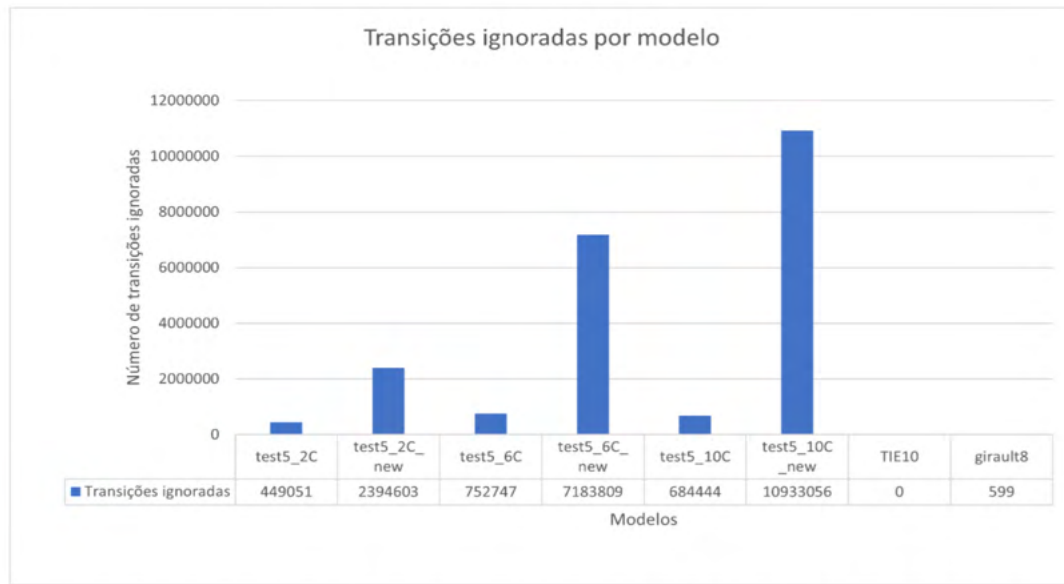


Figura 5.10: Gráfico de Transições ignoradas por modelo até ao final de uma execução completa.

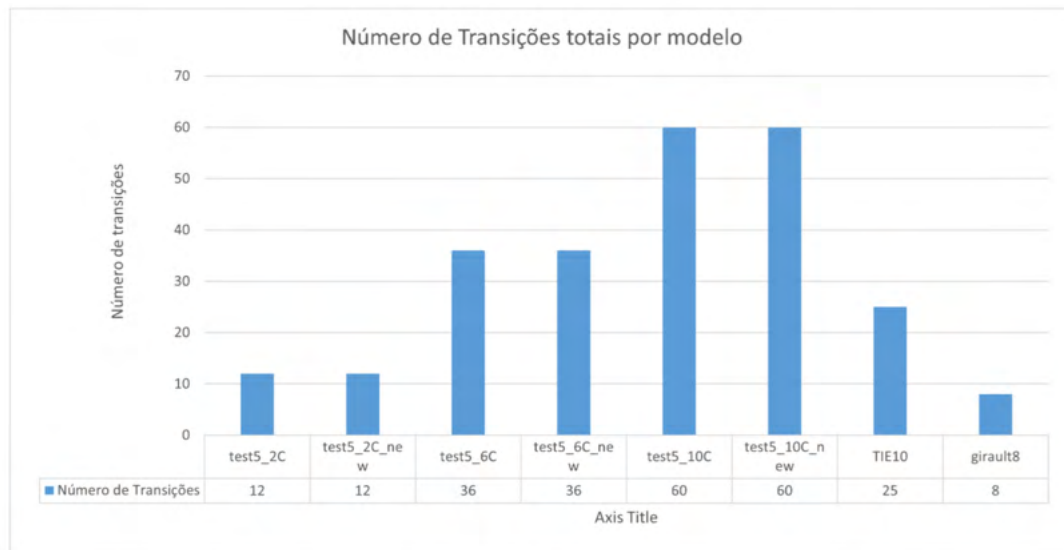


Figura 5.11: Gráfico de Transições totais por modelo.

a plataforma utilizada utiliza Windows, e muitos outros processos corriam ao mesmo tempo que a execução dos modelos em Visual Studio 2019, os resultados são pouco consistentes.

A pouca consistência dos resultados pode ser verificada pois este algoritmo aproveita-se das transições ignoradas, e pelo gráfico da figura 5.10 o modelo *TIE10* não deveria ter diferenças de tempo de execução do algoritmo original para o algoritmo desenvolvido, o que pelo gráfico 5.12 se verifica.

Desta forma foi também testado o mesmo algoritmo na plataforma *P8B-E/4L*, ou seja,

5.3. TESTES DE VALIDAÇÃO E ANÁLISE DE RESULTADOS

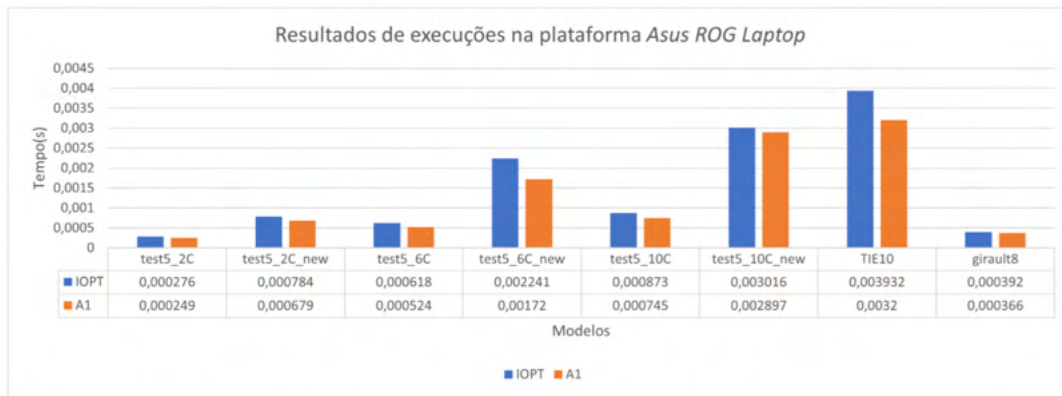


Figura 5.12: Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendas na plataforma *Asus ROG Laptop*.

no servidor da plataforma IOPT-Tools e embora este também tenha outros processos a decorrer são muito menos, e o sistema operativo é *Fedora*. Os resultados no gráfico da figura 5.13, assim como no gráfico anterior são apresentados como comparação do algoritmo desenvolvido e do implementado na ferramenta IOPT-Tools, e com as mesmas regras de execução e de obtenção de tempos que o gráfico anterior.

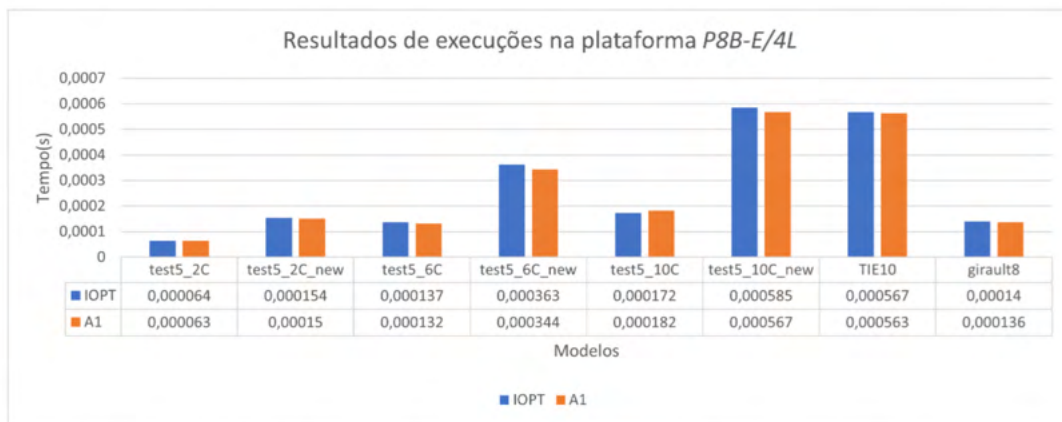


Figura 5.13: Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendas na plataforma *P8B-E/4L*.

Assim como no gráfico da figura 5.12, também o gráfico da figura 5.13 apresenta resultados de melhor desempenho do algoritmo desenvolvido, e embora apresente uma melhor consistência de resultados, ainda se verifica que no modelo *TIE10* existe uma diferença de tempos de execução que como explicado anteriormente não deveria existir. Esta diferença é resultado da gestão das threads pelos sistemas operativos e pelo facto de existirem outros processos a decorrer.

De forma a eliminar a inconsistência nos resultados foram testados os algoritmos em plataformas, que apenas executassem os algoritmos e os algoritmos apenas.

No seguinte gráfico da figura 5.14 é apresentado os resultados do teste efectuado na

plataforma *Zedboard*, em que são comparados os tempos de execução do algoritmo desenvolvido e do algoritmo já implementado, e à semelhança dos testes anteriores os tempos de execução são calculados na função *main()*, mas ao contrário dos testes anteriores apenas é tomado em conta uma execução ao invés da média de 900 execuções. Esta mudança resulta do facto que os modelos são corridos em *Bare Metal*, ou seja, o processador da plataforma apenas está a correr o modelo e nada mais, sem a necessidade de um sistema operativo.

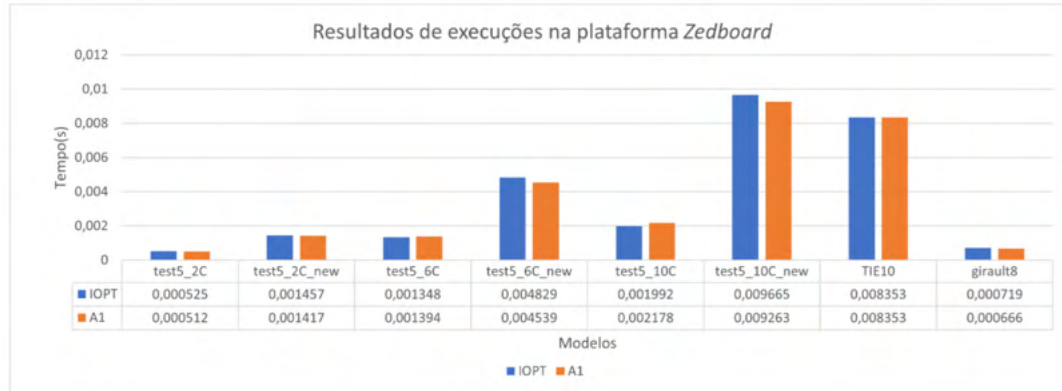


Figura 5.14: Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendas na plataforma *Zedboard*.

Na plataforma *Zedboard* já conseguimos notar bastantes diferenças em relação ao resultados obtidos anteriormente. Nomeadamente em relação ao modelo *TIE10* já verificamos uma consistência nos tempos de execução de ambos os algoritmos, e ao contrário dos resultados anteriores verifica-se que em alguns dos modelos não obtemos melhores resultados com o algoritmo desenvolvido.

A partir destes resultados verificamos que modelos com maior número de transições ignoradas obtêm tempos de execução menores com o algoritmo desenvolvido, e que o algoritmo desenvolvido ainda permite benefícios a quando da dimensão dos modelos, ou seja, se os modelos forem de pequena dimensão embora que com poucas transições ignoradas ainda se verificam resultados positivos.

Ainda relativamente ao algoritmo de contendas desenvolvido foram realizados testes na plataforma *Arduino Mega* representado na figura 5.15.

Como podemos verificar pelos resultado obtidos na Figura 5.15, todos os modelos na plataforma *Arduino Mega* apresentaram resultados negativos, pois a diferença entre o tempo do algoritmo já implementado e o algoritmo de aproveitamentos de contendas é positiva para trezentas, quinhentas e mil execuções, o que comprova que o factor que nos proporciona estes resultados não é o *overhead* criado pelas adaptações no código para funcionar na plataforma *Arduino Mega*.

De modo a permitir perceber o porque da diferença nos resultados obtidos entre a plataforma *Arduino Mega* e na plataforma *Zedboard* foram realizados testes na plataforma

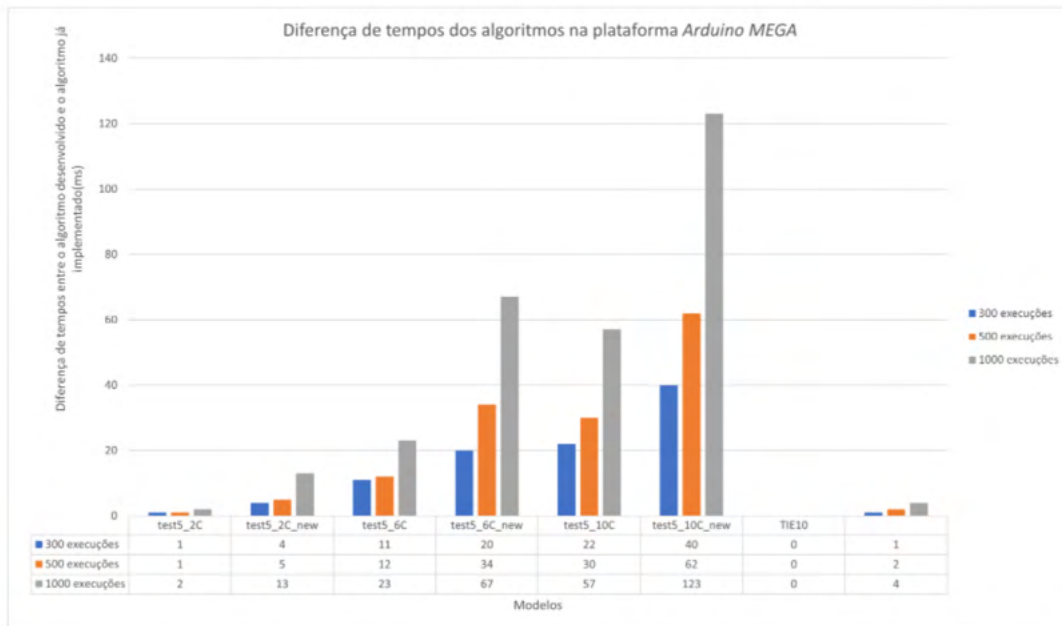


Figura 5.15: Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendas na plataforma *Arduino Mega*.

Arduino Due representados no gráfico da figura 5.16. Este teste na plataforma *Arduino DUE* permitiu perceber se os resultados obtidos na figura 5.15 são consequência da compilação pelo ambiente *Arduino IDE* ou se são consequência da baixa complexidade e qualidade da arquitetura do CPU *ATmega2560*.

Como podemos perceber pelo gráfico da figura 5.16, o comportamento dos resultados é diferente do comportamento dos resultados obtidos no gráfico da figura 5.15, podendo assim afirmar que a diferença dos resultados entre a plataforma *Arduino Mega* e *Zedboard* não é consequência da compilação do código pela ferramenta *Arduino IDE*.

E ainda no gráfico da figura 5.16, podemos verificar que existem modelos em que a diferença entre os tempos do algoritmo já implementado e o algoritmo de contendas é nulo e existe um caso (modelo *girault8*) em que a diferença é negativa, considerada um resultado positivo. Destes casos podemos retirar então a conclusão semelhante à das plataformas inicialmente utilizadas em que se obtida melhores resultados em modelos que o valor das transições ignoradas era elevado em relação ao tamanho do modelo.

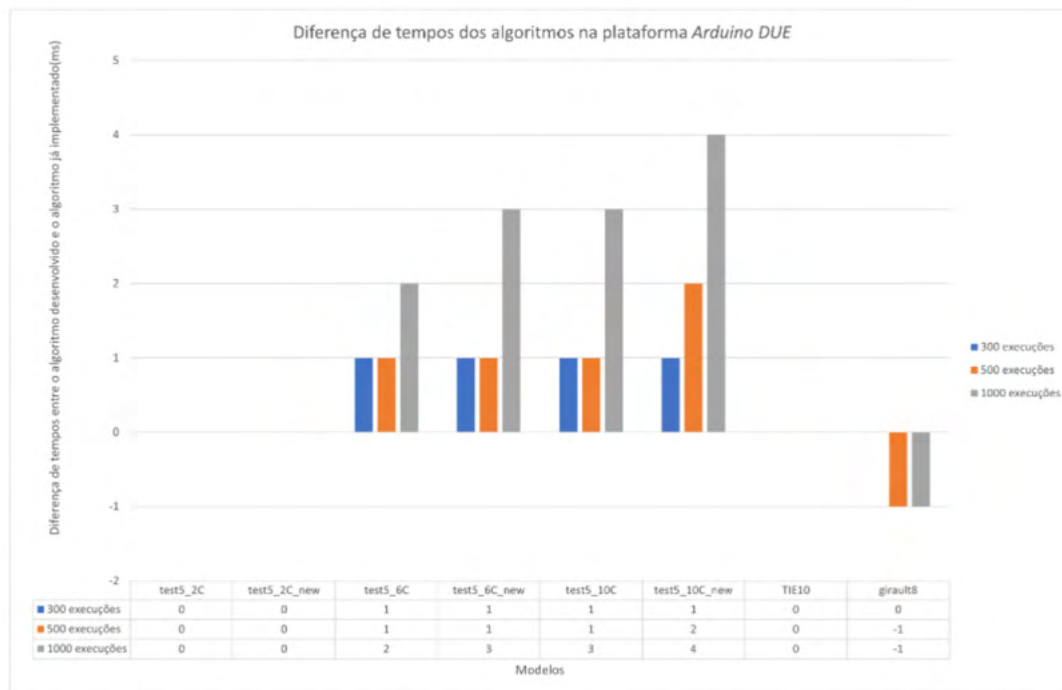


Figura 5.16: Gráfico de resultados obtidos da execução dos modelos com o algoritmo de contendas na plataforma *Arduino DUE*.

5.3.2 Testes e análise do algoritmo de computação paralela

Em relação ao algoritmo e computação paralela, devido à necessidade de utilizar a API OpenMP, apenas foram utilizadas as plataformas *Asus ROG Laptop* e *P8B-E/4L*, e embora em ambas tenhamos processos a correr além das execuções dos modelos verificou-se que assim como para o algoritmo de contendas, a plataforma *P8B-E/4L* apresentou resultados mais consistentes que a plataforma *Asus ROG Laptop*.

De modo a conseguir perceber o impacto da utilização do algoritmo de computação paralela foram realizados dois tipos de teste de tempo de execução em ambas as plataformas. Um primeiro teste realizado, à semelhança dos testes para o algoritmo de contendas, baseia-se na obtenção dos tempos médios de 900 execuções até que cada modelo atinga as 300 marcas nos respectivos lugares de controlo de execução. Este teste foi realizado para cada modelo, e comparando entre o algoritmo em série já implementado nas IOPT-Tools e o algoritmo de computação paralela utilizando desde uma a oito threads de processamento. Os resultados deste teste para as plataformas *Asus ROG Laptop* e *P8B-E/4L* estão representados nos gráficos das figuras 5.17 e 5.18 respectivamente.

Em ambas as plataforma podemos verificar que o algoritmo de computação paralela em nenhum modelo apresentou resultados positivos, e verificou-se que os tempos de execução aumentam com o aumento de threads para a computação. Este aumento pode ser dividido em duas partes sendo a primeira, o aumento de computação consequente da

5.3. TESTES DE VALIDAÇÃO E ANÁLISE DE RESULTADOS

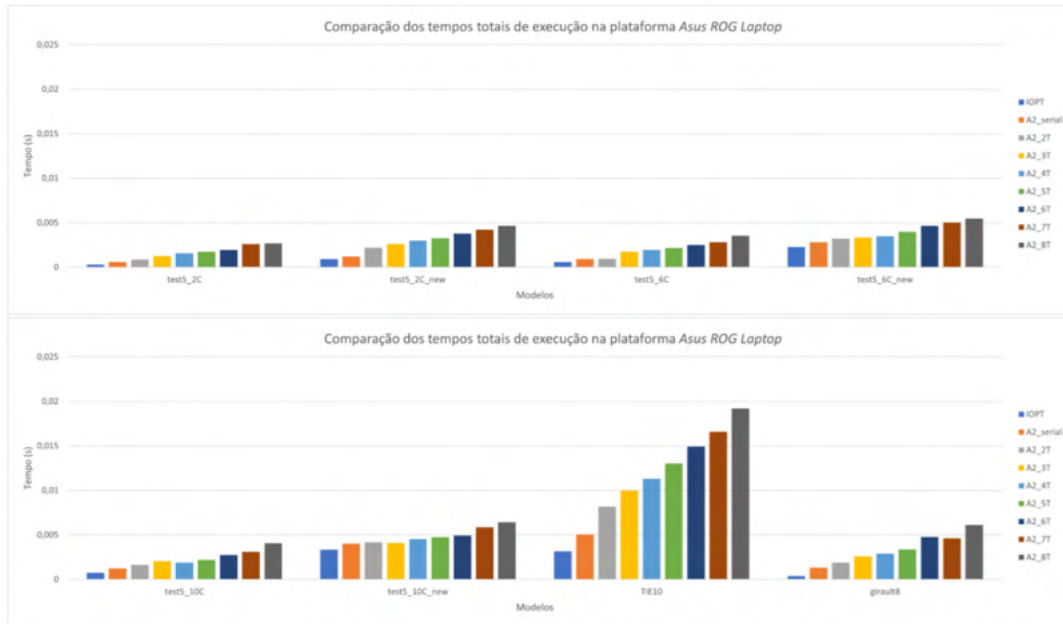


Figura 5.17: Gráfico de resultados obtidos da execução dos modelos com o algoritmo de computação paralela na plataforma *Asus ROG Laptop*.

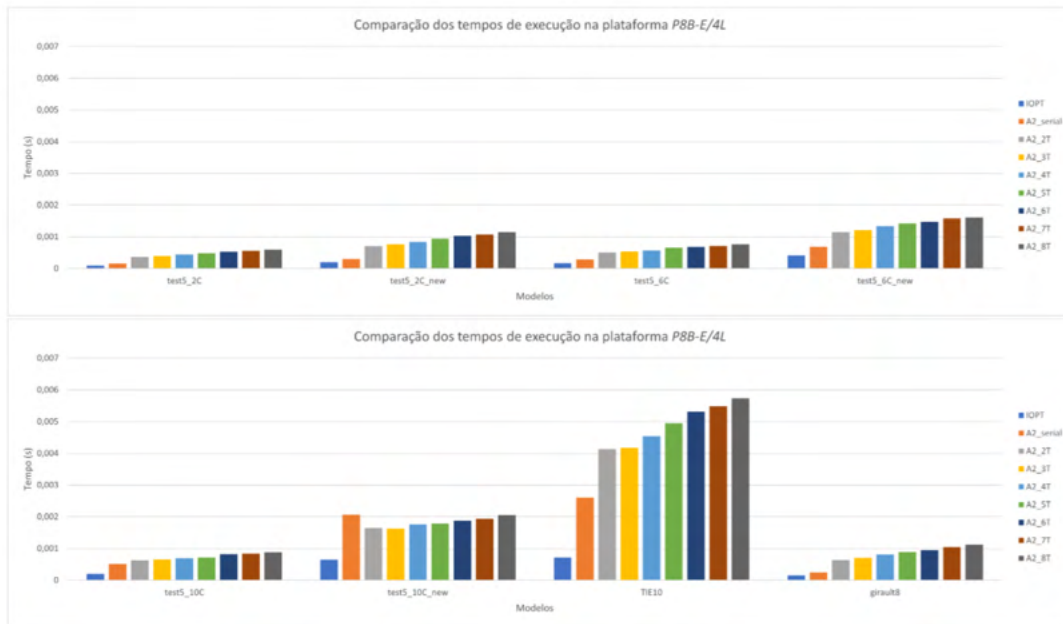


Figura 5.18: Gráfico de resultados obtidos da execução dos modelos com o algoritmo de computação paralela na plataforma *P8B-E/4L*.

divisão das estruturas de memória e a junção das mesmas, respectivamente no início e no fim da execução do passo de execução, esta divisão e reagrupamento dos dados em diferentes estruturas de memória é necessária de modo a evitar *race conditions* explicadas anteriormente. A segunda parte do aumento da computação é resultante do *overhead*

criado pela divisão da thread principal em threads de processamento paralelo e a junção das mesmas no final da secção paralela.

De forma a perceber o quanto as duas partes influenciavam no aumento da computação, como referido anteriormente, foi realizado um segundo teste a todos os modelos com o algoritmo de computação, mas neste teste foi retirado o tempo médio de 900 execuções do processamento das transições, ou seja, recorrendo à Listagem 4.8 foi calculado em cada passo de execução o tempo de execução da linha 7 à linha 24, desta forma isola-se o tempo de execução do *fork-and-join* do OpenMP, permitindo perceber o impacto deste no algoritmo do OpenMP. À semelhança, no algoritmo original da plataforma IOPT-Tools também foi apenas calculado o tempo de processamento das transições.

Nos gráficos das figuras 5.19 e 5.20 temos então representado os resultados obtidos do teste de tempos de processamento das transições nas plataformas *Asus ROG Laptop* e *P8B-E/4L* respectivamente, e são comparados os resultados para cada modelo entre o algoritmo já implementado (IOPT) e o algoritmo de computação paralela utilizando desde uma a oito threads de processamento.

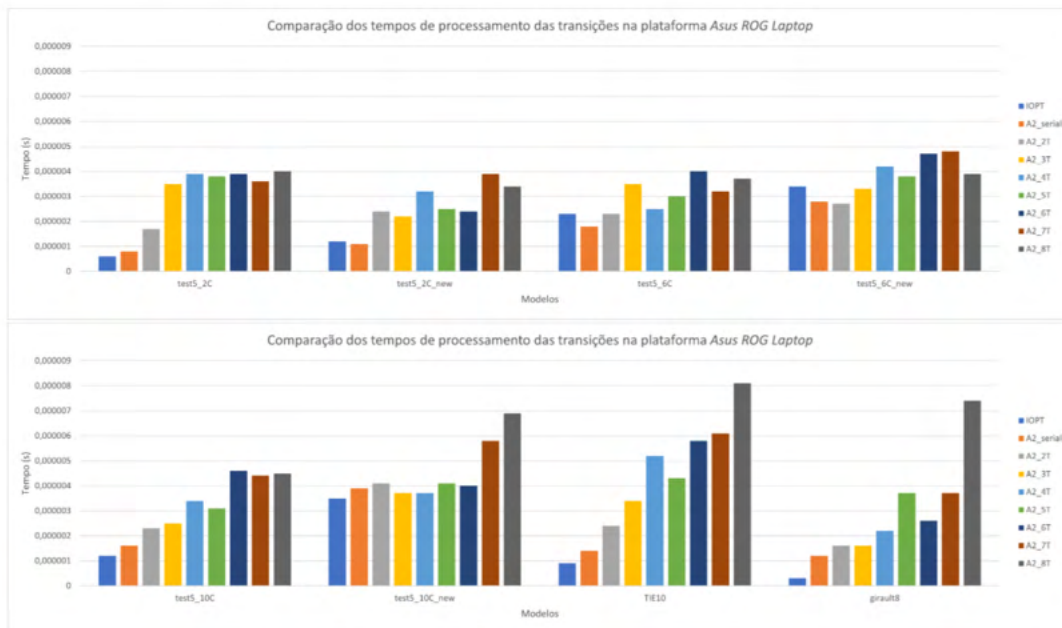


Figura 5.19: Gráfico de resultados obtidos do processamento de transições dos modelos com o algoritmo de computação paralela na plataforma *Asus ROG Laptop*.

Como podemos verificar pelos resultados obtidos, não se verificam tempos de processamento das transições menores em nenhuma das plataformas, e também se verifica um aumento do tempo de processamento das transições ao aumentar o número de threads de processamento, este comportamento é resultante do *overhead* criado com cada nova thread de processamento que precisa ser criada e no final reagrupada com a thread principal, e devido ao processamento a ser realizado dentro de cada *pragma section* não ser

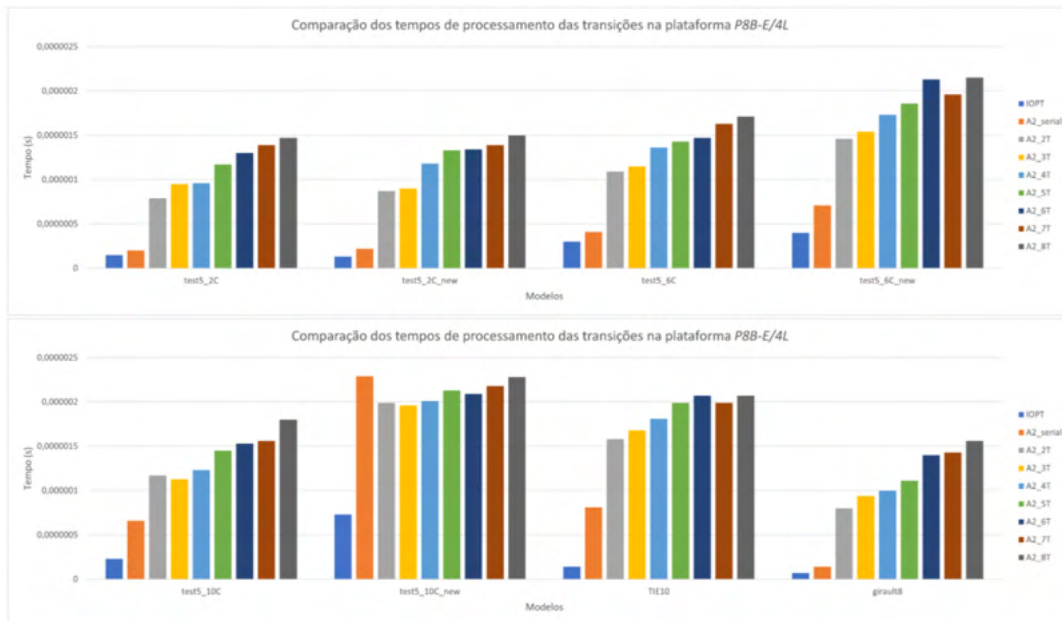


Figura 5.20: Gráfico de resultados obtidos do processamento de transições dos modelos com o algoritmo de computação paralela na plataforma *P8B-E/4L*.

muito pesado computacionalmente, não se retira benefícios do uso de várias threads.

5.3.3 Testes e análise do algoritmo final

Como consequência de ser a junção dos dois algoritmos referidos anteriormente, o algoritmo final assim como o algoritmo de computação paralela foi testado nas plataformas *Asus ROG Laptop* e *P8B-E/4L*, e assim como o algoritmo de computação paralela foram realizados os mesmos dois testes, ou seja, o teste para cada modelo da comparação dos tempos em média de 900 execuções até o lugar de controlo de execução obter as 300 marcas, e o teste do tempo de processamento das transições. Utilizando a listagem 4.10 como exemplo foi calculado o tempo desde a linha 11 à linha 38.

Nos gráficos das figuras 5.21 e 5.22 estão representados os resultados obtidos do primeiro teste realizado em ambas as plataformas *Asus ROG Laptop* e *P8B-E/4L* respectivamente.

Pelos gráficos das figuras 5.21 e 5.22 verifica-se que o algoritmo final demonstra em termos dos tempos de execução um comportamento parecido aos resultados obtidos nos gráficos do algoritmo de computação paralela para o tempo de execução (figuras 5.17 e 5.18), este comportamento idêntico é esperado pois como referido anteriormente o algoritmo final é a junção dos dois algoritmos e embora tenhamos resultados positivos no algoritmo de conteúdos, estes não são suficientes para cobrir o tempo de computação gerado pelos *overheads* do algoritmos de computação paralela, tanto os *overheads* criado pela divisão e junção das estruturas de memórias como do sistema *fork-and-join*.

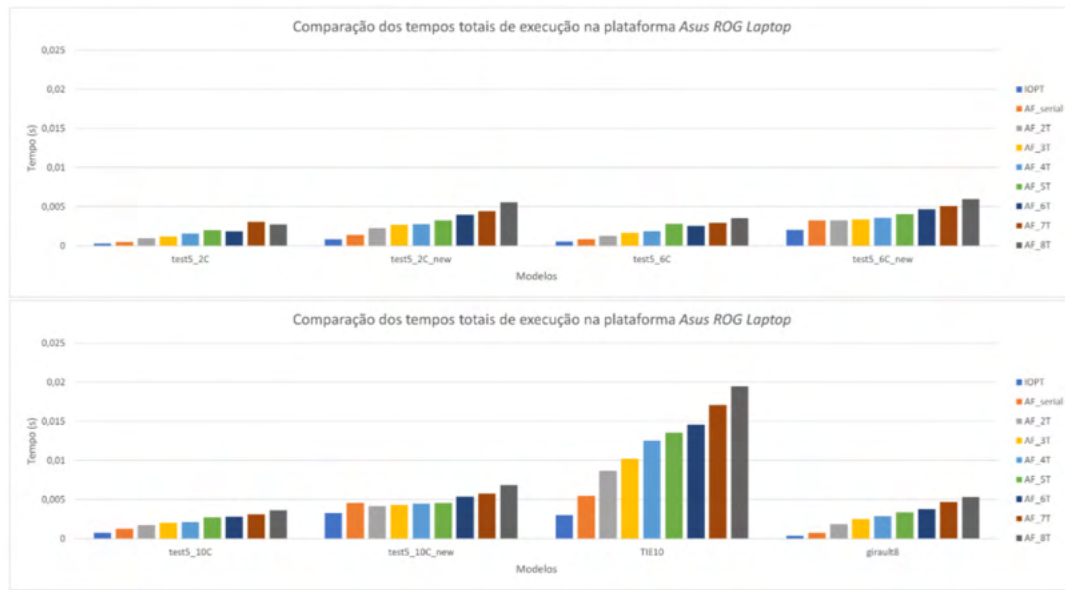


Figura 5.21: Gráfico de resultados obtidos da execução dos modelos com o algoritmo final na plataforma *Asus ROG Laptop*.

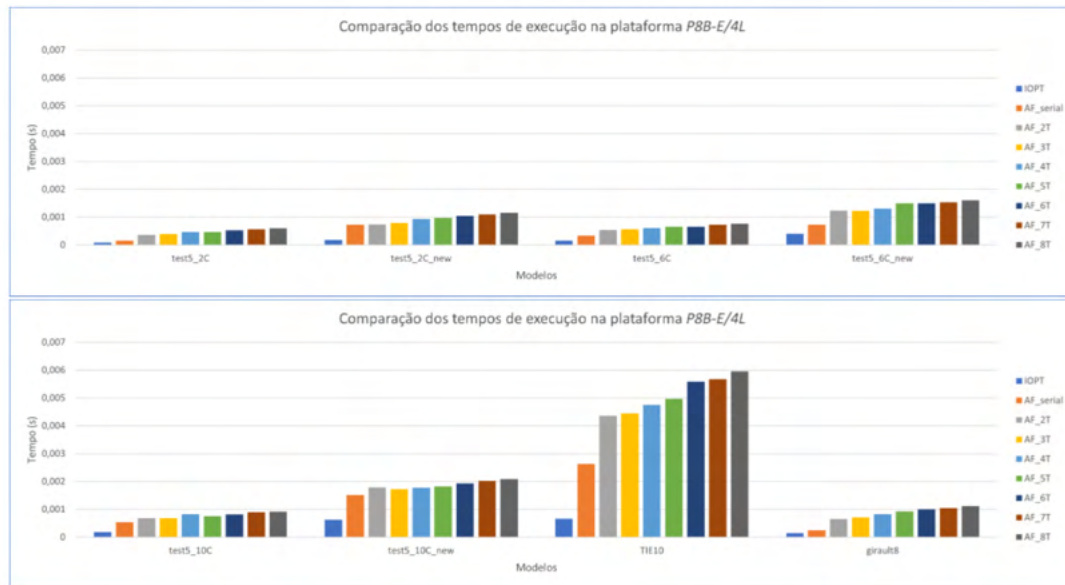


Figura 5.22: Gráfico de resultados obtidos da execução dos modelos com o algoritmo final na plataforma *P8B-E/4L*.

E de maneira a verificar o impacto do *overhead* criado pelo sistema de *fork-and-join* do OpenMP assim como no algoritmo de computação paralela, foi realizado o teste dos tempos de processamento das transições, para ambas as plataformas *Asus ROG Laptop* e *P8B-E/4L* representados nos gráficos das figuras 5.23 e 5.24 respectivamente.

Verifica-se então pelos resultados obtidos dos gráficos das figuras 5.23 e 5.24 que o comportamento dos gráficos é idêntico aos dos gráficos do processamento das transições

5.3. TESTES DE VALIDAÇÃO E ANÁLISE DE RESULTADOS

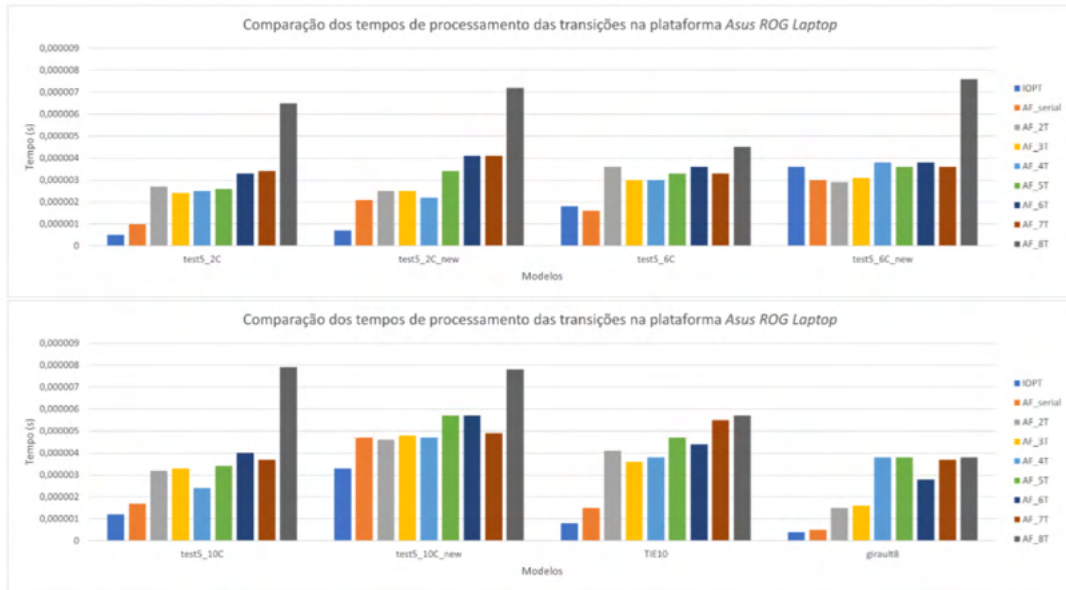


Figura 5.23: Gráfico de resultados obtidos do processamento de transições dos modelos com o algoritmo final na plataforma *Asus ROG Laptop*.

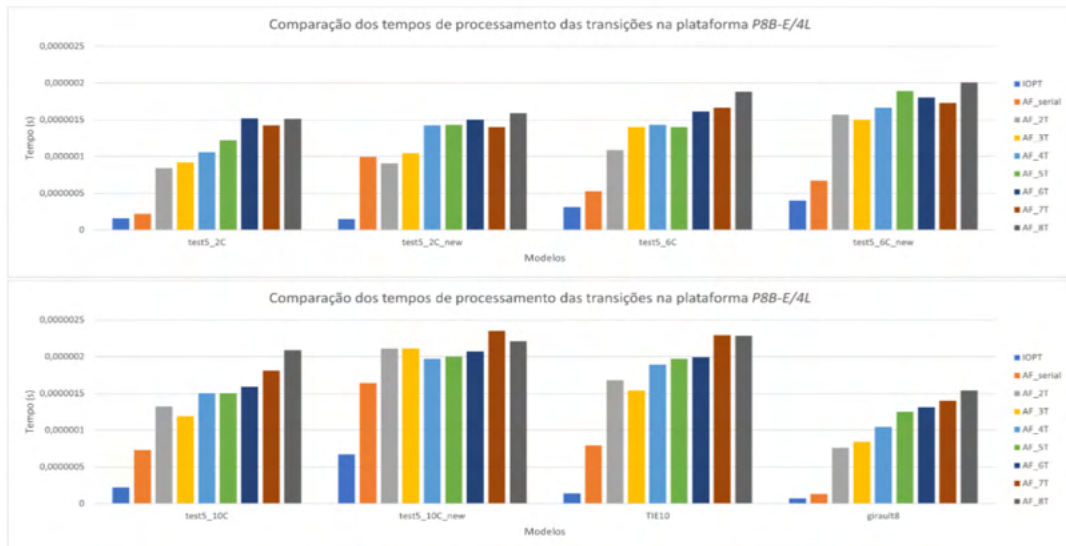


Figura 5.24: Gráfico de resultados obtidos do processamento de transições dos modelos com o algoritmo final na plataforma *P8B-E/4L*.

do algoritmo de computação paralela, e pela mesma razão que no algoritmo de computação paralela, ou seja, o *overhead* criado pelo sistema *fork-and-join* é muito superior aos benefícios da utilização do algoritmo de contendas e dos benefícios de utilizar múltiplas threads de processamento.

CONCLUSÕES

6.1 Balanço geral

Nesta dissertação pretendeu-se melhorar o código C gerado pelo gerador automático de código da plataforma IOPT-Tools. De forma a melhorar o código gerado, foram feitas modificações para que este incluísse mecanismos de aproveitamento do conceito de contendas e conflitos, mas também estivesse adaptado para processamento paralelo para que se possa utilizar tecnologia *multi-core*.

Neste trabalho foi utilizada a tecnologia OpenMP de forma a adaptar o código C gerado para a utilização de vários Cores. Sobre o gerador automático foram exploradas tecnologias como a linguagem XSLT que é utilizada para gerar os ficheiros de código C a partir de ficheiros PNML. Foi utilizado também a linguagem PHP de forma a modificar e criar algumas funções no servidor para a modificação dos ficheiros de PNML e para o conversor de modelos PIPE para modelos PNML.

Relativamente ao trabalho desenvolvido, este pode ser dividido em três partes significativas de contribuição para a ferramenta IOPT-Tools. A primeira focando em todas as modificações feitas a nível do ambiente IOPT-Tools, para que este integre as informações sobre contendas e conflitos de uma rede de Petri no seu respectivo PNML. A segunda sendo o melhoramento do código C gerado. E finalmente o um conversor de modelos PIPE para modelos PNML.

Em relação à primeira parte do trabalho desenvolvido, embora que não fosse um objetivo final da dissertação, adaptar o PNML para conter explicitamente as informações de contendas e conflitos, foi essencial para o desenvolvimento dos algoritmos. Fornecendo também um contributo ao nível de suporte para trabalhos futuros, que envolvam o mesmo tema.

Quanto às modificações feitas no código gerado, estas podem ser divididas em duas

partes, uma primeira que aproveita o conceito de contenda e conflito, de forma a evitar a verificação de transições, que devido à escassez de marcas e a conflitos diretos, já sabemos à priori que não irão disparar, permitindo assim um aproveitamento no tempo de processamento. A segunda parte engloba a utilização da API OpenMP, que nos permite adaptar código C para a utilização de vários *cores* de um CPU utilizando o conceito de *fork-and-join*.

Em relação ao conversor de modelos PIPE para modelos PNML utilizados na plataforma, IOPT-Tools, este foi inteiramente desenvolvido em PHP, e permitiu o uso de modelos de redes de Petri da plataforma Hippo (Hippo, s.d.) para o teste e validação do conteúdo desenvolvido.

Para a validação decidiu-se testar as diferentes modificações feitas ao algoritmo, assim como o algoritmo final de formas separadas. Esta divisão permitiu identificar o impacto da utilização do OpenMP, separadamente do impacto do algoritmo de aproveitamento de contendas e de conflitos.

Começando pelo impacto do algoritmo de aproveitamento de contendas e conflitos, foi possível perceber como o algoritmo afetava o código C gerado em modelos com diferentes quantidades de contendas e conflitos. Com os resultados obtidos em várias plataformas, foi possível perceber que o algoritmo apresentou resultados positivos para todos os modelos, em relação ao tempo de execução em plataformas com CPUs mais complexos, mas que por sua vez os resultados não apresentavam muita consistência (devido a outros processos que decorriam ao mesmo tempo). Na plataforma *Zedboard* obtivemos resultados mais consistentes, devido ao facto de executarmos os modelos em *bare metal*, ou seja, sem nenhum outro processo a decorrer, o que nos permitiu tirar conclusões fiáveis, nomeadamente, concluímos que para modelos de muitas transições e com pouca escassez de marcas, o algoritmo não apresenta resultados positivos, mas se o número de transições ignoradas já for suficiente em relação ao tamanho do modelo, já compensa o uso do algoritmo.

Ainda sobre o algoritmo de contendas, foi testado nas plataformas *Arduino MEGA* e *Arduino DUE*, e em ambas as plataformas o resultados não foram positivos, sendo que os testes efetuados nestas plataformas foram diferentes, dos realizados nas outras plataformas, de modo a perceber as diferenças nos resultados. O que se conclui com estas duas plataformas, é que para a plataforma *Arduino Mega* os resultados não são positivos de todo, e quanto à plataforma *Arduino DUE* os resultados para a maioria dos modelos não foram positivos também, embora um pouco melhores que na plataforma *Arduino Mega*. Desta forma os resultados obtidos não são consequência do ambiente de desenvolvimento de Arduino IDE, mas sim da arquitetura dos processadores utilizados, pois existe diferença no comportamento dos resultados para as duas plataformas *Arduino DUE* e *Arduino MEGA*. Logo a qualidade e complexidade dos CPUs influencia na utilização do algoritmo de contendas, sendo que obtemos melhores resultados com CPUs mais complexos e completos.

Quanto ao algoritmo de processamento paralelo, em ambas as plataformas não apresentou resultados positivos, sendo que é criado um *overhead* constante em cada modelo

devido à divisão das estruturas de memória, e relativamente ao método de *fork-and-join* o que os resultados demonstraram é que o *overhead* criado pela criação e junção da threads é sempre maior que o tempo de processamento poupado pela utilização de duas ou mais threads, sendo que este *overhead* aumenta com o aumento de threads a serem utilizadas.

Finalmente para o algoritmo final o que se conclui é que embora a parte do aproveitamento de contendas nos proporcione alguma poupança de tempo de processamento para modelos em que as transições ignoradas compensam o tamanho do modelo, o *overhead* criado pela utilização de OpenMP e o *overhead* de divisão e junção das estruturas de memória, é demasiado elevado e que torna a utilização do OpenMP não proveitosa na execução dos modelos de redes IOPT.

Desta forma e como conclusão final, a utilização do algoritmo de aproveitamento de contendas e conflitos é útil na medida de utilização de modelos de muitas transições ignoradas, ou seja, poucas marcas para muitas transições, e em plataformas com CPU com alguma complexidade, sendo o Servidor IOPT-Tools uma delas. Quanto à utilização da API OpenMP, esta não se provou útil devido aos *overheads* criados, sendo útil apenas em casos em que o disparo de transições se torne bastante pesado computacionalmente.

6.2 Trabalho futuro

Desta forma e tendo em consideração a conclusão obtida, uma sugestão para trabalho futuro, é de que se desenvolva a plataforma IOPT-Tools, de modo a incorporar Redes de Petri coloridas. Estas permitem a utilização de marcas complexas, que por sua vez tornam a verificação e disparo de transições um processo mais complexo, permitindo desta forma que o algoritmo de OpenMP ter algum impacto positivo, em termos de tempo de execução, em algoritmos futuramente desenvolvidos para processar tais redes de Petri.

Outra sugestão de trabalho futuro é modificar o PNML utilizado pelas redes de Petri IOPT, sendo que este não incorpora todas as definições da norma ISO/IEC 15909-2 (*Systems and software engineering – High-level Petri nets, Part 2: Transfer Format*. 2011) e facilitaria a utilização de modelos de outras plataformas assim como modelos PIPE da plataforma Hippo (Hippo, s.d.).

BIBLIOGRAFIA

- Akhter, S. & Roberts, J. (2006). *Multi-core programming*. Intel press Hillsboro.
- Arduino. (s.d.). Obtido 28 janeiro 2021, de <https://www.arduino.cc/en/guide/environment>
- Barros, J. P. (1996). *CpPNeTS: uma classe de Redes de Petri de alto-nível: implementação de um sistema de suporte à sua aplicação e análise* (tese de doutoramento).
- Cabasino, M. P., Giua, A. & Seatzu, C. (2013). Introduction to Petri nets. Em *Control of Discrete-Event Systems* (pp. 191–211). Springer.
- Campos-Rebelo, R., Pereira, F., Moutinho, F. & Gomes, L. (2011). From IOPT Petri nets to C: An automatic code generator tool. Em *2011 9th IEEE International Conference on Industrial Informatics* (pp. 390–395). IEEE.
- Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D. & McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan kaufmann.
- Clark, J. et al. (1999). Xsl transformations (xslt). *World Wide Web Consortium (W3C)*. URL <http://www.w3.org/TR/xslt>, 103.
- Clark, J. (2001). Relax NG specification. <http://www.oasis-open.org/committees/relaxng/spec-20011203.html>. Consultado: 2021-01-14.
- Gomes, L., Moutinho, F., Pereira, F., Ribeiro, J., Costa, A. & Barros, J.-P. (2014). Extending input-output place-transition Petri nets for distributed controller systems development. Em *2014 International Conference on Mechatronics and Control (ICMC)* (pp. 1099–1104). IEEE.
- Gomes, L., Barros, J. P., Costa, A. & Nunes, R. (2007). The input-output place-transition Petri net class and associated tools. Em *2007 5th IEEE International Conference on Industrial Informatics* (Vol. 1, pp. 509–514). IEEE.
- Gomes, L., Moutinho, F. & Pereira, F. (2013). IOPT-tools—A Web based tool framework for embedded systems controller development using Petri nets. Em *2013 23rd International Conference on Field programmable Logic and Applications* (pp. 1–1). IEEE.
- Gomes, L., Rebelo, R., Barros, J. P., Costa, A. & Pais, R. (2010). From Petri net models to C implementation of digital controllers. Em *2010 IEEE International Symposium on Industrial Electronics* (pp. 3057–3062). IEEE.
- Gomes, L. F. d. S. (1997). *Redes de Petri Reactivas e Hierárquicas-integração de formalismos no projecto de sistemas reactivos de tempo-real*.
- Hippo. (s.d.). Obtido 15 setembro 2021, de <http://www.hippo.iie.uz.zgora.pl/index.php>

- Hujsa, T. & Devillers, R. (2018). On deadlockability, liveness and reversibility in subclasses of weighted Petri nets. *Fundamenta Informaticae*, 161(4), 383–421.
- IOPT-Tools. (s.d.). Obtido 20 setembro 2021, de <http://gres.uninova.pt/IOPT-Tools-V1.1>
- Systems and software engineering – High-level Petri nets, Part 2: Transfer Format*. (2011). International Organization for Standardization.
- Itmi, M. (1993). Object modeling and simulating extended Petri nets. Em *Proceedings of IEEE Systems Man and Cybernetics Conference-SMC* (Vol. 4, pp. 650–655). IEEE.
- Jensen, K. (2013). *Coloured Petri nets: basic concepts, analysis methods and practical use*. Springer Science & Business Media.
- Kindler, E. (2006). The petri net markup language and iso/iec 15909-2: Concepts, status, and future directions. *Entwurf komplexer Automatisierungssysteme*, 9, 35–55. Consultado: 2021-02-25.
- Lima, P. L. G. (2009). *Geração de código VHDL a partir de especificações IOPT PNML2VHDL* (tese de doutoramento, FCT-UNL).
- Lourenço, J. M. (2021). *The NOVAtthesis L^AT_EX Template User's Manual*. NOVA University Lisbon. Obtido de <https://github.com/joaoamlourenco/novathesis/raw/master/template.pdf>
- Moutinho, F. & Gomes, L. (2014). Asynchronous-channels within Petri net-based GALS distributed embedded systems modeling. *IEEE Transactions on Industrial Informatics*, 10(4), 2024–2033.
- Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 541–580. doi:10.1109/5.24143
- Muthana, P., Swaminathan, P., Tummala, R., Sundaram, V., Wan, L., Bhattacharya, S. & Raj, P. (2005). Packaging of multi-core microprocessors: Tradeoffs and potential solutions. Em *Proceedings Electronic Components and Technology, 2005. ECTC'05*. (Pp. 1895–1903). IEEE.
- Nielsen, M., Plotkin, G. & Winskel, G. (1981). Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1), 85–108.
- OpenMp. (s.d.). Obtido 28 janeiro 2021, de <https://www.openmp.org/>
- Pais, R., Barros, S. & Gomes, L. (2005). A tool for tailored code generation from Petri net models. Em *2005 IEEE Conference on Emerging Technologies and Factory Automation* (Vol. 1, 8–pp). IEEE.
- Pereira, F. & Gomes, L. (2013). Automatic synthesis of VHDL hardware components from IOPT Petri net models. Em *IECON 2013-39th Annual Conference of the IEEE Industrial Electronics Society* (pp. 2214–2219). IEEE.
- Pereira, F., Moutinho, F. & Gomes, L. (2014). IOPT Tools User Manual. *Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologias, GRES Research Group, versão, 1*.
- Pereira, F., Moutinho, F., Gomes, L. & Campos-Rebelo, R. (2011). IOPT Petri net state space generation algorithm with maximal-step execution semantics. Em *2011 9th IEEE International Conference on Industrial Informatics* (pp. 789–795). IEEE.

- Pereira, F., Moutinho, F., Ribeiro, J. & Gomes, L. (2012). Web based IOPT Petri net Editor with an extensible plugin architecture to support generic net operations. Em *IECon 2012-38th Annual Conference on IEEE Industrial Electronics Society* (pp. 6151–6156). IEEE.
- Peterson, J. L. (1977). Petri nets. *ACM Computing Surveys (CSUR)*, 9(3), 223–252.
- Petri, C. A. (1962). *Kommunikation mit Automaten* (Dissertation, Schriften des IIM N.º 2). Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn. Bonn: Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn.
- php.net. (s.d.-a). Obtido 24 fevereiro 2021, de <https://www.php.net/manual/en/class.domdocument.php>
- php.net. (s.d.-b). Obtido de <https://www.php.net/manual/en/class.xsltprocessor.php#class.xsltprocessor>
- Reisig, W. (2012). *Petri nets: an introduction*. Springer Science & Business Media.
- Reisig, W. (2013). *Understanding Petri nets: modeling techniques, analysis methods, case studies*. Springer.
- Ribeiro, J., Moutinho, F., Pereira, F., Barros, J. P. & Gomes, L. (2011). An Ecore based Petri net type definition for PNML IOPT models. Em *2011 9th IEEE International Conference on Industrial Informatics* (pp. 777–782). IEEE.
- Schauer, B. (2008). Multicore processors—a necessity. *ProQuest discovery guides*, 1–14.
- w3schools. (s.d.). Obtido 27 fevereiro 2021, de https://www.w3schools.com/xml/xsl_intro.asp
- Weber, M. & Kindler, E. (2003). The Petri net markup language. Em *Petri Net Technology for communication-based systems* (pp. 124–144). Springer.
- Xilinx. (s.d.). Vitis Platform. Obtido 28 janeiro 2021, de <https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#overview>
- Ye, X., Zhou, J. & Song, X. (2003). On reachability graphs of Petri nets. *Computers & Electrical Engineering*, 29(2), 263–272.

