Pós-Graduação em Ciência da Computação

"ESTIMATIVA DO CONSUMO DE ENERGIA DEVIDO AO SOFTWARE: UMA ABORDAGEM BASEADA EM REDES DE PETRI COLORIDAS"

Por

Meuse Nogueira de Oliveira Júnior

Tese de Doutorado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, OUTUBRO/2006

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MEUSE NOGUEIRA DE OLIVEIRA JÚNIOR

"Estimativa do Consumo de Energia Devido ao Software: Uma Abordagem Baseada em Redes de Petri Coloridas"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: PROF. PAULO ROMERO MARTINS MACIEL

RECIFE, OUTUBRO/2006

Oliveira Júnior, Meuse Nogueira de
Estimativa do consumo de energia devido ao software: uma abordagem baseada em redes de Petri coloridas / Meuse Nogueira de Oliveira Júnior. – Recife : O autor, 2006.
213 p. : il., fig., tab.

Tese (doutorado) – Universidade Federal de Pernambuco. CIN. Ciência da Computação, 2006.

Inclui bibliografia e apêndices.

1. Redes de Petri. 2. Consumo de energia. 3. Sistemas embutidos. 4. Avaliação de consumo. I. Título.

511.35 CDD (22.ed.) MEI2007-017

Dedicatória

Este trabalho é dedicado a três pessoas de grande importância em minha vida: minha esposa Yale, cuja presença mostra-se em todas as minhas realizações, minha filha Teresa Luísa, que me inspira a cada momento, e a minha mãe Maria José, que edificou a fortaleza como um traço do meu caráter.

Agradecimentos

Após os anos de trabalho que compreenderam a edificação desta contribuição cientifica, tenho a sorte de dever agradecimentos a muitas pessoas. Devo agradecer, antes de mais nada, a minha família - minha esposa, minha filha e minha mãe - cujo amor me acompanhou ao longo do caminho. Agradeço ao amigo Frederico Braga, cujo apoio assumiu duas dimensões: a técnica e a pessoal. No plano técnico, agradeço-lhe pelo suporte na aplicação de sistemas embutidos comerciais como experimentos de validação desse trabalho. No plano pessoal, sou-lhe grato pelas reflexões epistemológicas sobre o doutorado e suas implicações. Agradeço aos colegas Silvino Neto, Raimundo Barreto, Eduardo Tavares, Ângelo Ribeiro e Cesar Oliveira, pelo espírito de equipe e pelas valorosas contribuições aos artigos científicos resultantes desta pesquisa. No plano público, sou profundamente grato à Profª. Judith Kelner, cuja nobreza de caráter sempre fez valer o eticamente correto sobre o politicamente cômodo. Agradeço ainda às equipes de suporte técnico e manutenção do CIn, cuja presteza e competência tornam possível ao pós-graduando abstrair-se de problemas menores, em prol da qualidade de sua pesquisa. Por fim, mas não em menor importância, agradeço ao meu orientador, o Prof. Paulo Romero Martins Maciel, cuja hombridade é inspiradora. Agradeço pela coragem intelectual em apoiar minhas idéias, e pelo suporte incessante em todas as etapas do trabalho, materializado num misto de amizade e orientação científica.

"All models are wrong, but some are useful."

— A. ENSTEIN (1879-1955)

Resumo

Esta tese tem seu foco na análise do consumo de energia de microprocessadores no contexto dos sistemas embutidos. Para tanto, são propostos dois modelos para simulação e análise. Tais modelos operam com base na descrição do conjunto de instruções da arquitetura alvo em redes de Petri coloridas. O primeiro modelo aplica as redes de Petri coloridas para a avaliação do comportamento do código frente a uma descrição determinística das instruções. O segundo estende o primeiro de forma a explorar o espaço de possibilidades de execução do código por meio da descrição probabilística dos possíveis fluxos de execução. Uma taxonomia para a descrição dos elementos presentes nos perfis de execução e consumo é proposta. Com base em tal taxonomia, formaliza-se um mecanismo para a análise de consumo de energia devido ao *software*. A abordagem proposta oferece três contribuições básicas: (i) criação de modelos de descrição de arquiteturas sobre uma linguagem de modelagem formal, as redes de Petri coloridas; (ii) proposição de um modelo de descrição estrutural do *software*, no qual os possíveis fluxos de execução estão explícitos na semântica de descrição; e (iii) proposição de um modelo probabilístico para descrição, simulação e avaliação do consumo de energia devido ao *software*. O modelo probabilístico ataca o *problema da dependência de padrão* pela eliminação do vetor de teste em prol de um modelo probabilístico de comportamento do *software*. Dessa forma, esta tese estabelece uma abordagem nova para a análise de consumo de energia, promovendo um formalismo baseado em redes de Petri coloridas para a criação de ferramentas.

Palavras-chave: Redes de Petri Coloridas, Consumo de Energia do Software, Descrição de Processadores.

Abstract

This thesis focuses on energy consumption analysis of microprocessors in the context of embedded systems. It proposes two simulation models, which adopt deterministic and probabilistic approaches. Both models are based on processor instruction set description using *Coloured Petri Nets* (CPN). The deterministic model makes it possible to build automatically an instruction level simulator formatted as a CPN executable model. The probabilistic model extends this concept by replacing the behavioral description with probabilistic inferences regarding execution flow. A taxonomy is presented in order to systematise the energy consumption analysis of code. The energy consumption evaluation of software is formalised on the basis of this taxonomy. The proposed approaches offer three basic contributions: (i) a formal model for simulation and analysis, applying a widely-used formal language, the Coloured Petri Nets, (ii) an executable description model where the code struture is inherent in the description, and (iii) a probabilistic model for dealing with the *dependence pattern problem* by eliminating the concept of test vector. As a result, this thesis presents a new approach to energy consumption analysis, promoting the application of CPN formalism to developing new concepts for tools construction.

Key-words: Coloured Petri Nets, Software Energy Consumption, Architecture Description Language.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

- ADL- *Archtecture Description Language*.
- ApCod- Aplicação de Código.
- AI- Anotação de Instrução.
- AC- Anotação de Cabeçalho.
- CI- Contexto Interno.
- CPN- *Coloured Petri Net*.
- CodInst- Código de Instrução.
- CodProbInst- Código Probabilístico de Instrução.
- CS- Cenário de Simulação.
- DT- Distribuidor de *Token*.
- FI- Função Interna.
- F_{R_n}- Função de Renomeação.
- F_{Flow}- Função de Fluxo.
- HCPN-*Hierarchical Coloured Petri Nets*.
- ID- Instrução Determinística.
- IP- Instrução Probabilística.
- ISA- *Instruction Set Architecture*.
- MCIOrd- Modelo CPN de Instrução Ordinária.
- MCICon- Modelo CPN de Instrução de Desvio Condicional.

- MCI- Modelo CPN de Instrução.
- cMCI- Conjunto de Modelos CPN de Instrução.
- MCA- Modelo CPN de Aplicação.
- MCD- Modelo CPN de Descrição.
- MCPIO- Modelo CPN-Probabilístico de Instrução Ordinária.
- MCPIC- Modelo CPN-Probabilístico de Instrução de Desvio Condicional.
- MCPI- Modelo CPN-Probabilístico de Instrução.
- cMCPI- Conjunto de Modelos CPN-Probabilístico de Instrução.
- MCPA- Modelo CPN-Probabilístico de Aplicação.
- MCPD- Modelo CPN-Probabilístico de Descrição.
- ProxyOrd- Proxy Ordinário.
- ProxyDesv- Proxy de Desvio.
- RH- Recurso de *Hardware*.
- RT- Raptora de *Token*.
- rCS- Cenário de Simulação de Referência.
- SA- Sondas de Análise.

Capítulo 1

Introdução

1.1 Contextualização

A designação de *Sistema Embutido*, ou *Embarcado*, cabe a quaisquer sistemas digitais que estejam incorporados a outros sistemas, com o fim de acrescer ou otimizar funcionalidades. Equipamentos eletromédicos, instrumentação eletrônica, automóveis, aeronaves, telefones, modernos equipamentos eletrodomésticos, todos esses sistemas têm, em maior ou menor grau, um sistema digital embutido que otimiza e implementa algumas funcionalidades que simplesmente não existiam em suas concepções originais. Alguns outros produtos só se tornaram possíveis comercialmente graças ao ato de embutir sistemas computacionais em sua base funcional, como é o caso do telefone celular, do sistema de injeção eletrônica, do freio ABS e da navegação eletrônica de aeronaves. A eletrônica - e particularmente o conceito de sistema digital- foi "embutida" em todos os produtos de base tecnológica, e por vezes em outros não associados à tecnologia. Esse fenômeno se deu devido aos avanços e barateamento das tecnologias de sistema microprocessados no decorrer da década de 1980, culminando em uma avalanche de novas concepções tecnológicas durante a década de 1990. Junto com o desafio de embutir sistemas computacionais de aplicação bem específica, veio o de projetar e produzir tais sistemas de forma competitiva. Novos paradigmas de projeto tiveram que ser criados visando estabelecer ganhos de produtividade e qualidade de projeto. A criação de tais paradigmas e as otimizações das técnicas existentes são, em última análise, o objetivo da maioria das pesquisas voltadas à sistemas embutidos.

Nos últimos anos, tem crescido o interesse sobre uma questão de particular importância: o consumo de energia. Baixo consumo de energia como critério de projeto tem sido um tópico presente no espaço de discussão tanto da comunidade acadêmica quanto da indústria. Por muito tempo, a preocupação com o consumo foi um tópico

restrito à uma pequena categoria de projetistas de áreas específicas, como equipamentos eletromédicos para implantes. Contudo, com a proliferação de produtos portáteis, com crescente capacidade computacional e grandes exigências quanto à autonomia, o consumo de energia tornou-se uma métrica de qualidade do projeto. Adicionalmente, a dissipação de energia tem impacto direto sobre os custos de encapsulamento de componentes, estabelecendo assim um fator de custo final do produto. Os fatores que determinam o consumo de um sistema são múltiplos e fortemente associados à sua arquitetura e ao seu domínio de aplicação, sendo largamente aceita a opinião que não existirá uma técnica universal para otimização do consumo de energia [111]. Em sistemas embarcados, podemos distinguir três agentes de consumo: o componente de *software* (processador e memórias), os componentes de *hardware* e as vias de comunicação (conexões). Contudo, os componentes de *software* e *hardware* são os principais protagonistas do consumo, pois suas descrições estão vinculadas diretamente ao padrão comportamental do sistema. Com referência ao componente de *hardware*, algumas técnicas de análise estão bastante amadurecidas e já são incorporadas em ferramentas como a Galaxy Power da Synopsys [3]. Há, no entanto, a necessidade de amadurecimento dos métodos de avaliação dos componentes de *software* e *hardware* em nível de sistema [53].

1.2 Justificativa

Em contraste com um computador pessoal, que potencialmente executará milhares de programas diferentes ao longo de sua vida útil, um sistema embutido executa um único programa por todo o seu tempo de vida, o seu *firmware*. Mesmo em sistemas nos quais o *firmware* pode ser atualizado, esta ação não implica em mudanças significativas do código. Essa situação de programa único permite a otimização do sistema de modo que esse alcance a sua eficiência máxima, seja pela otimização do código, seja pelo seu particionamento em *hardwares* que operem como co-processadores. Vahid *et al* [94] chamam tal processo de *architecture tuning*. A identificação de laços internos nos códigos de máquina está relacionada à identificação de centros de custo do código, não só em termos de tempo de execução, mas também em termos de consumo de energia. Uma vez identificados os laços, os códigos do corpo interno podem ser reescritos com instruções de menor consumo ou implementados em *hardware*. Classicamente, as análises de potência são realizadas por modelos simuláveis do *hardware*, que, para proverem estimativas com boa precisão, devem estar em um nível de abstração muito baixo, tornando o modelo complexo e de grande custo computacional. A esses fatores

são incorporados a baixa flexibilidade do modelo de energia [111] e a necessidade de acesso a detalhes da tecnologia de implementação do *hardware*, que constituem propriedades intelectuais fechadas. Trabalhos como os de Brooks e Vijaykrishnan [14, 110, 107] têm aberto novas perspectivas na direção de modelos RTL (*Register Transfer Level*) e baseados em arquiteturas. Infelizmente, mesmo esses modelos parecem pouco flexíveis para a modelagem de uma grande variedade de dispositivos [29]. Modelos de potência baseados em instruções aparecem como uma alternativa [99, 100, 52, 98, 13]. Tais modelos realizam a análise de potência pela simulação de instruções, não havendo necessariamente nenhum formalismo em seu modelo de computação interna. A descrição da estrutura do *software* (identificação de seqüências, laços e laços aninhados) não é inerente à análise por simulação, porque os simuladores são construídos com base na descrição do processador e não do programa que será executado. Isso abre espaço para pesquisas voltadas à criação de modelos de análise de potência baseados em linguagens de modelagem, os quais podem, de forma inerente, fornecer formalismo e descrição estrutural do programa. Nesse contexto, as redes de Petri Coloridas figuram como uma interessante opção formal, pois oferecem: (i) formalismo, através de um consolidado conjunto de regras; (ii) representação gráfica, facilitando a cognição do modelo;(iii) representação hierárquica, suportando modelos complexos; (iv) descrição algorítmica executável (linguagem de programação); e (v) mecanismos consolidados de simulação.

1.3 Contribuições

Motivado pelas considerações expostas, este trabalho apresenta técnicas para a modelagem de arquiteturas de conjunto de instruções de processadores (*ISA Instruction Set Architecture*) em redes de Petri coloridas [41] e mecanismos de análise que geram subsídios para a otimização do código, ou o seu particionamento em *hardware*, sem que haja modificação da arquitetura do processador. Primeiramente, um modelo determinístico de descrição de processadores é apresentado, sendo este modelo denominado MCD (Modelo CPN de Descrição). A partir desse modelo, a avaliação de códigos executáveis da arquitetura modelada pode ser efetuada. A Figura 1.1 apresenta a metodologia básica de avaliação, na qual o código executável (código binário) é convertido para um modelo em rede de Petri Coloridas. A avaliação é baseada na simulação da rede que representa o código. Dessa forma, o processo de simulação é realizado em um mecanismo formalmente construído e independente da arquitetura alvo. A metodologia de modelagem faz com que a semântica e sintaxe das redes de Petri coloridas operem como uma linguagem de descrição de arquitetura (*ADL*), fincando bases para uma fu-

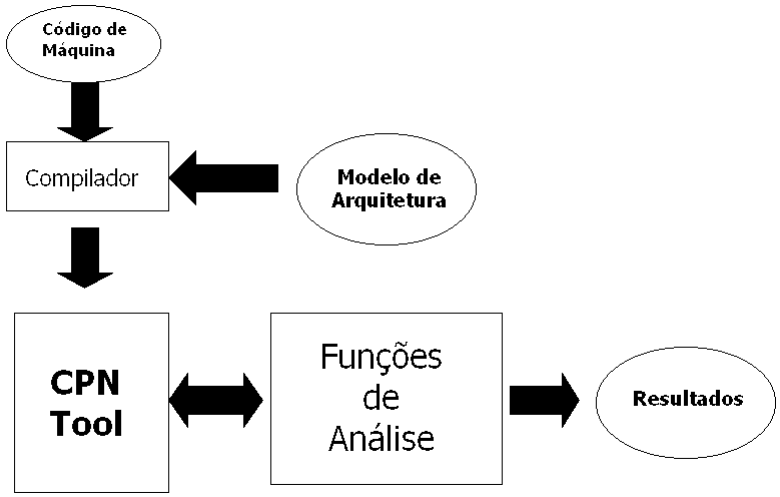

Figura 1.1: Metodologia Proposta

tura *ADL* de forte apelo formal. Um arcabouço de análise foi construído acoplando o ambiente EZPetri [42], desenvolvido pelo Departamento de Sistema Computacionais da Escola Politécnica de Pernambuco, à ferramenta de modelagem e análise de redes de Petri coloridas CPNTools. Para fins de validação, a arquitetura i8051 foi modelada. O modelo determinístico possibilita o instanciamento automático de um modelo de simulação do código de interesse. A simulação determinística, no entanto, esbarra em um clássico problema conhecido como *problema da dependência de padrão* [75]. O problema da dependência de padrão estabelece a limitação das análises por simulação devido à dificuldade de se construir vetores de teste cujo padrão descreva satisfatoriamente o universo de cenários de operação do sistema. De forma a atacar essa questão, criou-se uma extensão do modelo que permite uma simulação probabilística do código de interesse. Esse modelo é denominado MCPD (Modelo CPN-Probabilístico de Descrição) e o processo de simulação é baseado no método Monte Carlo [111]. O modelo probabilístico é construído pela descrição dos cenários de execução do código, em função das probabilidades de variação do fluxo de execução. Identicamente ao modelo determinístico, um conjunto de recursos de modelagem e análise foi construído, permitindo que a descrição probabilística dos cenários do sistema possam ser inseridas diretamente nas linhas de código *assembly*.

Diante disso, esta tese apresenta três contribuições básicas ao campo da modelagem e avaliação do consumo de energia em *softwares* embutidos:

- criação de modelos de descrição de arquiteturas sobre uma linguagem de modelagem formal, possuidora de mecanismos de simulação e análise;
- proposição de um modelo de descrição estrutural do *software*, no qual os possíveis

fluxos de execução estão explícitos na semântica de descrição e;

- proposição de um modelo probabilístico para descrição, simulação e avaliação de consumo de energia devido ao *software*, atacando o problema de dependência de padrão pela eliminação do vetor de teste (seja ele determinístico ou estocástico) em prol de um modelo probabilístico do comportamento do *software*.

1.4 Objetivos e Estrutura da Tese

A tese cumpre dois objetivos básicos, um de natureza geral e outro de natureza específica.

- **Objetivo Geral**: apresentar uma nova abordagem para a análise de consumo de energia devido ao *software*, aplicando redes de Petri coloridas.
- **Objetivo Específico**: especificar modelos, em redes de Petri coloridas, para arquiteturas de micro-controladores, bem como mecanismos de análise determinística e probabilística de tais modelos.

Para tanto, esta tese está dividida em sete capítulos e três apêndices. Em nome da clareza de exposição, certos termos em inglês foram utilizados de forma a prevenir construções obscuras decorrentes de sua tradução para o português, particularmente aqueles referentes à taxonomia proposta, que foram criados originalmente em inglês, visando à divulgação científica. O Capítulo 2 realiza uma revisão da literatura relacionada com a pesquisa. O Capítulo 3 introduz as redes de Petri e apresenta seus conceitos básicas e aplicações. O Capítulo 4 apresenta o modelo determinístico. O Capítulo 5 apresenta o modelo probabilístico. O Capítulo 6 descreve os experimentos e protótipos desenvolvidos para a validação dos modelos. E, por fim, o Capítulo 7 desenvolve um pequeno ensaio conclusivo, no qual proposições para trabalhos futuros são apresentadas.

O Apêndice *A* discute as questões relacionados a problemas metrológicos associados à caracterização de dispositivos enquanto que o Apêndice *B* apresenta sumariamente o método de Monte Carlo. Concluindo, o Apêndice *C* lista as publicações referentes a esta pesquisa e contribuições em outras pesquisas.

Capítulo 2

Revisão da Literatura

2.1 Introdução

Potência é conceitualmente uma medida de consumo de energia no tempo. Por vezes, encontram-se na literatura termos como **consumo de potência** ou **dissipação de potência**, quando seria mais adequado simplesmente o termo **potência**. Neste trabalho, emprega-se sempre o termo consumo de energia ou consumo de energia total para descrever o montante de energia consumida em um processo. O termo potência descreve a taxa com que essa quantidade de energia foi consumida. Este capítulo faz uma rápida apresentação dos conceitos associados à análise de consumo de energia em sistemas embutidos. Nas primeiras seções, são apresentadas as principais questões que envolvem a estimativa do consumo de energia em um sistema embutido, no final deste capítulo, discutem-se os trabalhos relacionados e as tendências de pesquisa.

2.2 Consumo de Energia - Mecanismos e Conceitos

Para o projeto de sistemas de baixo consumo, configuram-se dois desafios básicos: análise e otimização. A análise diz respeito aos mecanismo de estimativa de consumo em diferentes fases do projeto. O objetivo da análise é prover ao projetista mecanismos que assegurem as especificações de consumo. O modelo sobre o qual são construídos os mecanismos de estimativa precisa ser adequadamente avaliado em função do domínio de aplicação. Diferentes técnicas apresentam diferentes efeitos na relação existente entre tempo de análise e exatidão das estimativas. Nas primeiras fases do projeto, a ênfase é obter estimativas rápidas a partir de poucas informações sobre o sistema. Nessas fases, mecanismos de pouca exatidão são aceitáveis. Com o avanço do projeto, contudo, a necessidade de estimativas mais exatas cresce. A análise estabelece os

fundamentos das intervenções de otimização que o sistema deverá sofrer. A otimização deve ser entendida como o processo pelo qual os padrões de consumo do sistema são reduzidos sem violar as especificações do sistema. Otimizações manuais dependem da qualidade das ferramentas de análise e da experiência do projetista. A decisão pelo uso de uma técnica de otimização em particular envolve considerações sobre o impacto dessa escolha sobre o sistema. Por exemplo, na otimização do *hardware* a principal questão a ser analisada é o impacto sobre o retardo de propagação dos sinais, com influência direta sobre a velocidade síncrona dos circuitosOutras questões são tempo de projeto, testabilidade e capacidade de re-uso. Um projeto de baixo consumo não pode ser adequadamente realizado sem uma avaliação desses fatores. A tarefa do projetista é analisar cuidadosamente cada opção de implementação, garantindo as especificações originais. A seguir, são apresentados os conceitos básicos dessa análise.

2.2.1 Mecanismos Físicos

Existem dois tipos de consumo de energia em circuitos CMOS [111]: o consumo dinâmico e o consumo estático. O consumo dinâmico, causado pelo chaveamento dos circuitos, é decorrente da passagem dos transistores por uma região de operação linear (região ativa), que estabelece a condição em que tanto a tensão quanto a corrente são diferentes de zero simultaneamente, promovendo potência diferente de zero no transistor. Em última análise, esse efeito é regido pelas capacitâncias internas do transistor, pois essas determinam o intervalo de tempo que o transistor permanece dentro da região ativa. Esse intervalo está relacionado ao processo de carga e descarga dos capacitores intrínsecos (parasitas) sobre as resistências internas do dispositivo. Vias e barramentos partilham um modelo elétrico similar, apresentando também padrões de consumo dinâmico. Adicionalmente, devido à estrutura *totem-pole* dos circuitos CMOS (vide Figura 2.1(a)), a excursão de ambos os transistores pela região ativa simultâneamente, durante o chaveamento, promove uma súbita queda da impedância do *totem-pole*, resultando em surto de corrente durante o chaveamento (vide Figura 2.1(b)).

Uma vez que a tensão de alimentação é constante, a integração do pulso de corrente no tempo, multiplicado pela tensão de alimentação, fornece a energia consumida na transição. Esses surtos de corrente, presentes nas transições de circuitos CMOS, são denominadas *correntes de curto-circuito*. O consumo ocorre durante as transições de estados lógicos, sendo proporcional à taxa de chaveamento do circuito, ou seja à freqüência de operação. O consumo dinâmico é, então, relacionado ao processo de carga e descarga de capacitores parasitas do circuito. A Equação 2.1 descreve a relação entre potência, capacitância e freqüência de chaveamento, para o consumo dinâmico devido

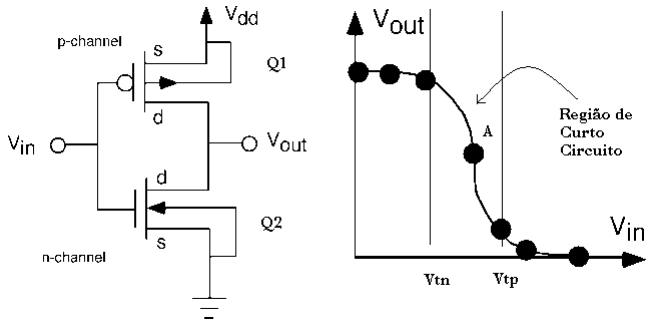

Figura 2.1: (a) Inversor CMOS (b) Curva de transferência

à carga e descarga dos capacitores parasitas. Onde P é a potência, V é a tensão e f a freqüência de chaveamento.

$$P = C_l \times V^2 \times f \tag{2.1}$$

Essa é uma equação fundamental para projetos de circuitos VLSI [111], pois é baseada em poucos parâmetros facilmente observáveis. A Equação 2.1 é referente ao consumo de uma única via ou transistor cuja capacitância associada é C_l. Uma generalização para o somatório do efeito de consumo de todas as capacitâncias presentes no circuito é definida pela Equação 2.2:

$$P = \sum_i \left(C_i \times V^2 \times f\right) = V^2 \times f \times \sum_i C_i = C_{total} \times V^2 \times f \tag{2.2}$$

A energia consumida durante uma transição, devido à corrente de curto-circuito, é dada pela Equação 2.3.

$$E_{curto} = \frac{\beta}{12} \times \tau \times (V_{tp} - V_{tn})^3 \tag{2.3}$$

Onde E_{curto} é a energia consumida, β é um coeficiente inversamente relacionado à resistência elétrica do material, sendo portanto dependente das dimensões do transistor[1], τ é a duração do sinal de entrada, V_{tp} e V_{tn} são os limiares de comutação (vide Figura 2.1(b)). Em componentes como portas, o consumo devido ao efeito curto-circuito flutua entre 5% a 10% do consumo total da porta [17]. Adicionalmente, a corrente de curto-circuito varia com a carga de saída do dispositivo. O consumo estático é relacionado aos estados lógicos dos circuitos. Em circuitos de tecnologia CMOS, o consumo estático é devido unicamente à fuga de corrente nos transistores em estado de corte, embora implementações que fogem ao estilo de projeto CMOS possam estabelecer novas fontes de consumo estático. Circuitos CMOS, teoricamente, não consomem energia

[1]Em geral, o coeficiente β é citado como o "*tamanho do transistor*", por estar diretamente relacionado com a razão entre a área transversal e comprimento do canal do transistor.

estaticamente; as correntes que ocorrem em circuitos CMOS são decorrentes do chaveamento de estado dos transistores. Variações de temperaturas, contudo, provocam flutuações de corrente (*drifts*) nos circuitos, as quais geram correntes estáticas. Embora essas correntes não possam ser ignoradas *a priori*, elas o são por fugir às ações possíveis dentro de metodologias de projeto cuja tecnologia de implementação física é fixa. Adicionalmente, em circuitos digitais de alta velocidade, a potência dinâmica é milhões de vezes superior à estática.

2.2.2 Métricas para Circuitos de Baixa Potência

Embora pareça uma questão técnica de abordagem direta, a análise da eficiência energética de um sistema é controversa quanto à escolha das métricas. Diferentes unidades de medidas têm sido usadas para quantificar a eficiência dos sistemas quanto ao consumo de energia. A forma mais intuitiva e direta é a potência expressa na unidade *Watts*. Como uma métrica derivada, encontra-se o conceito de potência de pico, que estabelece o limite mínimo da fonte de energia do sistema, podendo ser estimados elementos como, características mínimas dos condutores e a situação de pior caso com relação à autonomia de sistemas baseados em baterias. Contudo, para sistemas de alta velocidade, cujo padrão de comportamento não possui uma dinâmica que possa ser considerada regular, como microprocessadores ou co-processadores de controle, a relação entre energia consumida por atividade ganha uma conotação particularmente interessante. A métrica nesse caso é o consumo de energia em *Joules* por atividade. A métrica energia consumida por ciclo de relógio (*clock*), cuja unidade é $\mu W/\text{MHz}$, presta-se para avaliar microprocessadores, denotando a eficiência energética do circuito integrado. Tal métrica, contudo, não se presta à análise comparativa de dois processadores diferentes, não se mostrando útil à exploração do espaço de projeto. Para esse intento, o conceito de atividade deve subir em abstração, saindo de ciclo de relógio (*clock*) para instrução. A métrica adotada passa a ser unidades de consumo de energia por instrução, cuja unidade freqüentemente usada é o $\mu W/\text{MIPS}$. Essa métrica permite a comparação entre processadores de diferentes padrões de desempenho, mas com o mesmo conjunto de instruções, como é o caso de controladores e DSPs de uma mesma família. Devido à normalização, pode-se comparar os dispositivos apenas quanto à eficiência energética, independente do *clock* de operação ou desempenho. Para comparar processadores de arquiteturas diferentes, a métrica consumo de energia por instrução é inadequada. Para tanto, faz-se necessário comparar os processadores com base em tempos de computação de rotinas padrão (*benchmarks*), escritas em linguagens de alto nível. Os tempos de execução de tais rotinas definem a métrica de

tempo SPEC [111]. A métrica consumo de energia por SPEC é normalmente descrita pela unidade μW/SPEC. Métricas como consumo de energia por instrução e consumo de energia por SPEC partem de um modelo de máquina com execução regular de instruções, ou seja, máquinas cujo comportamento interno é independente do contexto de execução do código. Arquiteturas *simplescalar* e *superscalar* têm comportamento dependente (parada de *pipeline* e operação de despacho de instruções), o que estabelece uma execução não regular. Arquiteturas de alto desempenho, que implementam estratégias irregulares de execução de instruções, como arquiteturas *simplescalar* (um *pipeline*), *VLIW* e *superscalar*(múltiplos *pipelines*) acrescentam custos energéticos na instrução, a depender do contexto de execução [100]. Devido a isso, estimativas de consumo em arquiteturas de alto desempenho apresentam-se como um desafio de pesquisa, havendo, contudo, trabalhos como [12] e [13] relativos ao tema.

Uma métrica bastante difundida é o produto consumo de energia versus atraso [111]. Essa métrica é empregada para medir o mérito de um dado estilo de implementação de *hardware*. Quanto menor for o produto consumo de energia versus atraso, melhor será a qualidade do projeto. Esta métrica é particularmente interessante para avaliação de sistemas que operam com alimentação por bateria. Mínimo custo, em termos de consumo de energia e tempo, garantem máxima autonomia do sistema.

Por fim, a escolha de uma métrica depende do tipo de análise empregada e do domínio de aplicação. Em sistemas complexos, uma única métrica pode ser útil para um dado propósito, mas certamente não será eficiente para todos. A pesquisa de técnicas e metodologias para análise e otimização de consumo é certamente uma área estratégica.

2.2.3 Princípios Básicos de Projetos de Baixa Potência

Projetos com restrições de consumo devem ser abordados buscando redução do consumo de energia e boa relação custo-benefício. A abordagem centrada na redução de consumo busca a identificação (análise) e eliminação (otimização) dos desperdícios de energia. Cada decisão deve focar a relação custo-benefíco gerada, o que exige mecanismos de exploração de espaço de projeto e experiência do projetista. Diversas técnicas e metodologias de projeto de *hardware* de baixo consumo foram apresentadas, todas voltadas para domínios e situações bem específicas, não existindo uma abordagem genérica que satisfaça todas as possíveis situações de projeto [111]. Para uma abordagem eficiente, todos os ângulos da especificação devem ser considerados. Considerações de baixo consumo devem ser aplicadas em todos os planos de abstração e em cada etapa da metodologia de projeto. A redução do consumo *per se* afeta parâmetros

cruciais do sistema tais como: velocidade, área de silício, confiabilidade, re-usabilidade, testabilidade e complexidade de projeto. Por isso, é fundamental a realização de estimativas de potência já nas primeiras etapas de projeto. Nas subseções seguintes, serão apresentadas as técnicas básicas para a otimização de consumo de energia em *hardware*.

Redução da Tensão de Chaveamento

Como já exposto, a potência dinâmica é responsável por, praticamente, todo o consumo do sistema. A Equação 2.1 é regida por três parâmetros, sendo um referente à tecnologia de materiais (capacitância) e dois a fatores de implementação (tensão e freqüência). Devido ao efeito quadrático da tensão, sua redução implica em reduções dramáticas no consumo de energia. Por esse motivo, a redução da tensão de operação dos circuitos críticos é o método mais facilmente implementável. Existem, contudo, algumas considerações quanto à redução da tensão de alimentação, sendo as mais importantes: (i) há queda de desempenho com a redução da tensão de operação em circuitos CMOS; (ii) há redução da imunidade ao ruído; e (iii) faz-se necessário *hardware* adicional para interfacear os circuitos de diferentes tensões de alimentação.

Redução da Capacitância

A redução das capacitâncias parasitas é sempre um bom caminho para promover tanto o aumento do desempenho, como a redução do consumo de energia do sistema. No entanto, a redução arbitrária da capacitância, isoladamente, pode não alcançar os resultados desejados em termos de redução de consumo de energia. O que se deve buscar é a redução do produto capacitância-freqüência de chaveamento do componente. Sinais de alta freqüência devem ser roteados por circuitos de baixa capacitância. A redução das capacitâncias parasitas pode ser realizada em diferentes instâncias, seja na tecnologia de material, na tecnologia de processo, no mapeamento físico do projeto (*floorplanning*, *placement* e *route*), ou mesmo alterando o algoritmo do sistema, pois implementações diferentes de um mesmo comportamento geram *hardware* diferenciados.

Redução da Freqüência de Chaveamento

Para efeito de otimização de consumo, a redução da freqüência de chaveamento gera os mesmos resultados da redução das capacitâncias. Deve-se ressaltar que o que se busca é balancear o produto capacitância-freqüência. O desafio básico é localizar circuitos com altas capacitâncias parasitas que sejam passíveis de redução de freqüência de operação sem grande impacto sobre os demais parâmetros de projeto. Como um efeito adicional,

a redução seletiva da freqüência aumenta a confiabilidade da pastilha de silício, uma vez que certos mecanismos de falha estão associados à freqüência de operação. Um método de grande eficiência na redução de chaveamento é a exploração de alternativas de implementação com baixa incidência de transições nos circuitos. Similarmente à redução de capacitância, a exploração de diferentes algoritmos computacionais pode gerar diferentes taxas de chaveamento nos circuitos sem a necessidade de mudanças nos circuitos de *clock*. Adicionalmente, métodos de codificação numérica podem prover um grande impacto sobre a redução da freqüência de chaveamento dos circuitos [111].

Redução das Correntes Estáticas e de Fuga

Parâmetros referentes às correntes de fuga não são, em geral, úteis aos projetistas de sistemas digitais, porque eles têm pouquíssimo controle sobre esses parâmetros. O consumo por corrente de fuga ganha importância em modos de operação nos quais o sistema opera com baixas freqüências ou em *sleep modes*, onde atividades dinâmicas são suprimidas. A redução das correntes de fuga é alcançada pela aplicação de técnicas específicas nas etapas de projeto de mais baixo nível: otimização do processo de fabricação e circuitos. Circuitos de memória, devido à sua alta densidade, são particularmente suceptíveis às correntes de fuga. A corrente estática pode ser reduzida pelo re-dimensionamneto do tamanho dos transistores e disposição de circuitos (*layout*). Módulos de circuitos que consomem energia estaticamente podem ser desligados quando não estiverem em uso. Em alguns casos, o consumo estático depende do estado lógico da saída dos circuitos. Por vezes, a mudança de lógica positiva para lógica negativa é impactante sobre o consumo estático de energia. A implementação de mecanismos de gerenciamento dos modos de operação é crucial no gerenciamento do consumo estático, a identificação dos modos de operação nas especificações de alto nível é de grande importância na definição dos mecanismos de gerenciamento.

2.3 Consumo de Energia em Sistemas Embutidos

Do ponto de vista da descrição estrutural, um sistema embutido pode ser representado por um ou mais processadores associados a sistemas de *hardware* específicos, operando como co-processadores. Isso conduz à existência de dois elementos básicos de consumo: o *hardware* e o *software*. O consumo das vias de conexão será tratado como consumo de *hardware*, embora seja possível inferir componentes de consumo devido ao *software*, visto que o padrão de acesso ao barramento é regido pelo *software*. Nas próximas seções, será feita uma breve apresentação desses componentes de consumo e dos trabalhos

relativos a esses temas.

2.3.1 Consumo Devido ao *Software*

O consumo devido ao *software* advém do fato de que, a despeito das otimizações implementadas no projeto de seu *hardware*, o consumo de um processador é um padrão dinâmico regido pelo *software* em execução. Isso atribui à otimização do *software* mais um papel crucial em sistemas embutidos: a minimização do consumo de energia. O consumo devido ao *software* é descrito com base em um modelo de consumo das instruções do processador. Instruções consomem energia basicamente por dois mecanismos: (i) durante a sua execução, a instrução gera uma seqüência de estados internos ao processador cujas transições impõem um padrão de consumo ao *hardware*, denominado *custo básico de instrução* (*Instruction Base Cost*)[52]; (ii) de acordo com os operandos, a instrução realiza mudanças nos registradores e acessos à memória, implicando em chaveamentos de circuitos internos e barramentos, ou seja, consumo dinâmico. Adicionalmente, devido à influência dos operandos, o consumo básico sofre acréscimos em função das transições nos registradores e acessos à memória. Esses acréscimos podem ser modelados como fatores de sensibilidade da instrução ao contexto [78]. Os fatores de sensibilidade são os seguintes: número do registrador, valor do registrador, valor imediato presente na instrução, endereço do operando (sendo o operando uma posição de memória) e valor do operando (conteúdo da memória). Adicionalmente, estudos empíricos demonstram que existe uma relação aproximadamente linear entre o número de bits 1's, na palavra de dados do fator de sensibilidade, e o incremento de consumo [79]. Durante a execução do *software*, o consumo flutua em torno de um valor médio [79]. A Equação 2.4 modela a energia consumida por uma instrução.

$$E_i = b_i + \sum_i a_{i,j} \times n_{i,j} \tag{2.4}$$

Onde E_i é o consumo associado à instrução i, b_i é o custo de energia básico da instância de instrução i, $a_{i,j}$ é o coeficiente de sensibilidade da instrução i ao fator j e $n_{i,j}$ é o número de 1's presente no fator j.

Através da análise do fluxo do *software*, é possível identificar seções de alto consumo. A manipulação dessas seções, seja por otimização de código, seja pela implementação da seção em *hardware*, pode reduzir o consumo do sistema [93]. A caracterização do consumo de cada instrução é determinada executando a mesma instrução em um laço infinito e medindo o consumo do processador [78]. Infelizmente, esse procedimento não captura o consumo decorrente dos chaveamentos internos do processador, que ocorrem

devido à combinação de duas instruções consecutivas. Estudos empíricos demonstram que o consumo de um par de instruções consecutivas é diferente da soma dos consumos individuais [79]. Essa diferença é aceita como um custo de circuitos (*circuit state overhead*), devido à transição interna entre o último estado interno, referente à primeira instrução, e o primeiro estado interno, referente à segunda instrução. Trabalhos como os de Tiwari *et al* [98] relatam a relação entre variações da distância de *hamming* entre *opcodes* e o consumo de energia. Esse consumo é normalmente denominado: *consumo inter-instrução* (*inter-instruction cost*) [99]. O consumo associado à um programa p pode ser modelado pela Equação 2.5.

$$E_p = \sum_i E_i + \sum_{i,j} (O_{i,j} \times N_{i,j}) \tag{2.5}$$

Onde E_p é o consumo de energia associado ao programa p, E_i é o consumo associado à instrução i, $O_{i,j}$ é o consumo inter-instrução associado ao par de instruções i e j, e $N_{i,j}$ é o número de ocorrências consecutivas das instruções i e j.

O Fator *Pipeline*

Todos os pontos discutidos até agora são intuitivos, desde que o processador execute apenas uma instrução por vez. Em processadores que possuem *pipeline*, embora o conceito seja menos intuitivo, o consumo por instrução mantém o mesmo significado básico. Assumindo que cada estágio é modelado por uma fonte de consumo e ignorando, por hora, o efeito de conflitos de *pipeline*, observa-se que o consumo de uma instrução é o somatório do consumo dos estágios pelos quais ela transitou. Seja E_{jIk} o consumo de energia do estágio j do *pipeline* quando executando a instrução Ik, assumindo a seqüência de instruções I_1, I_2, I_3, I_4 e I_5, em um *pipeline* de cinco estágios. Com o *pipeline* cheio, o consumo de energia após um ciclo de *clock* será $E_{cycle} = E_{5I1} + E_{4I2} + E_{3I3} + E_{2I4} + E_{1I5}$. Uma instrução pode ser considerada executada quando percorre todo o *pipeline*. Sendo assim, o consumo básico de uma instrução deve ser a soma de cada fonte de consumo (estágio) utilizada pela instrução, ou seja, $I_{Kcons} = \sum_j E_{jIk}$[99]. O consumo inter-instrução pode ser modelado como distribuído ao longo do *pipeline*. Da mesma forma que há uma fusão da execução das instruções no tempo, existe uma fusão do consumo. O equilíbrio dessa fusão é rompido por eventos de conflito no *pipeline* (dados, estrutura e controle). Isso introduz uma dependência de contexto para o consumo. Para um contexto em que não há conflitos no *pipeline*, o modelo de consumo recai no modelo de consumo de instruções aplicáveis a processadores sem *pipeline*. A depender do evento de conflito, é acrescido um consumo adicional devido

ao contexto. Todos esses pontos são resumidos na Equação 2.6, proposta em [50].

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_h E_h \quad (2.6)$$

Onde E_p é o consumo de energia associado ao programa p, B_i é o consumo básico associado à instrução i, N_i é o número de instâncias da instrução i, $O_{i,j}$ é o consumo inter-instrução associado ao par de instruções i e j, $N_{i,j}$ é o número de ocorrências consecutivas das instruções i e j, e E_h é o consumo associado ao contexto de execução h (paradas do *pipeline* e falta de *cache*, por exemplo).

2.3.2 Estimativa em *Software*

Há duas abordagens básicas para análise de consumo de energia devido ao *software*: as baseadas em simulação de instruções e as baseadas em simulação de *hardware* [78]. A simulação de instruções consiste em caracterizar o modelo de consumo do processador de acordo com o consumo associado à suas instruções. Esse modelo alimenta um simulador de instruções e a análise é realizada com base nos padrões de saída desse simulador. Na abordagem por simulação de *hardware*, o simulador é baseado em uma descrição do *hardware* do processador. O modelo de consumo é construído com base no modelo matemático do circuito (mais baixo nível) e/ou descrições de mais alto nível (RTL- *Register Transfer Level*). A simulação de *hardware*, embora apresente resultados mais exatos, possui alta complexidade de implementação e alto custo computacional. A esses fatores são incorporados à baixa flexibilidade do modelo de energia e à necessidade de acesso a detalhes da tecnologia de implementação do *hardware*, que constituem propriedades intelectuais. Trabalhos como os de Brooks e Vijaykrishnan [14, 110, 107] têm aberto novas perspectivas na direção de modelos RTL e baseados em arquiteturas. Infelizmente, mesmo esses modelos parecem pouco flexíveis para a modelagem de uma grande variedade de dispositivos [29]. A simulação de instruções traz como vantagem o fato de ser baseada em um modelo de consumo que pode ser caracterizado a partir de medidas elétricas, realizadas diretamente no dispositivo de interesse. Para tanto, técnicas como a apresentada em [78] estão sendo amadurecidas. Desde o trabalho seminal de Tiwari *et al* [99], alguns dos mais significativos trabalhos na análise de consumo de processadores, tais como [52, 98, 100], são baseados na caracterização do consumo de instruções. Um desafio associado ao modelo de instruções é reduzir sua complexidade em função do efeito inter-instrução, em outras palavras, a explosão de parâmetros de consumo inter-instrução, gerados a partir da combinação de todas as instruções de uma arquitetura. Esta questão é particularmente severa em arquiteturas *VLIW* e *Superscalar*.

Trabalhos como os de Bona *et al* [13, 12], referentes à arquiteturas VLIW, propõem o agrupamento de instruções para a geração de um modelo inter-instruções regido pela combinação de grupos e não de instruções. A técnica baseia-se em colocar em um mesmo agrupamento as instruções que apresentarem consumo de energia próximos, inferindo um mesmo consumo inter-instrução dentro do agrupamento, e estabelecendo valores inter-agrupamento por meio de medidas. Russel *et al* [87] propuseram um modelo que considera o valor médio do consumo de energia invariante dentro de uma mesma arquitetura de conjunto de instruções (ISA- *Instruction Set Architecture*). O modelo observa que, dada uma classe de processadores, o consumo de energia por instrução mostra apenas uma pequena variação. Em última análise, o modelo de Russel assume que o consumo de energia tem maior relação com a estrutura de execução do código, em termos de localidade de execução (quantidade de laços), do que com as características de consumo do conjunto de instruções. Sarta *et al* [89] propõem um modelamento estatístico, ou seja, o modelo de consumo seria definido pelo consumo médio do conjunto de instruções da arquitetura.

Tal modelo mostrou-se adequado apenas para domínios de aplicação em que o comportamento do código é bastante regular. Klass *et al* [47] propuseram um modelo no qual o efeito inter-instrução é inferido como o valor do consumo inter-instrução entre uma instrução genérica e uma instrução NOP (*No OPeration*). Esse modelo foi denominado, pelo próprio Klass, como *NOP model*. Sendo baseado na proposição de que o custo inter-instrução é menos dependente da combinação de instruções do que do fato das instruções serem do mesmo tipo ou não. Klass construiu seu modelo mapeando o custo inter-instrução entre a instrução alvo e a instrução NOP, e generalizou o valor obtido como o valor inter-instrução da instrução alvo, não importando a instrução com a qual ela fizesse par. Klass demonstrou que, para o DSP 56800 da Motorola, esse modelo apresenta erros que variam de 1% a 8%. Outros pesquisadores, como Lee *at al* [51] e Naik *at al* [74], propuseram metodologias baseadas em pré-processamento e re-estruturação de código fonte, visando à redução do consumo do código executável. A conjunção de tais trabalhos com a avaliação de compiladores executada por Simunic *et al* [96] estabelece a necessidade de compiladores otimizados para consumo de energia. Simunic *et al* [96] analisaram um compilador C padrão para a arquitetura ARM, concluindo que os recursos de otimização padrão, presentes no compilador, resultavam em ganhos inferiores a 1%, enquanto otimizações manuais do código apresentaram ganhos de 90%. Desde o trabalho seminal de Tiwari *et al* [99], diversas técnicas de compilação otimizada para consumo foram propostas. Trabalhos como os de Su *et al* [95], Yang *et al* [109], Petrov *et al* [82] e Nevine *et al* [4] exploram tanto aspectos clássicos da

otimização como políticas de gerenciamento de recurso da memória *cache*. Métodos baseados em técnicas de escalonamento também foram propostos por Lee em [51]. A abordagem por escalonamento tenta minimizar o cheavamento nos barramentos pela escolha de pares de instruções. Trabalhos propostos por Parikh *et al* [80] e Tavares *et al* [97] buscaram modificar a lista de execução realizando um balanço entre consumo e velocidade. Adicionalmente, modelos de análise de consumo, mediante gráficos de fluxo de programa, foram propostos por Chatzigeorgiou *et al* [21].

No contexto do paradigma *Hardware/Software Co-Design*, Stitt *et al* [93, 92] demonstraram que a migração *software-hardware* promove reduções de até 79% do consumo total de energia. Tais conclusões estabelecem a importância de estimadores do consumo de energia devido ao *software* na análise da migração *software-hardware*. Adicionalmente, Bastos *et al* [11] demonstraram que, para micro-controladores de uso massivo, reduções de consumo da ordem de 53% são atingidas migrando fragmentos críticos de código entre módulos de memórias internas e externas. Mecanismo de estimativas de consumo avaliando códigos fontes de alto nível foram apresentados nos trabalhos de Julien *et al* [43, 44] e Senn *et al* [90]. Em seu trabalho, Senn *et al* implementaram estimativas de consumo de energia e potência em código fonte C, integrando o ambiente *code composer*[2] ao seu arcabouço de análise [90], o que estabeleceu interessantes possibilidades de aplicação. Trabalhos como os de Stitt e Bastos [94, 11] estabelecem a importância não apenas de estimadores de consumo total de energia, mas também de avaliadores da distribuição de consumo nas estruturas de controle do código.

2.3.3 Consumo Devido ao *Hardware*

Em contraste com o consumo devido ao *software*, o consumo de *hardware* é definido por sua descrição e tecnologia de implementação. Devido à isso, a otimização de projeto é regida pelos fatores apresentados na Seção 2.2.3. A abordagem básica aplicada a sistemas de *hardware*, por décadas, tem sido a simulação. A maioria dos simuladores operam considerando modelos matemáticos que descrevem as leis físicas que estabelecem o consumo. Infelizmente, o custo computacional associado dessa abordagem torna-se insustentável em sistemas realmente complexos. Em geral, simulações em baixo nível, baseadas em componentes, oferecem excelente exatidão. Simuladores de circuitos, tais como o SPICE (*Simulation Program with IC Emphasis*), são excelentes quanto à exatidão de seus resultados, mas não são viáveis para aplicação na análise de uma pastilha

[2]Ambiente de implementação e análise de códigos para DSPs Texas, largamente empregado na indústria.

de silício inteira [111]. Para abordar essa questão, é necessário aplicar o conceito de caracterização de sistema. A caracterização consiste na utilização dos resultados da análise de baixo nível (componentes) para construção de um modelo de potência de mais alto nível. Estabelecem-se, assim, camadas de abstração capazes de comportar diferentes modelos de potência[3]. Uma vez que nenhum nível de abstração é adequado para todas as etapas de projeto, uma metodologia *Top-Down* de estimativa, análise, refinamento e verificação é aconselhável e, normalmente, usada. Pode-se iniciar como uma simulação do comportamento do *hardware* baseado em um modelo comportamental para dada tecnologia, gerando uma estimativa inicial. Quando o projeto tiver uma resolução de portas lógicas, uma outra simulação com modelos de potência de portas e conexões pode ser realizada, refinando a estimativa inicial. Se a estimativa denunciar um não cumprimento das restrições de projeto, o projeto deve ser refinado nos pontos apontados pela análise das estimativas, atacando aspectos citados na Seção 2.2.3. Este mecanismo de refinamento e verificação deve continuar até que as restrições sejam atendidas.

Uma abordagem sistêmica implica no escalonamento de diversos níveis de abstração, inclusive a adoção de modelos de consumo de *software* para caracterizar componentes como micro-controladores e DSPs. A Figura 2.2 ilustra a variação da exatidão e demanda por recursos computacionais, em função do nível de abstração do modelo de potência.

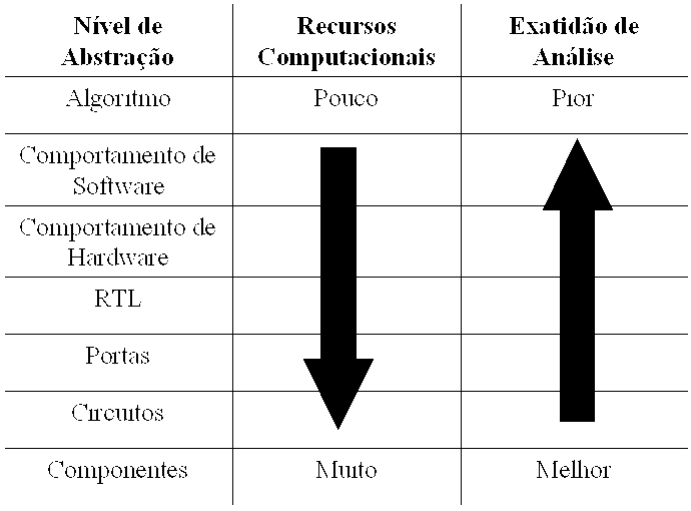

Figura 2.2: Variação da exatidão e demanda por recursos computacionais, em função do nível de abstração do modelo de potência

[3]Modelos de consumo de transistores constroem modelos de consumo de portas, que constroem modelos de consumo de *flip-flop* e assim por diante.

2.3.4 Estimativa em *Hardware*

Modelos de estimativas em portas, circuitos, *layout* físico e em descrições comportamentais estão presentes na literatura [32, 49, 76] já há alguns anos. Segundo Devadas *et al* [28], as abordagens de alto nível para a estimativa de potência em *hardware* podem ser classificadas em função de seu nível de abstração. No nível RTL, existem duas classes: abordagens analíticas e abordagens empíricas. Os métodos analíticos objetivam relacionar a potência com as capacitâncias e atividades de chaveamento das vias de conexão entre os circuitos. Dessa abordagem geram-se modelos baseados em *complexidades* e *atividades*. Os modelos de complexidade modelam a complexidade do circuito em termos de portas como uma medida da capacitância [28]. Quanto mais complexo for o circuito, maior a capacitância associada. Os modelos de atividade são baseados na medida do consumo a partir do conceito de entropia. A potência média é estimada como o produto da área e da entropia. Nesse contexto, a área representa um parâmetro relacionado à medida indireta da capacitância do circuito, sendo aceito que há um fator de distribuição da capacitância na área, ou capacitância específica. O termo entropia, nesse contexto, modela a atividade de chaveamento da mesma forma que é usado para modelar informação na teoria da informação [32]. Métodos empíricos são baseados em medidas da potência de implementações prévias, criando-se macro-modelos associados às topologias e estruturas do circuito. Os métodos empíricos podem ser divididos em: modelos de atividade fixa e modelos sensíveis à atividade [28]. Os modelos de atividade fixa desconsideram a influência das atividades de dados no consumo de energia. Os modelos sensíveis à atividade caracterizam estatisticamente o consumo devido à atividade de dados e de controle.

Para análises baseadas em descrições comportamentais, as técnicas são centradas na predição dinâmica e estática de atividades. O objetivo da predição de atividade estática é estimar a freqüência de acesso a diferentes recursos de *hardware*. Isso é feito por meio da análise estática da descrição comportamental. A predição dinâmica de atividades é baseada no perfil dinâmico do sistema. Mecanismos de simulação realizam o levantamento desse perfil. Uma característica básica da estimativa de potência, independente do nível de abstração de seu modelo, consiste na revelação da forte conexão entre atividade de circuitos e consumo médio. Com base nessa dependência da atividade (chaveamento), é enfatizado por Najm em [75] que o processo de estimativa de potência depende do padrão de excitação de entrada do circuito. Essa dependência pode ser forte ou fraca, em função do modelo de estimativa. Basicamente, as abordagens baseadas em simulação determinísticas possuem forte dependência. A principal vantagem da simulação é a exatidão; contudo, para obter-se uma estimativa de boa

exatidão, faz-se necessário uma modelagem detalhada do conjunto sistema-ambiente, de forma a construir um padrão de entrada representativo. Esta necessidade de detalhes faz com que essa abordagem seja particularmente cara em custos computacionais e de implementação. Além do mais, uma abordagem de baixo nível, baseada puramente em simulação, é quase impraticável devido à complexidade dos sistemas. Uma alternativa é a construção de processos de estimativa fracamente dependentes do padrão de entrada, usando como entrada um modelo de probabilidades de comportamento. Nesse caso, a estimativa depende das probabilidades fornecidas pelo projetista. As medidas de probabilidades compreendem frações do ciclo de *clock* (sistemas síncronos) em que os sinais de entrada transitam de um estado a outro. Essas informações são usadas para estimar o quão freqüentes são as transições internas do sistema e, conseqüentemente, a energia consumida. A Figura 2.3 ilustra a diferença básica da abordagem por pura simulação e a abordagem probabilística, também chamada *simulação probabilística* [75]. A etapa de inferência de probabilidade gera, como saída, a chamada forma de onda de probabilidade [75]. A Figura 2.4 descreve a probabilidade do sinal estar em nível alto (linhas horizontais), como também os instantes e probabilidades de transição (setas verticais).

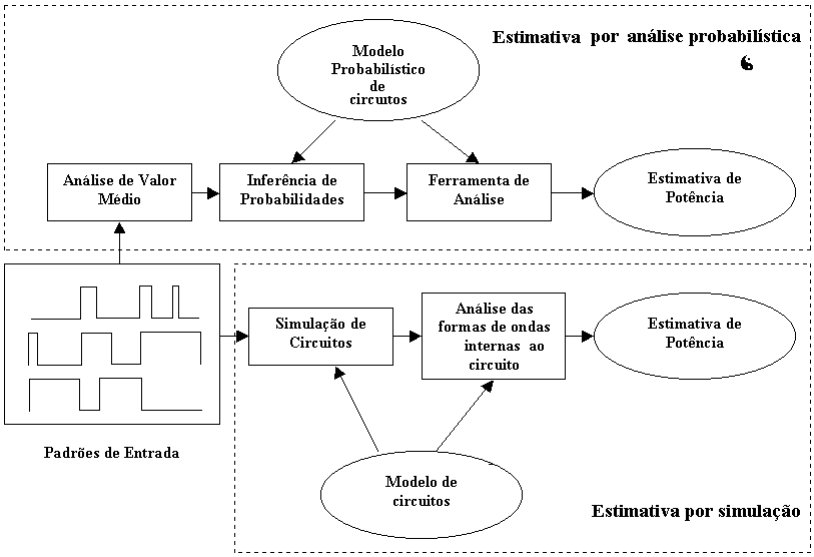

Figura 2.3: Estimativas baseadas em Simulação e Análise Probabilística

Uma outra abordagem é a chamada técnica estatística, no qual um padrão aleatório é introduzido na entrada e a simulação é realizada ciclicamente até que um padrão de consumo seja caracterizado. Entenda-se padrão de consumo como um comportamento em que o valor médio do consumo está bem definido. Isso é de particular importância no dimensionamento do encapsulamento do componente físico, uma vez que o calor

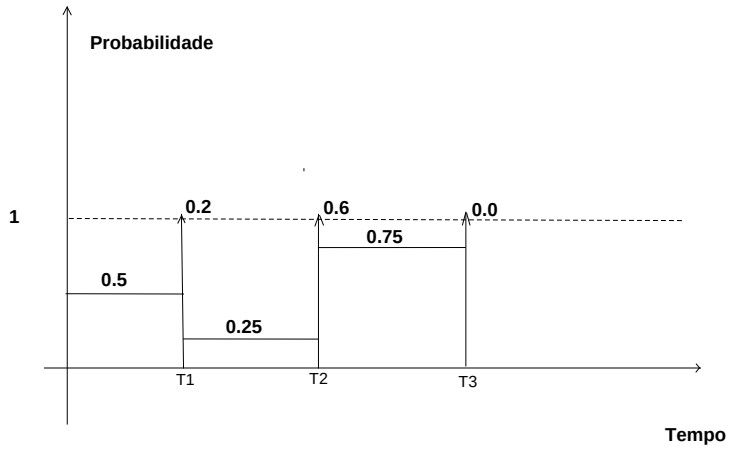

Figura 2.4: Exemplo de forma de onda de probabilidade

gerado por uma pastilha de silício está diretamente relacionado ao valor médio da energia consumida [75].

Mehra [66] apresenta mecanismos para a estimativa de potência em aplicações de tempo real baseadas em DSPs e ASICs. Mehra usa descrições comportamentais e demonstra que o consumo de energia devido à operações sobre os dados (fluxo de dados) pode ser estimado pela análise do CDFG (*Control Data Flow Graph*) associado. O número de acessos (operações) a cada componente é exatamente o número de nós do CDFG. A caracterização dos nós, juntamente com a análise do grafo, fornece uma estimativa do consumo de energia do circuito. Lloyd *et al* [55] e Muragavel *et al* [72, 73] mostram o uso de modelos em redes de Petri, suportando, assim, mecanismos formais de análise. O trabalho de Lloyd *et al* demonstrou a aplicação do conceito de análise de invariantes na estimativa de potência em circuitos assíncronos. Muragavel *et al* propuseram uma extensão semântica das redes de Petri coloridas denominada HCHPN (*Hierarchical Colored Hardware Petri net*) para o modelamento de circuitos combinacionais [72] e seqüenciais [73]. Estimativas de tempo de propagação e consumo de energia são capturados a partir da simulação do circuito modelado como uma HCHPN. Tal simulação é realiza em uma versão modificada da ferramenta Design/CPN. Devido ao nível de abstração do modelo e às técnicas de simulação do Design/CPN, foram registradas velocidades de simulação 46 vezes superiores às obtidas na ferramenta PowerMill [72].

Givargis *et al* [102, 103] apresentaram uma abordagem que associa o conceito de instrução a *hardwares* que não são processadores, promovendo um modelo de potência com elevado nível de abstração. O *hardware* é modelado como um processador virtual, cujas instruções são descritas como "*... ações que, coletivamente com outras ações,*

descrevem o conjunto de comportamentos possíveis de um circuito. " [103]. Em outras palavras, é introduzido um modelo de descrição algorítmica idêntico aos usados para descrever consumo de *software*. O fluxo de ação é definido como traços de execução (*traces*) e analisados por simulação. Na abordagem apresentada em [102], Givargis *et al* apresentam um modelo baseado nesse conceito de instrução, em que o circuito (*Core*) é caracterizado pela construção de macro-modelos relativos às instruções. Esses macro-modelos são caracterizados em descrições de circuitos lógicos (portas). A caracterização dos macro-modelos (instruções) é, então, armazenada em uma tabela; cada instrução, por sua vez, possui sua própria tabela onde estão seus padrões de consumo em função de parâmetros como modo de operação do circuito e dependência dos padrões de entrada, que foram previamente levantados por uma abordagem estatística. O modelo do circuito é representado dentro de um modelo de sistema por um objeto, no qual os métodos descrevem o comportamento das instruções. O modelo de circuito, portanto, é descrito de forma comportamental e implementado em uma linguagem orientada a objeto, como Java ou C++. Dessa forma, um sistema inteiro pode ser criado a partir do *instanciamento* de circuitos previamente caracterizados. A análise da seqüência de execução das instruções (invocação dos métodos) estabelece um traço de consumo (*traces*). Diferentes tecnologias podem ser analisadas por esta abordagem com grande flexibilidade, bastando, para tanto, atualizar o modelo com novas tabelas de caracterização.

No contexto da análise de *hardware* de processadores, o trabalho de Brooks *et al* propõe uma infra-estrutura de modelamento de consumo de energia por descrições da micro-arquitetura do processador. Esse trabalho gerou uma ferramenta denominada *Wattch*. Ye *et al* fazem a mesma proposta denominando sua infra-estrutura de *SimplePower*. Ambas aplicam a infra-estrutura de modelagem e simulação de micro-arquiteturas denominada *SimpleScalar* [7]. Ao longo dos últimos dez anos, a SimpleScalar tornou-se largamente empregada pela comunidade que pesquisa arquiteturas de processadores. Por operar sobre modelos *ad hoc* de simulação de micro-arquiteturas, a SimpleScalar é pouco flexível, possuindo sete simuladores associados a sete níveis de detalhamento da micro-arquitetura do processador de interesse.

2.4 Linguagens de Descrição de Arquitetura

Com a crescente demanda por otimizações extremas nos sistemas embutidos, surge a necessidade de arcabouços de projeto voltados à criação de processadores específicos, não mais centrados em domínios, mas sim em aplicações específicas do sistema alvo. Dado

um sistema em desenvolvimento, o arcabouço de projeto deve contemplar a criação de um processador otimizado para o sistema. O desafio de criar tal arcabouço tem sido o objeto das pesquisas ligadas às linguagens de descrição de arquiteturas (*ADL- Architecture Description Language*). Diferentemente das linguagens de descrição de *hardware*, as ADLs operam com base na descrição de elementos arquiteturais do processador, a partir de macro-modelos de unidades funcionais, barramentos e *pipeline*. Mais do que a síntese do *hardware*, as ADLs objetivam a criação automática de ferramentas, tais como compiladores e simuladores. A utilização de tais recursos se insere no campo da exploração do espaço de projeto (*DSE- Design Space Exploration*). As ADLs buscam promover a rápida avaliação do projeto em andamento. Dependendo do objetivo da ADL, ela pode ter uma estrutura de linguagem que permite diversos focos descritivos do processador. Tais focos podem ser centrados no *hardware* (micro-arquitetura e organização de memória) e/ou no *software* (conjunto de instruções). De acordo com o foco da ADL, ela é tradicionalmente classificada em três categorias: ADLs estruturais, comportamentais e mistas [83]. ADLs estruturais descrevem o processador com base em detalhes de componentes de *hardware*. Tais ADLs são adequadas à síntese de *hardware* e geração de simuladores de *hardware*. Em contraste, as ADLs comportamentais são focadas na descrição do processador como um conjunto de instruções. Elas se mostram adequadas para a geração de compiladores e simuladores. Por fim, as ADLs mistas buscam incorporar os adjetivos das duas outras categorias. Tomyama *et al* [101] propõem ainda classificar as ADLs em cinco categorias, de acordo com seus objetivos: ADLs orientadas (i) à síntese, (ii) à geração de compiladores, (iii) à geração de simuladores, (iv) à validação e (v) a outras abordagens.

2.4.1 ADLs Orientadas à Síntese

A linguagem MIMOLA é um exemplo de ADL orientada à síntese de *hardware*, muito embora ela esteja mais próxima de uma linguagem de descrição de *hardware* (HDL- *Hardware Description Language*) do que de uma ADL [101]. MIMOLA descreve o processador como uma lista de conexões descritas no nível de transferência de registradores (*RTL- Register Transfer Level*). Sendo assim, MIMOLA é uma linguagem estrutural de baixo nível de abstração, não suportando descrições da arquitetura de instruções. Embora MIMOLA tenha capacidade de síntese de circuitos e simulação em RTL, ela não permite uma descrição explícita de *pipelines* e de circuitos de tratamento de conflitos de *pipelines*.

UDL/I (*Unified Design Language for Integrated Circuit*) [101] é uma linguagem de descrição de *hardware* incorporada ao ambiente COACH [37], que permite tanto a

síntese de *hardware* como a geração de compiladores e simuladores. O processador é descrito em UDL/I em nível RTL. Sobre essa descrição, é aplicada a técnica proposta por Akaboshi *et al* [5] para extração do conjunto de instruções. O ambiente COACH opera com descrições de processadores RISC simples e não suporta explicitamente a descrição de *pipeline* e circuitos de tratamento de conflitos. A linguagem TIE (*Tensilica Instruction Extension Language*) [85] é uma ADL comercial desenvolvida para permitir extensões dos processadores Tensilica-Xtensa [1], visando a aplicações específicas. A micro-arquitetura básica é imutável e diz respeito à arquitetura Xtensa. A linguagem cumpre o papel apenas de otimizar a descrição original em termos de número de registradores e conjunto de instruções. Com base na especificação do processador em TIE, uma descrição em HDL sintetizável é gerada e as ferramentas de compilação e simulação do Xtensa são reconfiguradas.

2.4.2 ADLs Orientadas à Geração de Compiladores

Como exemplos de ADLs orientadas à geração de compiladores, pode-se citar: nML, ISDL e EXPRESSION. A nML [30] tem sido usada como ferramenta pela Cadence Design Systems para a geração de simuladores de conjuntos de instruções, montadores (*assemblers*) e montadores reversos (*disassemblers*). A nML objetiva principalmente a descrição de DSPs, de forma que o processador é representado com um conjunto de instruções. Vários modos de endereçamentos são suportados e mecanismos hierárquicos para descrição do conjunto de instruções estão presentes. A descrição do conjunto de instruções é realizada de forma que os conflitos entre instruções são implicitamente descritos, permitindo a geração automática de compiladores. A ISDL [35], suporta a síntese automática de compiladores e montadores. Adicionalmente, uma descrição ISDL pode ser convertida para Verilog[4] sintetizável. Devido a esta capacidade, é possível a realização de simulação em nível de cíclos de relógio. Similarmente à nML, a ISDL descreve o processador como um conjunto de instruções, especificando (i) o uso de recurso de *hardware*, (ii) o número de ciclos de relógio e (iii) o formato da linguagem *assembly*. A descrição de *pipeline* não é explicitamente suportada. EXPRESSION [36] é uma linguagem de descrição que opera sobre uma estrutura de gabaritos (*templates*) de forma a possibilitar uma boa diversidade de arquiteturas. O processo de descrição é separado em duas etapas. A primeira é de natureza estrutural, de forma que a micro-arquitetura do processador é descrita com um diagrama de blocos, por meio de um ambiente denominado VSAT [68]. A segunda é de natureza comportamental, na qual o

[4]Linguagem de descrição de *hardware* baseada na sintaxe do C.

conjunto de instruções é descrito em uma sintaxe similar à LISP. A geração de compiladores é realizada por um aplicativo denominado EXPRESS, que compila a descrição EXPRESSION gerando um compilador C. Identicamente, um aplicativo denominado SIMPLESS gera um simulador preciso em ciclos de relógio (*cicle-accurate simulator*). Adicionalmente, a EXPRESSION suporta a descrição de arquiteturas de memórias. A EXPRESSION, contudo, não suporta síntese de *hardware*. Adicionalmente, uma vez que os modelos de simulação são construídos com base em gabaritos, a flexibilidade está restrita à biblioteca de gabaritos da linguagem, como um *lego* cujas peças novas não podem ser concebidas pelo projetista.

2.4.3 ADLs Orientadas à Geração de Simuladores

As ADLs LISA, ArchC e BABEL são exemplos de linguagens orientadas à geração de simuladores. A linguagem LISA [113] descreve tanto o conjunto de instruções quanto a micro-arquitetura do processador. Recursos específicos para a descrição de paradas (*stall*) de *pipeline* estão presentes. A característica principal da descrição em LISA é o recurso de descrição de operações internas a instrução, de forma a modelar técnicas de *bypassing* para o *pipeline*. A LISA suporta ainda detecção de conflito de recursos. Essa linguagem integra a ferramenta comercial *Processor Designer*^{TM} da CoWare. Na infra-estrutura da ferramenta *Processor Designer*^{TM}, a descrição LISA pode ser convertida para uma descrição RTL em Verilog, VHDL ou SystemC[5], permitindo estimativas de potência a partir de uma simulação RTL da descrição de *hardware*.

A linguagem ArchC [84] foi desenvolvida pela Universidade de Campinas (Brasil). Similarmente à EXPRESSION, ArchC descreve o processador em duas etapas: descrição do conjunto de instruções e descrição da micro-arquitetura. A linguagem ArchC constrói simuladores com base em construtores SystemC. Dada uma descrição em ArchC, um pré-compilador instancia uma descrição SystemC, que, após ser compilada, dá origem a um simulador. Adicionalmente, ArchC suporta a geração de montadores, por reconfiguração automática da biblioteca GNU Binutils do montador do gcc [8]. A linguagem BABEL foi proposta por Mong *et al* [69] como um mecanismo de reconfiguração automática da infra-estrutura SimpleScalar [7] de modelagem e simulação de micro-arquiteturas.

[5]Conjunto de construtores de modelos de *hardware* baseados em bibliotecas de C++.

2.4.4 ADLs Orientadas à Validação

A linguagem AIDL [70] foi desenvolvida pela para a validação de processadores com arquitetura *superscalar* de alto desempenho. Embora Morimoto *et al* [70] tenham definido a AIDL como uma HDL, sua aplicação de recursos específicos para a descrição de arquiteturas *superscalar out-of-order* a caracteriza com uma ADLs. Sua característica principal é possibilitar a validação do comportamento de modelos de unidades de despacho seja *in-order* ou *out-of-order*. A validação é realizada por meio de um simulador AIDL específico. A AIDL não tem capacidade de geração de compiladores e simuladores.

O conceito de ADL permite a geração automática e multi-alvo de ferramentas de análise. Sendo assim, tal conceito estabelece possibilidades interessantes para a geração automática de ferramentas de análise de consumo devido ao *software*. Na presente revisão da literatura, não foi encontrado nenhum trabalho referente a ADLs aplicadas à geração de estimativas de consumo de energia.

2.5 Considerações Finais

Em sistemas embutidos, há uma forte necessidade de mecanismos de estimativa de consumo, tanto para os componentes de *hardware* como de *software*. O modelo de potência de *software* é na verdade a aplicação do conceito de macro-modelos de *hardware*. Cada instrução de uma arquitetura de processador possui um conjunto de *hardware* agregado, responsável por um elemento de consumo[6]. A caracterização dessas instruções corresponde à caracterização do macro-modelo de *hardware* associado. A vantagem inerente ao modelo de potência de *software* é que essa estrutura de *hardware* é representada pela instrução. Dessa forma, um modelo de potência de *software* pode ser construído sem qualquer inferência da implementação física do *hardware*. É necessário apenas um modelo de arquitetura de instruções e um conjunto de medidas.

Existe o problema da explosão do conjunto de medidas, devido ao fator de custo inter-instrução, que se apresenta como um desafio de pesquisa. Modelos iniciais para reduzir o conjunto de medidas foram propostos, seja pela redução do número de valores inter-instrução, seja pela formação de grupamentos (*Clusters*)[12], portanto, pela padronização do efeito inter-instrução[47]. Apesar dos primeiros trabalhos a terem surgido na primeira metade da década de 1990, o tema renasce com renovada importância frente à crescente integração e re-uso de soluções de *hardware*o que destaca a relevância

[6]Embora esses *hardwares* sejam compartilhados em sua maioria, seu efeito de consumo é alocado no tempo pela ação das instruções

de uma perspectiva de baixo consumo, centrada no *software* [53, 88]. Adicionalmente, a aplicação de ADLs para a especificação de processadores permite, a exemplo do que se faz de forma menos específica no *Processor Designer*^{TM}, a criação de infra-estruturas de otimização de ISAs de processadores focadas em consumo.

Em relação à estimativa de potência em *hardware*, formam-se cenários onde paradigmas centrados no re-uso irão privilegiar um modelo de potência cuja granularidade será de módulos de *hardware*, normalmente referidos como *Intelectual Properties*. Trabalhos como [102, 103] demonstram a viabilidade dessa perspectiva. Cria-se, no entanto, um novo espaço de projeto, o projeto das *Intelectual Properties*. Nesse espaço, a busca de soluções para melhores modelos de potência e estimativa continua. Adicionalmente, a aplicação de modelos computacionais baseados em *tokens* pode representar uma alternativa para a geração de modelos analíticos [55]. As rede de Petri definem um formalismo para descrição de modelos computacionais baseados em *tokens*, sendo elas a base formal dos modelos propostos nesta tese. Uma descrição sumária do universo das redes de Petri, e do formalismo empregado nelas, será apresentado no próximo capítulo.

Capítulo 3

Bases Formais do Modelo Proposto

3.1 Introdução

Redes de Petri são famílias de técnicas formais para o modelagem de sistemas [71]. As redes de Petri modelam ações e estados através de quatro tipos de entidades: lugares, transições, arcos e *tokens*. Os lugares são usados para representar estados ou condições, enquanto as transições representam ações ou eventos. Os arcos estabelecem a relação de dependência entre lugares (estados/condições) e transições (ações/eventos) e vice-versa. Os *tokens* são marcadores que identificam que lugares estão "ativos" no sistema, ou seja, a marcação dos *tokens* em uma rede de Petri caracteriza o conjunto de estados internos do sistema. Cada uma dessas entidades possui uma identidade gráfica: os lugares são representados por circunferências; as transições, por retângulos; os arcos, por segmentos orientados; e os *tokens*, por círculos pretos. A rede resultante desses grafos é denominada rede de Petri.

Na Figura 3.1, um processo paralelo é descrito por uma rede de Petri. Analisando a figura observa-se que: o arco A_1 liga o lugar P_1 (estado local 1) à transição T_1 (evento 1), demonstrando que o evento 1 requer o estado local 1, representado por um *token* no lugar P_1. Os arcos A_2 e A_3 conectam a transição T_1 aos lugares P_2 e P_3, descrevendo a dependência dos estados associados a P_2 e P_3 com o evento associado a T_1. É de fácil observação dois caminhos paralelos envolvendo os eventos associados a T_2 e T_3. A estrutura da rede descreve que o evento associado à T_4 está sujeito a co-existência dos estados associados a P_4 e P_5, ou seja, a presença de *tokens* em P_4 e P_5. Observa-se ainda que a rede descreve o processo de sincronização realizado pelo evento associado a T_4.

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

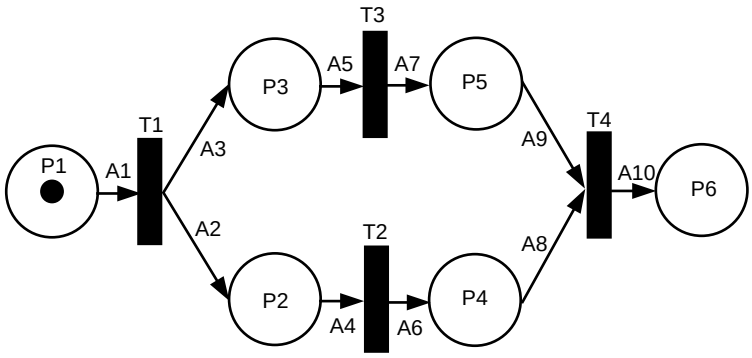

Figura 3.1: Exemplo de Rede de Petri

gerados pelas transições a cada mudança no sistema. O processo de consumo e geração dos *tokens* representa a dinâmica das mudanças de estados do sistema, a simulação gráfica desse processo é denominada *Token Game* [108]. O conjunto de estados internos caracterizado pela distribuição de *tokens* na rede define um *estado* do sistema, denominado *marcação da rede*. Cada uma das possíveis distribuições dos *tokens* nos diversos lugares ilustra estados do sistema alcançados pelo modelo.

Como o conceito de estado interno requer uma definição em termos do nível de abstração em que o sistema está sendo modelado, k *tokens* podem ser colocados nos lugares representando k itens de dados ou recursos associados ao estado interno do sistema [71]. Dessa forma, um peso deve ser associado ao arco para modelar a relação da sujeição com o número de *tokens*. O peso determina quantos *tokens* devem ser consumidos para a ocorrência dos eventos. O modelo de redes de Petri baseado em tais premissas, onde o tempo não é modelado e *tokens* representam marcações de estados ou abstrações de recursos, estabelece um tipo de rede denominada *Place/Transistion*, [58].

Formalmente, as redes de Petri (*PN-Petri Nets*) podem ser construídas a partir da seguinte definição [71]:

Definição 3.1 (Rede de Petri) .
Uma rede de Petri é uma quíntupla, $PN = (P, T, F, W, M_0)$*, onde:*
$P = \{p_1, p_2, p_3, p_4, ..., p_n\}$*, é um conjunto finito de lugares,*
$T = \{t_1, t_2, t_3, t_4, ..., t_n\}$*, é um conjunto finito de transições,*
$F \subseteq (P \times T) \cup (T \times P)$*, é um conjunto de arcos (relação de sujeições ou fluxo),*
$W : F \rightarrow \{1, 2, 3, 4, ..., n\}$*, é uma fução de atribuição de pesos aos arcos,*
$M_0 : P \rightarrow \{0, 1, 2, 3, 4, ..., n\}$*, é uma função de mapeamento da marcação inicial da rede,*
$P \cap T = \phi \ e \ P \cup T \neq \phi$*.*

Tabela 3.1: Semânticas para lugares e transições

Lugares de Entrada	Transições	Lugares de Saída
pré-condições	eventos	pós-condições
dados de entrada	passo e computação	dados e saída
sinal de entrada	processamento de sinal	sinal de saída
disponibilidade de recursos	tarefa	liberação de Recursos
condições	cláusulas lógicas	conclusões
buffers	processador	*buffers*

Uma rede de Petri sem uma marcação inicial específica ($PN = (P, T, F, W)$) é denotada por N, enquanto uma que tenha uma marcação inicial é denotada por (N, M_0). A depender do sistema modelado, os lugares e as transições podem ter significados diversos. A Tabela 3.1 explora algumas possibilidades segundo [71].

Os conceitos básicos das Rede de Petri foram primeiramente desenvolvidos como um modelo de análise de sistemas de comunicação. Extensões como as redes temporizadas - com modelos de tempo determinísticos e estocásticos [71]-, redes de alto nível de abstração - como as redes orientadas a objetos [104]- e as redes coloridas [41], possibilitaram a aplicação do formalismo de Petri a uma grande conjunto de aplicações. Devido aos seus mecanismos de análise de propriedades, as redes de Petri podem ainda ser definidas como um formalismo capaz de especificar sistemas, agregando instrumentos de verificação formal [71]. A capacidade de simulação, permite que as redes de Petri possam ser usadas como linguagem de especificação, gerando especificações executáveis.

3.2 Transições: Habilitação e Disparo

O comportamento de um sistema pode ser descrito em termos de seus possíveis estados internos e os processos de mudança desses estados. As redes de Petri representam tais processos por meio da mudança de sua marcação, ou seja, mudanças no mapeamento $M : P \rightarrow \{0, 1, 2, 3, 4, ..., n\}$. Essas mudanças seguem as chamadas *regras de habilitação de transições* [71]. Em última análise, uma regra de habilitação é uma lei que gerencia a ocorrência de um evento no sistema. Em um modelo em rede de Petri, existem três regras.

1) Uma transição t é habilitada, se cada lugar de entrada p da transição t é marcado

com no mínimo $\sum_i w_i(p,t)$. Onde $w_i(p,t)$ é uma função que mapeia a tupla (p,t) em um número natural que determina o número de *tokens* em p que habilita t a disparar. A função $w_i(p,t)$ é associada ao arco i que liga p e t, sendo denominada *peso do arco* i. Sendo assim, uma transição estará habilitada se a soma dos *tokens* presentes em seus lugares de entrada for, no mínimo, igual ao somatório dos pesos dos seus arcos de entrada, $\sum_i w_i(p,t)$.

2) Uma transição habilitada pode ou não disparar, dependendo da rede de Petri que está sendo considerada. Podendo ser regida por uma guarda temporal (redes temporizadas) ou por um predicado lógico, que indicará quando o evento associado deve ocorrer.

3) Disparar uma transição habilitada t significa remover (consumir) $w_i(p,t)$ *tokens* dos lugares de entrada para cada arco i, e adicionar (gerar) $w_j(p,t)$ nos lugares de saída para cada arco j.

Transições sem lugares de entrada são designadas *transições fontes* (*source transitions*), e as sem lugares de saída são designadas *transições de dreno* (*sink transições*) [71]. Uma transição fonte está incondicionalmente habilitada. Uma transição de dreno sempre consome *tokens*, mas nunca os produz. Adicionalmente, um par (p,t) é designado como um auto-laço quando p é simultaneamente lugar de entrada e de saída de t. Uma rede de Petri é dita pura se não apresentar estruturas de auto-laços [58].

3.3 Classificação das Redes de Petri

A literatura apresenta diversas classificações para as redes de Petri. Segundo Reising e Rozemberg [86], *rede de Petri* é um termo genérico para uma classe de modelos estratificados em três camadas referentes ao nível de abstração pretendido. Na primeira camada, encontram-se as redes elementares nas quais os estados locais representam valores booleanos. Para a modelagem de sistemas reais de tamanho não trivial, as redes elementares explodem em dimensão de forma a se tornarem muito grandes para gerarem modelos manipuláveis. A segunda camada oferece a possibilidade a abstração do significado do estado local, gerando uma representação mais compacta. Essa camada tem como modelo básico as redes *Place/Transition*. Por fim, a terceira camada inclui as redes de Petri de alto nível, nas quais são usadas essencialmente lógica e álgebra para a construção de modelos. Nas redes de alto nível, os *tokens* podem modelar tipos de dados complexos e as transições encapsulam outras instâncias de rede ou mesmo

algoritmos codificados em linguagem de programação de alto nível. As redes de Petri coloridas [41] definem o modelo de rede mais representativo dessa camada.

Silva *et al* [91] sistematizam as camadas de abstração por meio de um espaço de formalismos de redes de Petri. Segundo essa sistematização, o espaço de formalismos relacionados às redes de Petri pode ser definido como um produto cartesiano entre o nível de abstração alcançável pela categoria de redes de Petri e a interpretação que pode ser atribuída à rede que modela um sistema.

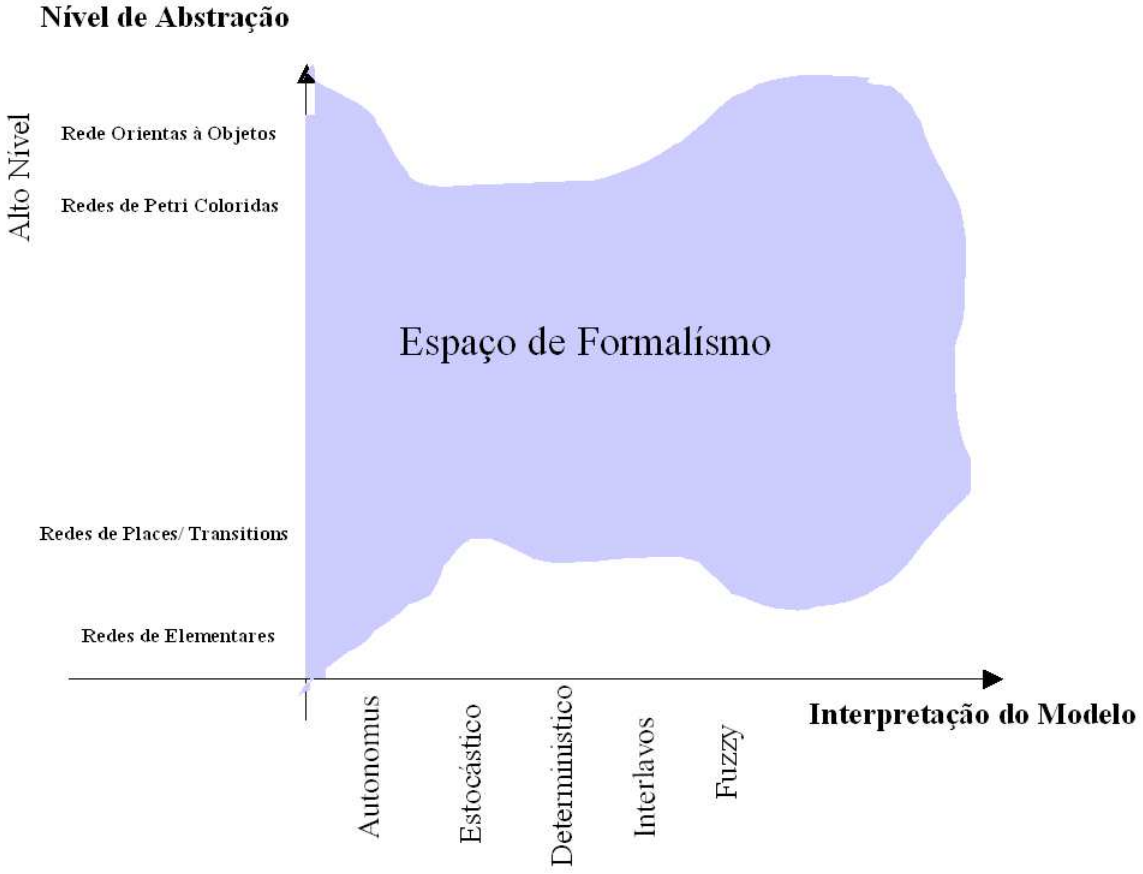

Figura 3.2: Espaço de Formalismo

A Figura 3.2 ilustra esse conceito. Como representante das redes de mais baixa capacidade de abstração estão as redes elementares. O próximo incremento da capacidade descritiva das redes de Petri nos leva às redes *Places/Transitions*. Extensões das redes *Places/Transitions*, como as redes temporizadas e estocásticas, estabelecem ganhos incrementais na capacidade de representação do modelo, marcados pelo acréscimo de parâmetros sem mudança da semântica dos elementos básicos da rede. Embora ampliem a capacidade de representação, tais extensões não incrementam o nível de abstração, apenas melhoram a expressividade do modelo em dado nível de abstração. Tal

processo permite que paradigmas de interpretação de comportamento de sistemas (modelos), tais como comportamentos *Fuzzies*, *Estocásticos* e *Determinísticos*, possam ser explorados em um mesmo eixo de abstração. Subindo o nível de abstração, encontram-se as redes de alto nível, representadas pelas redes coloridas e orientadas a objetos. Em tais redes, a mudança do nível de abstração é implementado pela flexibilização dos elementos de rede em representarem subsistemas complexos. Uma transição pode não somente representar uma ação atômica, mas encapsular todo um modelo. O *token* pode ser tratado como uma estrutura de dados (redes de Petri coloridas) ou mesmo como um modelo encapsulado (redes de Petri orientadas a objeto).

Com relação às possíveis interpretações de sistema proporcionadas pelas redes de Petri, Silva *et al* enumeram três categorias: *autonomus*, temporizadas e interpretação de dados. A categoria *autonomus* compreende uma percepção do sistema livre de relações com conceitos concretos, como tempo, abstração de dados, determinismo e não-determinismo do comportamento do sistema. Essa interpretação apenas modela o sistema como uma rede de Petri de forma autônoma das métricas do mundo real. O conceito de interpretação temporizada estabelece a capacidade de modelagem do sistema em termos dos tempos de operação do sistema, se determinísticos, estocásticos ou decorrente de um comportamento nebuloso (*Fuzzy*). A interpretação com base em dados explora a percepção do sistema em função de sua dinâmica de fluxo de dados, estabelecendo a necessidade de mecanismos específicos do modelo de rede. As rede de alto nível, devido a sua capacidade de expressar dados, provêm interessantes recursos para essa interpretação do modelo.

3.4 Redes Elementares

Construir um modelo compreende mapear elementos significativos da realidade em um conjunto de conceitos funcionalmente equivalentes. A base para tal mapeamento é a identificação dos elementos comportamentais do sistema e sua imagem no conjunto de conceitos funcionais disponíveis. Nas redes de Petri, os elementos do comportamento do sistema são representados por estruturas características ou redes elementares [58]. A Figura 3.3 mostra cinco estruturas: (a) seqüência, (b) concorrência, (c) sincronização, (d) escolha e (e) atribuição.

Seqüência

A seqüência é uma rede que representa ações consecutivas (vide Figura 3.3(a)). Após cada ação (disparo de transições), uma ação (ou conjunto de ações) estará habilitada

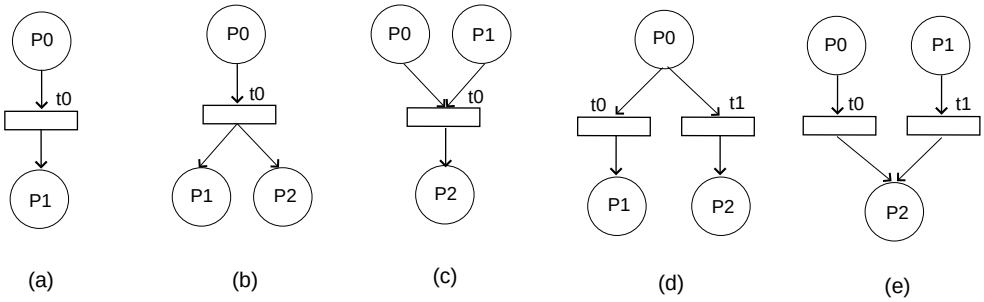

Figura 3.3: Estruturas Elementares

de forma que cada lugar é capaz de habilitar apenas uma transição.

Distribuição (ou *fork*)

Essa estrutura descreve a criação de dois ou mais caminhos na rede representando eventos simultâneos (vide Figura 3.3(b)). Com essa representação, torna-se possível criar modelos de processos paralelos. A estrutura representa o início de dois caminhos paralelos de eventos sujeitos à existência de pré-condições representadas por *tokens* nos lugares p_1 e p_2.

Sincronização (ou *join*)

Essa estrutura (vide Figura 3.3(c)) descreve o mecanismo de sincronização de processos paralelos. Uma sub-rede de sincronização garante que um outro processo só irá iniciar após o término de todos os processos paralelos que o antecedem. Isso é realizado pela co-dependência de um evento a duas ou mais condições internas.

Conflito ou Escolha

Se duas ou mais transições estão em conflito, significa que o disparo de uma (ou mais) muda a marcação de rede de forma tal a inibir a habilitação da(s) outra(s) (vide Figura 3.3(d)). Havendo um único *token* em p_0, t_0 e t_1 tornam-se conflitantes, e o disparo de uma transição elimina a possibilidade de disparo da outra. Esse tipo de estrutura é adequada para modelar estruturas *if-then-else* de uma descrição algorítmica.

Merging

A estrutura *Merging* define uma rede elementar que permite que dois ou mais processos habilitem um terceiro processo (habilitar o evento inicial de um terceiro processo)(vide

Figura 3.3(e)). Tanto o disparo de t_0 como de t_1 colocam um *token* no lugar p_2, ou seja, tanto t_0 quanto t_1 podem atribuir a condição inicial para disparo dos eventos sujeitos a p_2.

3.5 Subclasses

A depender das regras de formação, uma rede pode descrever determinados modelos de computação. Isso estabece uma sub-classificação das redes de Petri em função do modelo de computação representado.

Máquinas de Estado

Na subclasse Máquina de Estado [58] as transições possuem apenas um arco de entrada e um arco de saída. Uma rede Máquina de Estado pode representar conflito e *merging*, mas é incapaz de descrever paralelismo e sincronização.

Grafo Marcado

Uma rede Grafo Marcado ou Grafo de Eventos [58] possui apenas um arco de saída e um de entrada para cada lugar[1]. Redes dessa classe podem representar concorrência e sincronização, mas não conflito e *merging*.

Rede de Petri Escolha Livre

A subclasse denominada Escolha Livre [58] possibilita a modelagem de conflitos, bem como, concorrência e sincronização. Existem duas categorias de redes para essa subclasse, uma proposta por Hack [34] e outra por Commoner [26]. Na definição de Hack, uma rede pertence à subclasse Escolha Livre, se para um dado conjunto de transições conflitantes $T_1 \subseteq T$ houver apenas um lugar de entrada para estas transições, ou seja, $I(t_j) = \{p_i\}, \forall t_j \in T_1$, ou se para dado conjunto de lugares existir apenas uma transição de saída, ou seja, $O(p_i) = \{t_j\}, \forall p_i \in P$. Onde $I(t_j)$ representa o conjunto de lugares de entrada da transição t_j e $O(p_i)$ representa o conjunto de saída do lugar p_i. Segundo Commoner, dado um conjunto de pares de transições $\{(p_i, p_k)\}$ de uma rede, ela pertence a subclasse Escolha Livre, se houver transições de saídas comuns, de forma que $O(p_i) = O(p_k)$. Em ambas as definições, a escolha do evento (disparo da transição) é livre.

[1] É, portanto, uma classe dual da classe Máquina de Estado

3.6 Avaliação Qualitativa

As redes de Petri não são apenas usadas para a descrição de modelos. Elas permitem análise de propriedades dos sistemas modelados. Existem três técnicas normalmente usadas para avaliar propriedades de um modelo: análise, verificação e validação. Valmari [106] define análise como o processo de se encontrar respostas para perguntas formais a respeito do comportamento do sistema. Um algoritmo de análise deve permitir questões do tipo "*qual o número máximo de mensagens armazenadas na pilha?*" [106]. Verificação significa executar um algoritmo bem definido para checar a existência ou não de dada propriedade formal. Por propriedade formal, entendam-se preceitos gerais formalmente definidos, tais como capacidade de alcançar um estado a partir de um dado estado inicial (*alcançabilidade*), limitação na geração de estados (*limitação*) e comportamento livre de travamento (*deadlock-free*). Validações são métodos para checar se o comportamento do sistema é o desejado. O termo "comportamento desejado" está associado à expectativa do observador, sendo, portanto, a validação um processo inerentemente informal [106].

As redes de Petri podem ser usadas como mecanismo de validação e verificação a partir da análise de dois tipos de propriedades da rede: propriedades comportamentais e estruturais. As propriedades comportamentais descrevem os tipos de comportamentos que a rede representa, e que, conseqüentemente, estarão presentes no sistema modelado. A definição de propriedades comportamentais requer a execução da rede e é dependente do valor inicial do modelo (marcação inicial da rede). As propriedades estruturais, por outro lado, elucidam comportamentos a partir da estrutura da rede, sendo independente da marcação inicial.

3.6.1 Propriedades Comportamentais

As propriedades comportamentais de uma rede de Petri são de grande importância, pois demonstram padrões de comportamento do sistema não evidentes em outros modelos de descrição, como nos modelos puramente baseados em linguagem de programação. Nesta seção, será feita uma rápida descrição das propriedades comportamentais capturáveis em uma rede de Petri e de sua relação com o comportamento do sistema.

Alcançabilidade

Denomina-se *alcançabilidade* a capacidade da rede de, partindo de uma marcação M_0 (dita marcação original ou inicial), alcançar a marcação M_i. A marcação M_i é dita alcançável a partir da marcação M_0, se existe uma seqüência de disparos que transforma

M_0 em M_i. Uma seqüência de disparos, algumas vezes denominada ocorrência, é definida pelo conjunto $\sigma = \{t_1, t_2..., t_i\}$, que descreve a seqüência de transições a serem disparadas. Uma marcação M_i alcançável a partir de uma marcação M_0 através de σ é formalmente descrita como $M_0|\sigma > M_i$. O conjunto de todas as marcações possíveis em uma rede, a partir de uma marcação inicial M_0, é representado por $R(M_0)$. O conjunto de todas as seqüências de disparo que levam a rede à marcação M_i, a partir da marcação M_0, é representado por $L(M_0)$. Lipton [54] demonstra que a análise de alcançabilidade é um problema cuja complexidade é exponencial. Isso estabelece um limitante à análise de sistemas complexos em que o número de estados alcançáveis "explode" tornando a análise inviável. Técnicas de redução do espaço de estados, em função do domínio de aplicação e complexidade, devem fazer parte da metodologia de análise.

Limitação e Segurança

Uma rede de Petri é dita ser *k-limitada* (ou simplesmente limitada) se o número de *tokens* em cada lugar não exceder o número finito k, para qualquer marcação alcançável a partir de M_0. Uma rede é dita ser segura se $k = 1$. Em diversos modelos, os lugares representam *buffers* de armazenamento de dados, de forma que o conceito de limitação mapeia na rede a restrição de tamanho associada aos *buffers* do sistema. Redes seguras são classicamente usadas para representar sistemas digitais. Por vezes, é usado o termo rede binária como sinônimo de rede segura. Mecanismos de transformação podem ser usados para converter uma rede insegura ($k > 1$) em uma rede segura [58].

Liveness (ou rede viva)

Uma rede de Petri é dita "viva" se, não importando quais marcações sejam alcançáveis a partir de M_0, for possível disparar qualquer transição através do disparo de alguma seqüência de transições $L(M_0)$. Observa-se, então, que o conceito de *deadlock* está fortemente conectado ao conceito de *liveness* [58]. O fato de um sistema ser livre de *deadlock* não implica que este seja vivo, embora um sistema vivo seja livre de *deadlocks*. Exemplos de redes sem *deadlock*, mas não vivas, são aquelas onde não existe nenhum estado que trave o sistema, mas existe ao menos uma transição que nunca disparará. A análise de *liveness* de uma rede permite verificar se os eventos modelados efetivamente ocorrem durante o funcionamento do sistemas, ou se foram definidos eventos mortos no modelo.

A análise de *liveness* é ideal para avaliar diversos sistemas reais, pois informa que o sistema é passível de mudanças de estados. É, contudo, muito caro observar esta

propriedade em modelos de sistemas muito complexos. Este fato faz com que sejam usadas abordagens "relaxadas" do conceito de *liveness*. Para tal, são analisados diferentes níveis de *liveness* presentes em uma transição [71]. Uma transição t é dita ser viva em cinco níveis:

- L_0**- (morta)**. Se t nunca poder ser disparada em qualquer seqüência $L(M_0)$, ela é dita uma transição morta.
- L_1**-Potencialmente disparável**. Se t pode ser disparada ao menos uma vez em alguma seqüência $L(M_0)$, ela é dita potencialmente disparável.
- L_2. Se dado qualquer inteiro positivo k, t puder ser disparada ao menos k, em um alguma seqüência $L(M_0)$, ela é dita viva em nível L_2.
- L_3. Se t aparece um número infinito de vezes em alguma seqüência $L(M_0)$, ela é dita viva em nível L_3.
- L_4**- Viva (ou simplesmente viva)**. Se t é potencialmente disparável para todas as marcações da rede, ela é dita viva em nível L_4, ou simplesmente viva.

Reversibilidade e *Home State*

Uma rede é dita reversível se, para cada marcação (ou estado) M em $R(M_0)$, M_0 é alcançável a partir de M. Sendo assim, uma rede reversível pode voltar sempre para sua marcação original (ou estado original). A identificação dessa propriedade no modelo é de suma importância, principalmente em sistemas de controle, pois informa a capacidade do modelo de retornar a um ponto de operação inicial. Em algumas aplicações não é necessário voltar-se ao estado inicial, mas sim a um estado base, de onde o sistema evolui ciclicamente. Esse estado base é denominado *Home State* [58].

Cobertura

Quando se deseja saber se alguma marcação M_i pode ser obtida a partir de uma marcação específica M_j, temos o problema denominado *cobertura de uma marcação*, ou seja, a cobertura da marcação M_j. Uma marcação M_j é dita coberta se existe uma marcação M_k tal que $M_k \geq M_j$. Em alguns sistemas, deseja-se apenas observar o comportamento de determinados lugares. Para isso, restringe-se a pesquisa a apenas um conjunto de lugares de particular interesse. Esse mecanismo é denominado cobertura de submarcações.

Persistência

Uma rede é dita persistente se, para qualquer par de transições habilitadas, o disparo de uma transição não desabilita o disparo de outra transição. A noção de persistência é muito importante quando tratamos de sistemas paralelos em circuitos digitais que tenham atividades assíncronas. Vale ressaltar que toda rede da subclasse grafo-marcado é uma rede persistente, pois esta subclasse não possui conflitos. A presença da propriedade persistência em uma rede de Petri denota que, no sistema modelado, nenhum evento inibe a ocorrência de um outro.

Justiça

Embora existam diferentes pontos de vista sobre o significado do termo justiça em sistemas concorrentes, aqui apresentam-se os dois principais conceitos.

- *Justiça Limitada.* Segundo este ponto de vista, duas transições t_i e t_j são classificadas como *B-fair* (*Bounded-fairness*), se o número de vezes que uma delas dispara, enquanto a outra não dispara, é limitado.

- *Justiça Incondicional.* Nesse ponto de vista, uma seqüência de disparo de transições s_1 é classificada como incondicionalmente justa se essa seqüência é finita ou se todas as transições da rede aparecem um número infinito de vezes nessa seqüência.

A propriedade de justiça mostra o equilíbrio que o sistema modelado tem, em relação ao número de ocorrência de seus eventos. Ou seja, elucida se os eventos possuem iguais possibilidades de ocorrência. Isso é particularmente útil para inferir sobre o grau de utilização dos recursos no tempo, ou elucidar mecanismos que fazem com que o comportamento do sistema seja tendencioso.

3.6.2 Propriedades Estruturais

As propriedades estruturais de uma rede permitem uma análise independente da marcação inicial, possuindo dependência apenas da topologia da rede. Para isso, na análise é necessário considerar as redes como puras, ou seja, não possuidoras de *auto-laços* [58]. *Auto-laços* são estruturas caracterizadas pela presença de lugares que são simultaneamente lugares de entrada e saída de uma mesma transição, como mostra a Figura 3.4. A Figura 3.4 mostra ainda que a presença de um auto-laço pode ser neutralizada pela inserção de um conjunto lugar-transição em série como o *laço*. Esse conjunto é denominado *par Dummy* [58]. A partir da análise da rede, propriedades estruturais podem ser

capturadas, dentre elas: *Limitação Estrutural*, *Conservação Estrutural*, *Repetitividade* e *Consistência*.

- *Limitação Estrutural*. Uma rede é limitada se seu número de *tokens* e seus lugares forem limitados. Isso pode ser estruturalmente definido pelo equilíbrio dos pesos dos arcos de entrada e saída dos lugares.

- *Conservação Estrutural*. A conservação permite a verificação da não destruição de recursos. Em modelos nos quais recursos são modelados em *tokens*, a simples conservação do total de *tokens* informa a conservação de recursos no sistema.

- *Repetitividade*. Uma rede é repetitiva, se para uma dada marcação e uma dada seqüência de transições disparáveis, todas as transições de rede são disparadas ilimitadamente. A existência de repetitividade em um rede que modela um dado sistema demonstra a existência de comportamentos cíclicos nesse sistema.

- *Consistência*. Uma rede de Petri tem a propriedade de consistência se dada uma seqüência de disparos de transições, a partir de uma marcação inicial M_0, retorna-se a M_0 de forma que todas as transições sejam disparadas ao menos uma vez. A existência de consistência em um rede que modela um dado sistema demonstra a existência de comportamentos cíclicos no qual todos os eventos modelados ocorrem efetivamente.

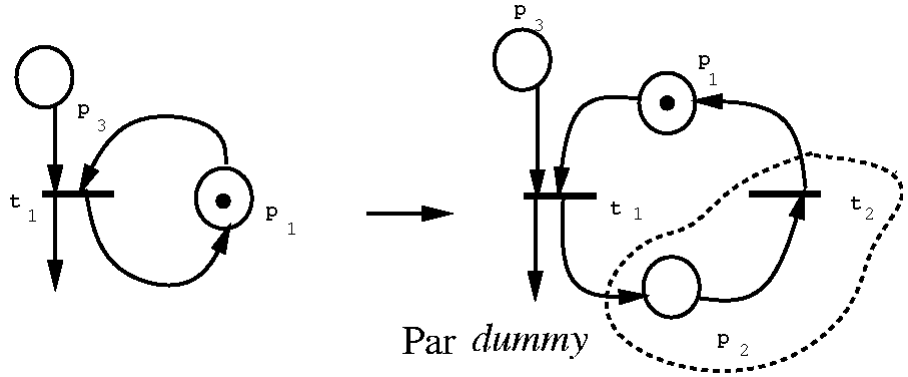

Figura 3.4: Transformação de uma rede impura em pura

3.7 Métodos de Análise

Os métodos de análise podem ser divididos em três tipos. O primeiro tipo de método é baseado na construção do espaço de estados. Tal espaço é construído com base no

gráfico de alcançabilidade da rede, também chamado árvore de cobertura. O gráfico de alcançabilidade é dependente da marcação inicial, sendo usado para a análise de propriedades comportamentais. O principal problema desse método consiste na alta complexidade computacional, mesmo quando se usam técnicas de redução de gráficos [105]. O segundo método é baseado nas equações de estados. A principal vantagem desse método sobre o gráfico de alcançabilidade é o uso de equações algébricas simples para a determinação das propriedades da rede. O uso dessas equações permite a análise de invariantes, os quais se subdividem em invariantes de transição e invariantes de lugar. Por invariantes de transição, entenda-se o conjunto de transições que sempre disparam em qualquer mudança de estado do sistema. Por outro lado, o conceito de invariante de lugar está associado à identificação do conjunto de lugares cujo número total de *tokens* é invariante para qualquer estado do sistema modelado. A análise de invariantes de transição permite a inferência dos *componentes repetitivos estacionários* do sistema modelado. Já os invariantes de lugar permitem a inferência dos *componentes conservativos* do sistema modelado. Uma terceira técnica, compreendendo a redução da rede para análise, é freqüentemente utilizada em modelos de grandes dimensões. A redução consiste na fusão de lugares e transições, de forma a gerar uma rede menor, mas com propriedades equivalentes à rede original.

3.7.1 Árvore de Cobertura

O método de análise denominado *Árvore de Cobertura* baseia-se na construção de uma árvore que possibilite a representação das marcações de uma rede [58]. O conjunto de marcações de uma rede é representado por uma estrutura de árvore, cujos nós são as marcações e cujos arcos são as transições disparadas. Para uma rede limitada, a árvore de cobertura é denominada *árvore de alcançabilidade*, pois descreve todas possíveis marcações alcançáveis pela rede [71]. Para possibilitar a representação finita de incontáveis conjuntos de marcações, é utilizado o símbolo ω. O ω abstrai o valor da marcação de um lugar de forma a representar um grupo de marcações com um única notação.

Seja uma rede com três lugares p_0, p_1 e p_2, cuja marcação é descrita na forma (p_0,p_1,p_2), a notação (1,0,ω) representa o conjunto de marcações da rede (estados do sistema modelado), no qual o número de *tokens* em p_2 é expresso por ω. O símbolo ω informa que o lugar acumula marcas, sem precisar quais as marcações alcançáveis. Para manter a árvore finita, ω pode representar valores infinitos (em redes não limitadas). Devido a isso, Murata [71] interpreta ω como representando o "infinito". A árvore de cobertura é gerada por um algoritmo exaustivo que explora as possíveis marcações

decorrentes de uma marcação inicial M_0. No exemplo da Figura 3.5, a avaliação da árvore de cobertura apresenta $\{(1,0,0),(0,1,1),(1,0,\omega),(0,1,\omega)\}$ como conjunto de marcações da rede, no qual são caracterizadas repetições de estados a partir de $(0,1,1)$ voltando para o estado inicial, e uma "armadilha" de repetições locais a partir de $(1,0,\omega)$ retornando para $(0,1,\omega)$.

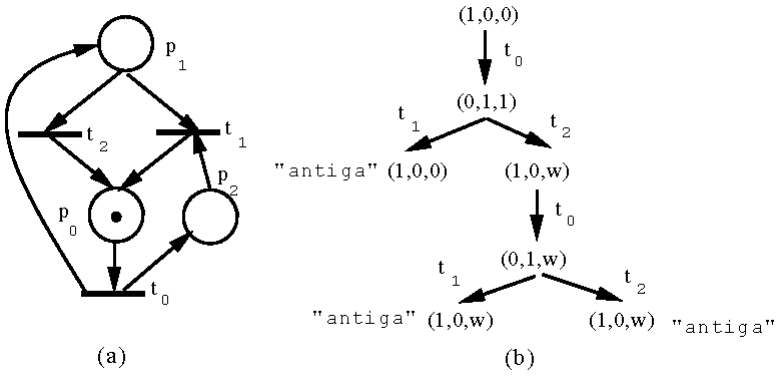

Figura 3.5: (a) Exemplo de rede (b) Árvore de cobertura

Algumas propriedades como limitação, segurança, presença de transições mortas e alcançabilidade de marcações podem ser analisadas [112]. A identificação de tais propriedades a partir da árvore de cobertura é realizada como segue:

1. Uma rede (N, M_0) é limitada, se e somente se ω não aparecer em sua árvore de cobertura.

2. Uma rede (N, M_0) é segura, se e somente se apenas e 0's e 1's aparecem na árvore de cobertura.

3. Uma transição é morta, se e somente se ela não aparece como arco na árvore de cobetura.

4. Se uma marcação M é alcançável a partir de M_0, então existe o nó M', tal que $M \leq M'$.

3.7.2 Análise da Equação de Estado

A equação matricial que descreve o comportamento dinâmico das rede das Petri é normalmente denominada *equação fundamental das redes de Petri* ou *equação de estado*. A equação fundamental das redes de Petri provê meios para a verificação da

alcançabilidade das marcações [58]. Para que possa ser aplicada a equação fundamental, é necessário que a rede seja pura, ou ao menos transformada em uma rede pura pela introdução de pares *Dummy*. A equação fundamental descreve a inserção e remoção de *tokens* nos lugares por meio do conceito de matriz de incidência. A matriz de incidência descreve a estrutura da rede em termos de uma matriz *lugares* × *transições*, onde cada elemento da matrix, a_{ij}, representa o número de *tokens* que a transição j insere no lugar i. Caso a transição consuma *tokens* do lugar em questão, o elemento a_{ij} assume valores negativos. A equação fundamental das redes de Petri é apresentada na Equação 3.1, na qual $M'(p)$ e $M_0(p)$ são vetores que representam as marcações finais e iniciais respectivamente, C é a matriz de incidência, s é o vetor característico cujos componentes s_i são números naturais, representando o número de vezes que cada transição t_i é disparada para se obter a marcação $M'(p)$ a partir de $M_0(p)$ [71, 58].

$$M'(p) = M_0(p) + C \cdot \bar{s}^T, \forall p \in P \tag{3.1}$$

Como exemplo, seja a rede da Figura 3.6. Deseja-se verificar a alcançabilidade da marcação $M' = (0, 0, 0, 1, 1, 0, 1)$ a partir da marcação inicial apresentada.

$$\begin{vmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{vmatrix} - \begin{vmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 2 \end{vmatrix} = \begin{vmatrix} -1 & 0 & 0 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ -1 & 1 & -1 & 1 & 0 & 0 \end{vmatrix} \times \begin{vmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{vmatrix}$$

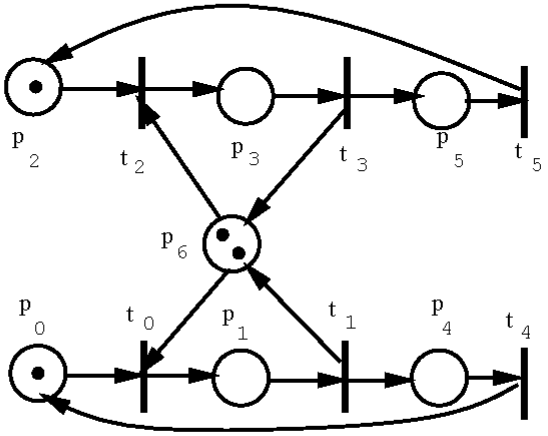

Figura 3.6: Exemplo de análise de alcançabilidade usando a equação fundamental

A solução da equação matricial é

$$s = (s_0, s_0, 1 + s_3, s_3, s_0 - 1, s_3)$$

.

A solução encontrada representa uma dentre várias possíveis seqüências de disparos das transição que levam a rede para a marcação $M' = (0, 0, 0, 1, 1, 0, 1)$. Assumindo $s_0 = 1$ e $s_3 = 0$, tem-se $s = (1, 1, 1, 0, 0, 0)$ como solução, demonstrando que $M' = (0, 0, 0, 1, 1, 0, 1)$ é alcançável após disparar t_0, t_1 e t_2 uma vez. Infelizmente, da resolução da equação fundamental, pode-se inferir resultados incorretos, uma vez que a equação não descreve as restrições de disparo da transições, ou seja, a equação assume que as transições sempre estão habilitadas a disparar.

3.7.3 Análise de Invariantes

A análise de invariantes compreende a busca por elementos invariantes na rede que expressem elementos do modelo que possam ser classificados com componentes repetitivos estacionários ou como componentes conservativos. A busca por elementos repetitivos compreende capturar os padrões de disparo das transições, enquanto a busca por elementos conservativos compreende a avaliação do uso de recursos normalmente representados pelos *tokens*. O padrão de disparo das transições e a conservação dos *tokens* são acessados por meio dos conceitos de invariante de transição e de lugar.

Invariantes de Transição

Se em uma rede é possível disparar cada transição n_i vezes de forma que no final do processo a rede se encontre na marcação de origem do processo, esta rede possui componentes repetitivos estacionários [58]. Em outras palavras, a rede modela um sistema que possui comportamentos cíclicos. Os componentes repetitivos estacionários são obtidos pela solução do sistema $C \cdot \bar{s} = 0$. Diz-se que o vetor característico encontrado $\bar{s} \geq 0$ é o invariante de transição I_t. Em outras palavras, um invariante de transição é um vetor de dimensão igual ao número de transições na rede, onde cada componente deste vetor corresponde ao número de vezes que cada transição deve ser disparada para que o sistema retorne ao seu estado inicial. Cada invariante fornece um conjunto de transições que, quando disparadas, não alteram a marcação da rede.

Por meio do cálculo dos invariantes de transição, é possível verificar a existência dos componentes repetitivos estacionários, bem como verificar a consistência parcial ou total de uma rede. Uma rede é dita consistente se, disparando uma seqüência de transições a partir de uma marcação inicial M_0, retorna-se a M_0, de forma que todas as

transições da rede foram disparadas ao menos uma vez. Tal análise permite inferir sobre o comportamento cíclico do sistema modelado, caracterizando as transições (eventos) críticas para a manutenção da natureza cíclica.

Invariantes de Lugar

A análise de invariantes de lugar permite a avaliação dos componentes conservativos do sistema, sem a necessidade da observação exaustiva da árvore de cobertura. Esses componentes estão associados ao conjunto de lugares em que a soma ponderada das marcas desses lugares permanece constante quando uma seqüência de transições é disparada [58]. A ponderação diz respeito aos pesos atribuídos aos lugares. A avaliação dos invariantes de lugar, em última análise, permite verificar se o sistema é conservativo em termos de recursos (*tokens*) associados ao conjunto de estados (lugares).

3.8 Extensões de Redes de Petri

Em modelos de sistemas reais, os eventos estão co-relacionados no tempo por vezes de forma não-determinística. Existe a necessidade de abstrair detalhes do sistema por meio de estruturas hierárquicas e por fim necessita-se que a rede gere uma especificação executável do sistema. Adicionalmente, o modelo precisa ser validado e verificado frente a complexas estruturas de dados de entrada. A necessidade específica de modelos baseados em redes de Petri para a avaliação de desempenho, particularmente em sistemas concorrentes, fez surgir extensões com temporizações determinísticas (de valor determinado) e com temporizações não-determinísticas, no qual há a associação do tempo com variáveis aleatórias regidas por distribuições exponenciais, as chamadas *redes de Petri estocásticas* e *redes de Petri estocásticas generalizadas* [65]. A necessidade de fazer as redes de Petri operar como linguagens de descrição, agregando capacidade de simulação e altos níveis de abstração do modelo, fez sugir as redes de alto nível. Nas próximas seções, será feita uma sumária descrição de algumas das extensões mais difundidas.

3.8.1 Extensões com Temporização

A necessidade de se representar modelos cujo tempo é um elemento fundamental proporcionou o desenvolvimento de extensões temporizadas das redes de Petri. Áreas de estudo como modelagem de *hardware*, protocolos de comunicação e sistemas de tempo real, necessitam de modelos que tenham o tempo como parâmetro. Adicionalmente, o modelo deve suportar técnicas de análise da rede, visando à extração de informações

referentes ao desempenho do sistema modelado. O tempo pode ser associado a qualquer entidade da rede (lugares, transições, arcos e *tokens*). No entanto, a maioria das extensões associa o tempo à transição, por esta ser normalmente associada aos eventos. Serão consideradas aqui apenas as características básicas das redes de Petri temporizadas cujo tempo é associado às transições.

As redes temporizadas podem usar duas políticas de disparo com o objetivo de modelar a passagem do tempo [58].

- **Disparo em três fases**. Os *tokens* dos lugares de entrada são consumidos quando a transição é habilitada (fase 1), transcorre o atraso associado à transição (fase 2) e finalmente *tokens* são gerados nos lugares de saída (fase 3).

- **Disparo atômico**. Os *tokens* permanecem nos lugares de entrada após a habilitação da transição por um tempo igual à duração associada à transição. No final do intervalo de tempo associado a transição, os *tokens* são consumidos dos lugares de entrada e gerados nos lugares de saída (disparo da transição).

Na política de disparo atômico, o disparo de uma transição temporizada pode desabilitar outras cujas temporizações foram iniciadas com a habilitação, mas não chegaram ao fim, pois as transições se tornaram desabilitadas no momento em que a outra transição consumiu os *tokens* dos seus lugares de entrada. O que deve ocorrer com esta transição cujo disparo foi habilitado, mas abortado antes do final da temporização? Existem dois mecanismos básicos para tratar tal situação: (i) o temporizador associado à transição mantém o valor presente e continua a contagem quando reabilitado e (ii) o temporizador associado à transição é re-iniciado, ou seja, seu valor presente é descartado e um novo valor é carregado. A partir desses dois mecanismos básicos, é possível construir várias políticas de "memória", sendo três, contudo, as mais usuais [62].

- **Re-amostragem (*Resample*)**. O contador de tempo de todas as transições é re-inicializado quando ocorre o disparo de alguma transição da rede. A temporização de todas as transições habilitadas na nova marcação tem suas temporizações re-inicializadas.

- **Habilitação de Memória (*Enabling Memory*)**. Transições que permanecem habilitadas na nova marcação mantêm suas contagens de tempo, enquanto as que perdem a habilitação têm seus contadores zerados.

- ***Age Memory***. Os contadores são mantidos inalterados mesmo se a transição não estiver habilitada na nova marcação. Nesse caso, haverá paralisação da contagem. Se ela voltar a ser habitada, a contagem prosseguirá do valor memorizado.

3.8.2 *Timed Petri Nets*

Uma *Timed Petri Net* é definida pelo par (PN,D), no qual PN descreve uma rede e D é uma função que associa à cada transição da rede um número real não-negativo. D define o tempo da transição, normalmente citado como duração da transição. Transições em uma *Timed Petri Net* são habilitadas da mesma forma que uma rede não temporizada. O disparo de uma transição habilitada pode ocorrer seguindo a política de disparo em três fases ou a de disparo atômico, conforme já descrito.

O número de vezes que uma transição pode disparar antes de torna-se desabilitada (ter consumido todos os *tokens* dos lugares de entrada) é denominado *grau de habilitação da transição* (*enabling degree*) [64]. Quando o grau de habilitação de uma transição é maior que um (> 1), deve-se ter especial atenção à semântica de temporização. Para tratar tal questão, existem três semânticas usuais [63].

- *Semântica de servidor único (Single-server sematics).* O retardo de disparo é estabelecido quando a transição é habilitada pela primeira vez, e novos retardos são gerados a cada novo disparo da transição. O conjunto de estados habilitados é tratado serialmente, e a especificação temporal associada é independente do grau de habilitação. Marsan [63] discute sobre funções especiais de dependência de marcação, estabelecendo um mecanismo de exceção à independência do grau de habilitação.

- *Semântica de servidor infinito (Infinite-server sematics).* Todos os conjuntos de *tokens* são processados tão logo estabeleçam a marcação local dos lugares de entrada das transições temporizadas. A temporização associada a todas essas transições é procesada paralelemente em contagem regressiva dos tempos. Nesse caso, é dito que a especificação total do conjunto de transições depende diretamente dos graus de habilitação.

- *Semântica de servidor múltiplo (Multiple-server sematics).* As marcações são processadas como disparando em paralelo (similarmente à semântica anterior) limitadas; contudo, a um número máximo de disparos paralelos, ditos K disparos. Para altos valores do grau de habilitação, as temporizações associadas às novas habilitações das transições são programadas só se o número de transições em

concorrência for menor que K. A temporização total das transições depende diretamente dos graus de habilitação das transições, até o valor limite K.

3.8.3 Redes de Petri Estocásticas

Uma rede de Petri estocástica [64] é uma *Timed Petri Net*, com disparo atômico, na qual todas as transições possuem um retardo aleatório definido por uma distribuição exponencial. Isso permite que processos estocásticos possam ser modelados. Entendam-se processos estocásticos como processos em que os eventos têm natureza probabilística. Supondo um retardo d_i de um evento associado à transição t_i, retardo este aleatório e definido por uma função de distribuição exponencial expressa na Equação 3.2, o retardo médio é estabelecido pela Equação 3.3, na qual λ_i é a taxa de disparo da transição t_i.

$$F_X(\chi) = Pr[X \leq \chi] = 1 - e^{\lambda_i \chi} \tag{3.2}$$

$$\overline{d}_i = \int_0^\infty [1 - F_X(\chi)]dx = \int_0^\infty e^{\lambda_i \chi} dx = \frac{1}{\lambda_i} \tag{3.3}$$

Observa-se que em uma distribuição exponecial a expectância (valor médio) é definido pelo inverso do coeficiente λ_i. Como a grandeza que se está modelando é o tempo, o inverso de sua expectância é uma taxa (ou freqüência). Observa-se que, pelo uso da função exponencial, descrevem-se os disparos de transições (eventos) unicamente por sua taxa média (λ_i), o que facilita a especificação comportamental do sistema. Adicionalmente, a concorrência de duas transições cujos atrasos seguem funções de distribuição exponencial $\lambda e^{-\lambda \chi}$ e $\mu e^{-mu\chi}$ define um evento exponencial cuja função é $(\mu + \lambda)e^{-(\mu+\lambda)\chi}$. Como a função exponencial não possui memória, mostra-se que a evolução dos estados da rede e a cadeia de Markov são isomórficos. Ou seja, a criação de um gráfico de estados da rede define uma cadeia de Markov.

Marsan *et al* [65] introduziram as GSPN (*Generalized Stochastic Petri Net*). As GSPN empregam dois tipos de transições: (i) transições temporizadas, em que os tempos de disparos são definidos por uma função de distribuição de probabilidade exponencial, na qual o parâmetro associado à transição representa o parâmetro de distribuição (taxa de disparo média); e (ii) transições de disparo imediato. Marsan *et al* demonstram [65] que as GSPN podem ser representadas como cadeias de Markov de tempo contínuo.

Redes determinísticas e estocásticas ou DSPN (*Deterministic and Stochastic Petri Nets*) foram introduzidas como uma extensão das GSPNs [61]. As DSPN permitem

a associação de transições temporizadas com tempos estocásticos (distribuição exponencial) e com tempos determinísticos (fixos). Dessa forma, modelos que precisam representar conjuntamente eventos estocásticos e eventos determísticos, referentes a tempos de *time-out* e tempos de propagação, podem ser facilmente descritos.

3.8.4 Redes de Petri Coloridas

Para as redes de Petri apresentadas nas seções anteriores, existe apenas um tipo de marca, o que não permite a diferenciação de recursos em um lugar, sendo necessário lugares distintos para a representação de recursos distintos. Adicionalmente, nas redes apresentadas, não há suporte para hierarquia, o que torna as redes inadequadas à especificação de sistemas de grande dimensão e complexidade. Kurt Jensen *et al* [41] propuseram uma extensão das redes de Petri que combina as proriedades analíticas e o poder de descrição de paralelismo com a expressividade de uma linguagem de programação. Inicialmente, *tokens* foram definidos como possuidores de cores, representando valores distintos (recursos distintos), o que deu origem à denominação de redes de Petri coloridas (*Coloured Petri Nets ou CPN*). No formato atual, as CPNs definem os *tokens* com tipos de dados sofisticados, de forma que um *token* possui um valor. O tipo de dado em uma CPN determina um conjunto de cores, sendo por isso usado o termo *conjunto de cores* (*colour set*) no lugar do termo *tipo* (*Type*). As transições processam os *tokens* gerando novos *tokens* (novos valores). Especificações mais elaboradas podem ser descritas por códigos escritos em linguagem de alto nível (CP-ML)[2]. Estruturas hierárquicas podem ser criadas embutindo em transições outras redes cujas transições podem, por sua vez, embutir outras redes, e assim por diante. Devido a esse poder de expressão, as redes de Petri coloridas tornaram-se uma das mais proeminentes variantes das redes de Petri. As redes de Petri coloridas definem suas entidades com as seguintes distinções.

- *Tokens* expressam valores e possuem estruturas de dados similares às linguagens de programação. O tipo do *token* estabelece o seu "conjunto de cores". Conjunto de cor no contexto das redes de Petri coloridas é sinônimo de tipo de dado.
- Lugares são associados a tipos de dados (conjunto de cores), determinando o tipo de *token* que o lugar pode receber.

[2]CP-ML é um subconjunto da linguagem funcional Standard ML. Esse subconjunto foi especificamente criado para as redes de Petri coloridas.

- Transições podem expressar comportamentos complexos, "processando" os valores transportados pelos *tokens*.
- Estruturas hierárquicas podem ser manipuladas de forma a obter-se representações mais compactas, uma transição pode encapsular toda uma rede que processa o *token* consumido por ela.
- Os eventos associados às transições podem ser representados por um algoritmo descrito em CPN-ML.

Segundo a semântica proposta por Jensen [39], as redes de Petri coloridas são formalmente construídas em função de entidades básicas definidas a seguir.

Definição 3.2 (*Token*) .
O tokens é uma representação de valor (literal), podendo ter um tipo (conjunto de cor) primitivo ou composto.

Definição 3.3 (*Multiconjunto*) *:*
Multiconjunto é a representação de uma coleção de elementos que possuem o mesmo "conjunto de cor" (tipo de dado), identificando a repetição de valores. Seja N um conjunto de números inteiros não negativos, um Multiconjunto MS, definido a partir de um conjunto não-vazio S, é construído a partir de uma função de contagem $m : S \rightarrow N$, onde $m(s)$ mapeia o número de incidência do elemento s no conjunto S em inteiro positivo no conjunto N. Onde

$$MS = \sum_{s \in S} m(s)'s.$$

Sendo denotado S_{MS} como o conjunto de todos os Multiconjunto contidos em S, de forma que o conjunto $\{m(s)|s \in S\}$ é denominado conjunto de coeficientes do Multiconjunto.

Seja uma coleção de *tokens* representando os animais de uma fazenda, o *token* é definido como do tipo ANIMAL, ou seja, seu conjunto de cores é ANIMAL. Admita-se que a coleção de *tokens* que representa um estado da fazenda com relação à quantidade de animais é representado por: 4 *tokens* representando o valor "vaca", 3 *tokens* representando o valor "cabra" e 5 *tokens* com o valor "frango". Essa coleção de *tokens* é representada pelo *Multiconjunto*:

$$MS = \sum_{s \in S} m(s)'s = 4'vaca + 3'cabra + 5'frango.$$

Definição 3.4 (Operações de *Multiconjunto*) .
Seja o conjunto de Multiconjuntos $\{m, m_1, m_2\} \subseteq S_{MS}$ *e* n *um inteiro não negativo. São definidas as seguintes operações básicas entre Multiconjuntos:*
(i) $m_1 + m_2 = \sum_{s \in S}(m_1(s) + m_2(s))'s$ *(adição)*
(ii) $n \cdot m = \sum_{s \in S}(n \cdot m(s))'s$ *(multiplicação por um escalar)*
(iii) $m_1 \neq m_2 \Rightarrow \exists s \in S | m_1(s) \neq m_2(s)$ *(comparação* $\neq$*)*
(iv) $m_1 \leq m_2 \Rightarrow \exists s \in S | m_1(s) \leq m_2(s)$ *(comparação* $\leq$*)*
(v) $m_1 \geq m_2 \Rightarrow \exists s \in S | m_1(s) \geq m_2(s)$ *(comparação* $\geq$*)*
(vi) $|m| = \sum_{s \in S} m(s)$ *(dimensão)*
(vii) $m_2 - m_1 = \sum_{s \in S}(m_2(s) - m_1(s))'s$*, se e somente se,* $m_2 \geq m_1$ *(subtração)*

Adicionalmente, cabe estabelecer os operadores primitivos para o modelo CPN.

Definição 3.5 (Operador Tipo) .
Sejam V*,* EXP *e* T*, respectivamente, o conjunto de variáveis, expressões e tipos de um modelo. A função* $Type : V \cup EXP \rightarrow T$ *é denominada operador de tipo, onde* $Type(v\_exp)$ *mapeia a variável ou expressão* v_exp *em um tipo válido.*

Definição 3.6 (Operador Conjunto de Variáveis) .
Sejam V *e* EXP*, respectivamente, o conjunto de variáveis e expressões de um modelo. A função* $Var : EXP \rightarrow V$ *é denominada operador de conjunto de variáveis, onde* $Var(exp)$ *extrai o conjunto de variáveis presentes na expressão* exp*.*

De posse dos operadores primitivos, uma rede de Petri colorida é construída com base na definição abaixo.

Definição 3.7 (Rede de Petri Colorida-*Coloured Petri Net-CPN*) .
Uma rede de Petri colorida (CPN) é definida pela tupla $(\Sigma, P, T, A, N, C, G, E, I))$*, onde:*
(i) Σ *é um conjunto finito de tipos, onde cada elemento é um "conjunto de cores";*
(ii) P *é um conjunto finito de lugares;*
(iii) T *é um conjunto finito de transições;*
(iv) A *é um conjunto finito de arcos, tal que* $P \cap T = P \cap A = T \cap A = \phi$*;*
(v) N *é uma função de nó (node), definida como* $N : A \rightarrow P \times T \cup T \times P$ *;*
(vi) C *é uma função de "cor" (colour), definida como* $C : P \rightarrow \Sigma$*;*
(vii) G *é uma função de guarda (função predicado), definida como* $G : T \rightarrow EXP$*, onde* EXP *é um conjunto de expressões tal que:*
$\forall t \in T | Type(G(t)) = bool$ *e* $Type(Var(G(t))) \subseteq \Sigma$*, onde* $bool \in \{true, false\}$*;*

(viii) E é uma função de expressões de arcos, definida como $E : A \rightarrow EXP$*, tal que* $\forall a \in A|Type(E(a)) = C(p)_{MS}$[3] *e* $Type(Var(E(a))) \subseteq \Sigma$*. Onde* $N(a) = p$*;*
(ix) I é uma função de inicialização, definida como $I : P \rightarrow CloseEXP$*, tal que* $\forall p \in P|Type(I(p)) = C(p)_{MS}$*, onde* $CloseEXP$ *é uma expressão sem variáveis.*

Analisando a definição acima, observa-se que:

- o conjunto de tipos, item (i), determina os possíveis valores de dados, bem como as operações e funções em expressões da rede (expressões de arcos, guardas e inicializações). Se for desejável, os tipos e suas operações podem ser definidos por meio da teoria de tipos abstratos. O modelo sintático da rede assume que cada tipo possui ao menos um elemento na rede.

- Os lugares, transições e arcos, itens (ii), (iii) e (iv), são descritos por três conjuntos, P, T e A, necessariamente finitos e desconexos. Estabelecendo tais conjuntos como conceitualmente finitos, inúmeros problemas de implementação de ferramentas são evitados [39].

- Nós, item (v), são elementos de rede conectados por arcos, ou seja lugares e transições. A função de nó mapeia cada arco em pares cujo primeiro elemento é a entrada e cujo segundo é a saída, o que estabelece o sentido do arco. É permitido a uma CPN possuir vários arcos associados ao mesmo par de nós.

- A função de cor, C, item (vi), mapeia cada lugar p a um tipo $C(p) \in \Sigma$. Isso significa que cada lugar p comporta *tokens* que possuem valores do tipo $C(p)$.

- A função de guarda, G item (vii), mapeia cada transição em uma expressão booleana, na qual todas as variáveis tem tipos pertencentes ao conjunto de tipos da rede.

- Cada expressão de arco, item (vii), é mapeada em um tipo $C(p)_{MS}$, significando que cada expressão de arco deve ser avaliada em função do tipo de seu lugar adjacente.

- A função de inicialização, item (ix), mapeia cada lugar em um conjunto de expressões sem variáveis (*closed expression*), o qual possui tipos pertencentes ao conjunto de tipos da rede.

[3] O subescrito MS significa que C(p) possui o mesmo tipo (cor) dos possíveis *Multiconjuntos* do modelo.

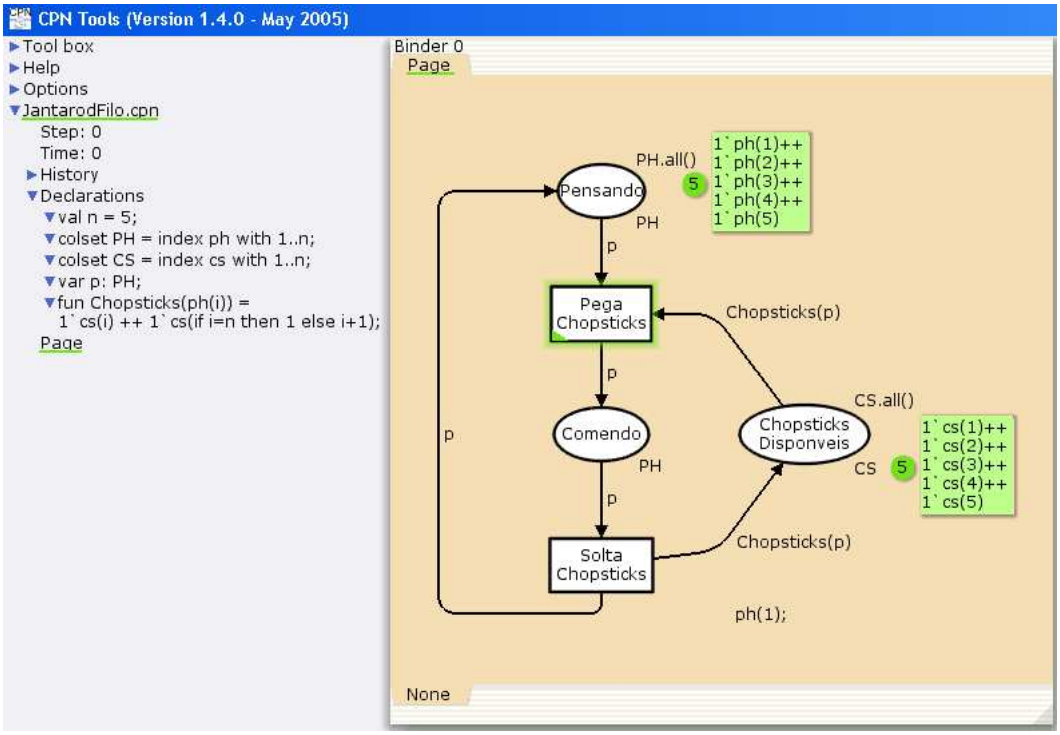

Figura 3.7: Jantar dos filósofos com rede de Petri colorida

A Figura 3.7 apresenta o problema do jantar dos filósofos modelado em uma rede de Petri colorida. Observa-se que capacidade e distinção dos *tokens* pelos tipos permite que uma mesma estrutura de rede modele diferentes padrões de comportamento a depender dos *tokens* de entrada. No ambiente CPNTools o conjunto de constantes, variáveis, funções e tipos são declarados em um espaço próprio, distinto do gráfico da rede (vide lado direito da Figura 3.7). No modelo do jantar dos filósofos, são definidos dois tipos (*colorset*) de *tokens*: PH e CS, identificando, respectivamente, filósofos e *chopsticks*. Ambos são definidos pelo tipo indexado presente na linguagem CPN-ML. O tipo indexado compreende uma seqüência de valores inteiros, em que o próprio valor é um índice de acesso. No modelo, $ph(2)$ representa o filósofo 2 enquanto $cs(4)$ representa o *chopstick* 4, ou seja, neste caso, a definição de cores (valores) é utilizada para distinguir elementos da coleção de valores que possuem o mesmo tipo de dados (conjunto de cores), o tipo indexado. De forma a distinguir os estados locais referentes aos processos associados aos filósofos (pensando e comendo) e a situação dos recursos (*chopsticks* disponíveis), são definidos os lugares: *pensando, comendo* e *chopsticks disponíveis*. Os eventos que determinam esses estados são modelados pelas transições: *pega chopsticks* e *solta chopsticks*. A variável p do "conjunto de cor" PH opera como um canal de dados entre lugares quando usada como expressão nos arcos que definem a seqüência de estados assumidos pelos filósofos, e como parâmetro de

alocação de recursos (*chopsticks*) quando usada como variável de entrada da função *Chopsticks(p)*, associada aos arcos do lugar *chopsticks disponíveis* (vide Figura 3.7). A função *Chopsticks(p)* mapeia cada filósofo em dois *Chopsticks* específicos. Pela construção do espaço de estados, verifica-se que apenas dois filósofos podem comer simultaneamente.

Em ferramentas tais como o DesignCPN e CPNTools, uma transição pode ter o seu disparo associado à execução de uma descrição em CPN-ML, que processa o valor do *token*. Transições que possuem código associado são denominadas *Transições de Código* (*Code Transitions*) e os códigos CPN-ML associados são referidos como *Segmentos de Código* (*Code Segments*) [38]. A função que relaciona transições com segmentos de código não é incluída na definição formal das rede de Petri coloridas.[4] Contudo, ela será necessária no Capítulo 4, quando da definição do modelo proposto neste trabalho. A função usada no Capítulo 4 é denominada *Função de Associação de Código-FCod*.

Definição 3.8 (Função de Associação de Código-*FCod*) .
Seja T um conjunto de transições e $cSegCod$ um conjunto de segmentos de código. Em uma rede de Petri colorida, uma função de associação de código é definida como $FCod : T \rightarrow cSegCod$. $FCod$ admite descontinuidade, ou seja, pode existir um T_i com $FCod(T_i)$ não definido.

Dado um modelo descrito em redes de Petri coloridas, os segmentos de códigos podem ser usados de diversas formas, por exemplo: (i) aplicação de descrições algorítmicas, (ii) geração de perfil estatístico durante a simulação da rede (iii) e geração de interfaces entre ferramentas. Embora os segmentos de código sejam muito úteis, sua aplicação impõe limitações à análise da rede [38]. Isto se deve ao fato de sua aplicação permitir a ocorrência de efeitos colaterais (*side effects*), uma vez que ele possibilita a mudança de variáveis internas do modelo. Esse fato estabelece limitações na análise do gráfico de ocorrência, uma vez que o comportamento do modelo não mais está restrito às regras presentes nos arcos e nas guardas das transições. Contudo, para abordagens centradas em simulação, a aplicação dos segmentos de código estabelece grande praticidade e flexibilidade de descrição.

Redes de Petri Coloridas Hierárquicas

Uma característica de particular importância das redes de Petri coloridas é o suporte a estruturas hierárquicas. As redes de Petri coloridas hierárquicas possibilitam que

[4]Possivelmente por está mais relacionada a uma característica de ferramenta do que ao modelo de rede propriamente dito.

transições e lugares "encapsulem" modelos complexos [38]. Embora lugares e transições possam embutir estruturas hierárquicas, ferramentas CPN, como o Design/CPN e o CPNTools, suportam apenas transições hierárquicas. Transições com tal capacidade de encapsulamento são denominadas *transições de substituição* (*Substitution Transitions*). O modelo encapsulado pela transição é denominado *sub-página* (*subpage*), enquanto o modelo "encapsulador" é denominado *super-página*. A aplicação de redes hierárquicas possibilita a estratificação da descrições bem como a utilização de diferentes níveis de abstração na definição dos elementos do sistema modelado. A seguir será definido o conceito formal das redes de Petri coloridas hierárquicas (HCPN-*Hierarchical Coloured Petri Nets*) [38].

Definição 3.9 (Redes de Petri Coloridas Hierárquica -HCPN) .
Sejam I e R conjuntos de índices. Uma rede de Petri colorida hierárquica é definida pela tupla HCPN=$\{S, SN, SA, PN, PA, FS, FT, PP)$, onde:
(i) $S = \{S_i | i \in I\}$ é um conjunto finito de páginas tal que:

1. *Cada* ***Página*** S_i *é uma CPN não-hierárquica, tal que:*

$$Si = (\Sigma_i, P_i, T_i, A_i, N_i, C_i, G_i, E_i, N_i).$$

2. *Os conjuntos de elementos de rede são pares disjuntos, tal que:*

$$\forall (i,k) \in I, i \neq k \Rightarrow (P_i \cup T_i \cup A_i) \cap (P_k \cup T_k \cup A_k) = \phi.$$

(ii) $SN \subseteq T$ é um conjunto de ***nós de substituição****.*
(iii)SA é uma ***função de atribuição****, $SA : SN \rightarrow S$, tal que:*

1. *Nenhuma Página é sub-página de si mesmo:*

$$\left\{i_0, i_1 ... i_n \in I^* | n \in N \wedge i_0 = i_n \wedge \forall k \in 1..n : S_{i_k} \in \left\{SA(SN_{i_{k-1}})\right\}\right\} = \phi.$$

(iv) $PN \subseteq P$ é um conjunto de ***nós de portas****. (v) PA é uma função de* ***atribuição de portas****, PA mapeia de SN em uma relação binária tal que:*

1. ***Nós de soquetes*** *estão relacionados aos* ***nós de portas****:*

$$PA(x) \subseteq X(x) \times PN_{SA(x)}.$$

Onde x é um nó de substituição ($x \in SN$), $X(x)$ é uma função que retorna uma partição de P, ou seja um conjunto de lugares, representando lugares ligados a x, sendo esses lugares denominados soquetes. $PN_{SA(x)}$ representa o conjunto de nós

de portas (lugares) presentes na Página $SA(x)$. Logo a função $PA(x)$ retorna um conjunto de tuplas onde são relacionados os pares de lugares (soquete $\times$ portas) para um dado nó de substituição x.

2. *Nós relacionados possuem conjuntos de cores idênticos e expressões de inicialização equivalentes:*

$$\forall x \in SN, \forall (p_1, p_2) \in PA(x) : [C(p_1) = C(p_2) \wedge I(p_1) = I(p_2)].$$

(vi) $FS = \{FS_r\}_{r \in R}$ é um conjunto finito de ***conjuntos de fusões*** *tal que:*

1. *FS é uma partição de P.*

2. *Membros de uma conjunto de fusões possuem o mesmo conjunto de tipos (colour sets) e expressões de inicialização equivalentes:*

$$\forall r \in R, \forall p_1, p_2 \in FS_r : [C(p_1) = C(p_2) \wedge N(p_1) = N(p_2)].$$

(vii) FT é uma função de ***fusão de tipos****, tal que:*

1. *Cada fusão é do tipo:* ***global****,* ***page*** *ou* ***instance****.*

2. *Os conjuntos de fusão do tipo* ***page*** *e* ***instance*** *pertencem a uma única Página:*

$$\forall r \in R : [FT(FS_r) \neq global \Rightarrow \exists i \in I : FS_r \subseteq P_i].$$

(viii) $PP \in S_{MS}$ é o multiconjunto das ***Prime Pages****.*

Uma HCPN é básicamente um conjunto de CPNs não hierárquicas, denominadas **Páginas** - item (i) tópico 1 da Definição 3.9 - interligadas por funções de vínculo, SA e PA - item (iii) e (v). A função SA vincula transições às Páginas, desde que nenhuma transição seja vinculada à Página que a contém - item (iii), tópico 1. Tal vínculo faz com que a transição represente o comportamento da rede vinculada. As transições que pertencem ao domínio de SA são classificadas como **nós de substituição** (SN) ou transições de substituição (*Substitution Transitions*)- item (ii). O termo nó de substituição denota uma entidade de uma rede que representa, "substitui", toda uma outra rede de uma hierarquia inferior do modelo, podendo, portanto, referir-se a lugares também, embora não seja usual. A função PA vincula lugares de uma transição de substituição aos lugares da rede representada por ela, onde os lugares ligados à transição de substituição são denominados **nós de soquetes** e os lugares pertencentes a rede vinculada são denominados **nós de portas** (PN)- item (v) tópico 1. A função é restrita

a soquetes e portas que possuam a mesma cor e expressões de inicialização equivalentes - item (v) tópico 2. Adicionalmente, a função PA pode relacionar um soquete a várias portas e vice-versa [38].

Conjuntos de fusões (FS) são partições de P que agregam lugares *fundidos* de forma a representarem um lugar presente em várias Páginas. Portanto, tais lugares possuem a mesma cor e expressões de inicialização equivalentes, o que significa que um lugar pode pertencer apenas a um conjunto de fusões - item (vi). Uma vez que uma HCPN é um conjunto de CPN's (Páginas), a construção de um modelo em HCPN pode significar a aplicação de várias instâncias de uma mesma rede (Página). Portanto, o conjunto de Páginas estabelece um multiconjunto. Uma **Prime Page** (PP) é um multiconjunto definido a partir do conjunto das páginas da HCPN - item (viii) -, que define as páginas de onde se deve iniciar a simulação do modelo [27]. Os **tipos de fusões** (FT) definem conjuntos de conjunto de fusões, classificando-os por tipos: **global**, **page** e **instance** - item (vii), tópico 1. Dizer que uma **fusão de tipo** é do tipo **global**, significa dizer que os conjuntos de fusões que ela define possuem lugares *presentes* em várias Páginas. As fusões do tipo **page** possuem conjuntos de fusões cujos lugares estão apenas nas instâncias de uma Página, enquanto as do tipo **instance** possuem lugares apenas em uma instância de Página específica - item (vii), tópico 2. Avaliando a definição da HCPN, observa-se que as CPNs não hierárquicas são casos particulares das HCPNs, onde existe apenas uma Página e não existe portas ou soquetes. A Prime Page é definida por uma única Página com uma única instância.

As Figuras 3.8, 3.10, 3.9, 3.11 e 3.12 mostram uma rede de Petri colorida hierárquica construída no ambiente da ferramenta CPNTools. Nessa rede, o conceito de hierarquia é aplicado na modelagem da operação lógica *ou exclusivo* (XOR). A operação *ou exclusivo* não é uma operação booleana primitiva, sendo, portanto, expressa por uma função lógica $XOR(A, B) = A * \bar{B} + \bar{A} * B$. A Figura 3.8 apresenta a versão mais abstrata do modelo, em que uma transição de substituição "encapsula" a operação XOR. Essa transição de substituição está vinculada à Página *Circuito XOR*, de forma que a operação é descrita em um nível mais baixo de abstração, através de um modelo de circuito lógico (vide Figura 3.10). A Página *Circuito XOR*, por sua vez, possui várias transições de substituição *encapsulando* as operações primitivas *inversora* (INV), *e lógico* (AND) e *ou lógico* (OR) (vide Figuras 3.9, 3.11, 3.12). Cada transição de substituição do modelo XOR encapsula detalhes do modelo, até atingir a descrição dos operadores primitivos, em que transições vinculadas a segmentos de código CPN-ML são aplicadas para descrever as operações.

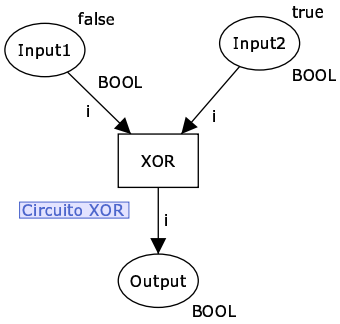

Figura 3.8: Modelo CPN Hierárquico de uma Porta XOR

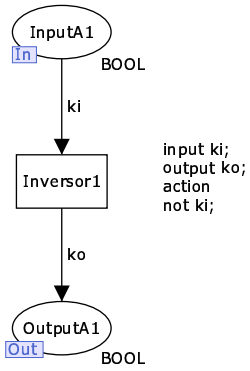

Figura 3.9: Modelo CPN: Porta Inversora.

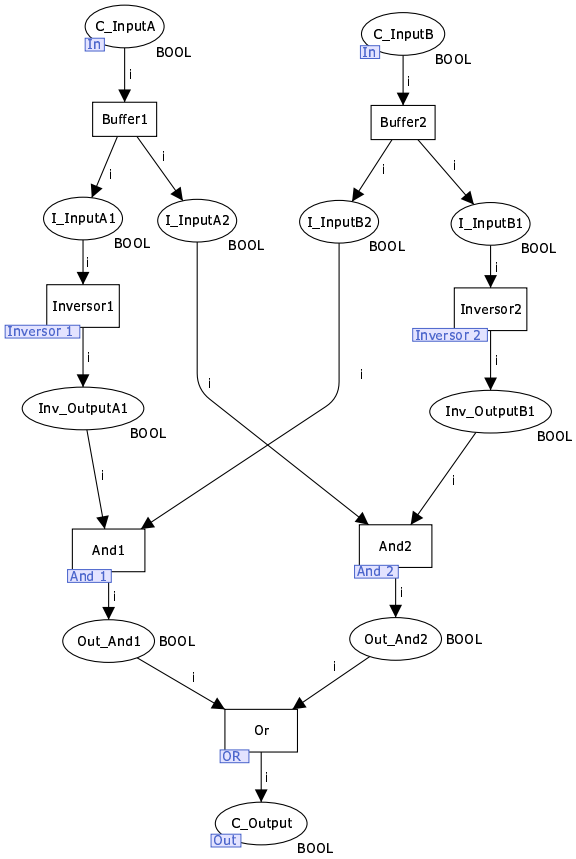

Figura 3.10: Modelo CPN: Circuito XOR

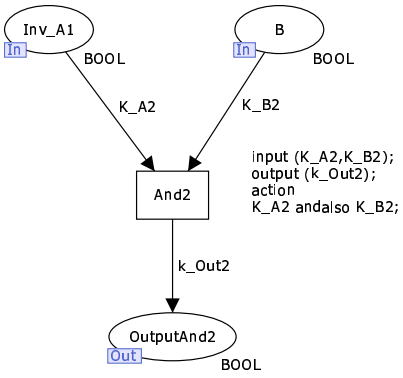

Figura 3.11: Modelo CPN: Porta AND.

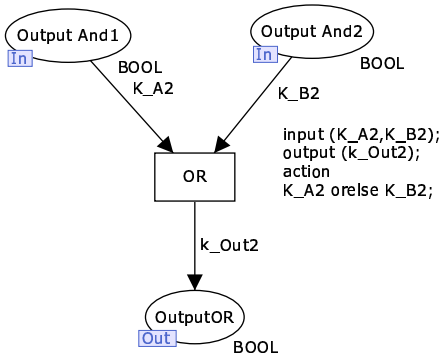

Figura 3.12: Modelo CPN: Porta OR.

3.8.5 Redes de Petri Orientadas a Objetos

Em uma busca por maior abstração, os *tokens* podem ser interpretados como objetos. Em redes Place/Transition, na maioria dos casos, esses objetos representam recursos ou indicam estados de controle. A solução para a representação de objetos mais complexos foi primeiramente introduzida com o uso de *tokens* tipados, nas redes de Petri coloridas. Em modelos orientados a objetos, associa-se comportamento ao "tipo", transformando-

o em outra entidade: o objeto. Do ponto de vista das redes de Petri, é natural associar tais objetos aos *tokens*, de forma que o comportamento é representado por uma rede de Petri (interna ao *token*). Essa abordagem tem sido apresentada como uma família de extensões comumente chamada de redes de Petri orientadas a objeto [104].

Em várias aplicações, os objetos são móveis. Exemplos de tais objetos são os agentes, como os módulos de *software* no contexto do paradigma de programação orientada a agentes. Tal abordagem permite modelar sistemas, como redes *wireless*, nos quais computadores móveis conectam-se para a realização de certas atividades e podem migrar entre ambientes. O mesmo computador (agente) pode se conectar a diferentes *links* seguros em diferentes redes. Um outro exemplo é a modelagem de uma mesma tarefa que pode/deve ser executada em uma máquina "A" ou "B". O objeto em questão nesse caso é a tarefa a ser executada na máquina "A" junto a um plano para a execução dos procedimentos, que podem ser transportados para a máquina "B". Por fim, pode-se citar também a modelagem de ações em um fluxo de trabalho (*workflow*), que flui de um empregado "E_A" em direção a um empregado "E_B". Observa-se que em todos os exemplos os estados internos do objeto foram alterados, durante o movimento para diferentes locações. Se os exemplos forem abstraídos em uma rede de Petri, o modelo de movimento entre locações pode ser representado por uma transição habilitada por *tokens*-objeto em seus lugares de entrada. Adicionalmente, pode-se atribuir a este objeto um comportamento dinâmico, modelado em uma rede marcada. Esse tipo de *token* é normalmente chamado *token*-rede ou objeto-rede (*token-net ou object-net*).

3.8.6 Redes de Petri *Fuzzy*

A extensão *fuzzy* das redes de Petri permite a descrição e simulação de redes neurais. Para tanto, Looney [56] estendeu as redes de Petri associando à transição o modelo do corpo de um neurônio, aos arcos de entrada o modelo de dendritos e ao arco de saída o modelo de axiônios. Para operar como um neurônio modelado como um ponderador de estímulos, o *token* passa a transportar uma variável *fuzzy* que pode assumir valores entre 0 e 1 (*fuzzy-token*). De forma a representar sistemas lógicos, a rede de Looney é proposta segura por definição. A regra de disparo das transições é modificada de forma que a transição dispara (ou está habilitada a disparar) quando ao menos um *fuzzy-token* de entrada apresentar valor maior que um valor limiar associado a transição. Para evitar a desabilitação devido a conflitos, Looney propôs que as transições não consumissem os *tokens*, mas apenas os copiassem. Isso é necessário, pois o modelo de Looney busca a avaliação de múltiplas condições simultâneas resultantes de um mesmo estímulo. Para definir tal distinção, Looney propôs seu postulado [56].

Postulado de Looney

Em uma rede (Petri) lógica, um nó (lugar) ativado passa cópias de seus tokens através de todos os arcos que partem dele. Um neurônio (transição) ativado passa cópias de seus tokens para todos os arcos que partem dele.

Em termos de lógica *fuzzy*, a rede de Looney é uma rede de avaliadores "*AND*" com uma entrada de habilitação - o valor *fuzzy* de limiar de disparo. Looney apresentou um algoritmo de avaliação da rede e demonstrou que mediante manipulação da estrutura e limiares de disparos da rede, regras de inferência podem ser modificadas minimizando erros, uma clássica abordagem para processos de aprendizagem em classificadores de sistemas.

Devido à melhor adequação do modelo *fuzzy* para a avaliação de sistemas de decisão do mundo real, as redes de Petri *fuzzies* têm proporcionado um excelente mecanismo de representação de conhecimento *fuzzy* [15, 6, 33, 23]. Bugarin *et al* [15] usaram as *Fuzzy Petri Nets* para a representação de regras encadeadas, adicionando avanços nas técnicas que lidam com raciocínios *fuzzies*. Chen [22] apresenta uma metodologia para verificação de bases de conhecimento através de avaliações em redes de Petri *fuzzies*. Chen [24] propõe um modelo em redes de Petri *fuzzies* para representar regras de produção fuzzies, e um algoritmo para implementação de raciocínios*fuzzies* automaticamente a partir de rede. Koriem [48] apresenta uma aplicação dos modelos *fuzzies* sobre redes de Petri *fuzzies*, no qual a gerência de bagagens de um aeroporto é modelada. A partir de redes de Petri *fuzzies*, redes semânticas podem ser construídas de forma que conhecimentos *ad hoc* possam ser facilmente capturados e analisados.

3.9 Considerações Finais

As redes de Petri constituem uma excelente ferramenta de modelagem, conjungando representação gráfica com formalização da análise e exploração de estados, além da capacidade de análise por simulação. Diversas extensões foram criadas permitindo o seu uso em diversos domínios relacionados a sistemas embarcados, tais como: sistemas especificados em nanoestruturas [81], estimativas de potência em sistemas síncronos [72] e assíncronos [55], modelagem de algoritmos de reconfiguração dinâmica de *hardware* [77], escalonamento de síntese de *software* [10, 9], escalonamento de *software* com restrição de potência [97], geração de estimadores para projetos de *Hard/Soft Codesign* [57, 59, 31] e modelagem de sistemas de conhecimento [24].

Capítulo 4

Modelo Determinístico

4.1 Introdução

Como apresentado no Capítulo 2, o comportamento do *software* é responsável pela dinâmica de consumo do processador, a despeito das otimizações implementadas no projeto de seu *hardware*. Objetivando a realização de estimativas do consumo de energia devido ao *software*, este trabalho propõe dois modelos: um de natureza determinística e outro probabilística. As abordagens propostas compreendem a modelagem do *software* como uma cadeia de fontes de consumo (instruções), cujo acionamento é encadeado no tempo em função do contexto de operação do sistema. Esta cadeia é representada por um modelo computacional baseado em *tokens*, expresso redes de Petri coloridas (CPN - *Coloured Petri Nets*).

O modelo determinístico proposto encadeia as instruções em uma rede CPN, cada elemento de consumo (instrução) é acionado de acordo com o fluxo do programa. A rede resultante representa o *software* sob uma óptica estrutural, identificando explicitamente laços, aninhados e subrotinas. Cada elemento da cadeia representa uma instrução, cujo consumo está associado à porção de *hardware* usada por ela. Como discutido no Capítulo 2, o modelo de consumo da instrução pode ser interpretado como macro-modelos de consumo do *hardware* (vide Figura 4.1). O código é modelado em uma rede de Petri colorida segura, na qual cada estado possível do processador, em termos de acesso às instruções, é modelado como um lugar e o contexto interno do processador [1] como uma estrutura de dados no *token*. O disparo das transições representa o processamento das instruções. Quando uma transição processa o *token*, consumindo e gerando *tokens*, modifica os estados internos da estrutura de dados do *token*, ou seja, modifica o contexto interno do processador exatamente como uma instrução. A

[1] Valores de registradores, memórias internas e *flags*

Figura 4.2 ilustra esse modelo. Devido à característica hierárquica das redes de Petri coloridas, as transições podem embutir novas redes que descrevem detalhadamente o comportamento da instrução em diferentes níveis de abstração. Sob a interpretação proposta, essa descrição pode chegar à descrição do *hardware* associado à instrução modelada, por meio do uso do modelo proposto por Murugavel [72, 73]. Este trabalho, contudo, propõe uma abordagem baseada na descrição comportamental das instruções. Modelos foram especificados em ferramentas CPN usando CPN-ML (sub-conjunto SML implementado nos ambientes Design/CPN e CPNTool).

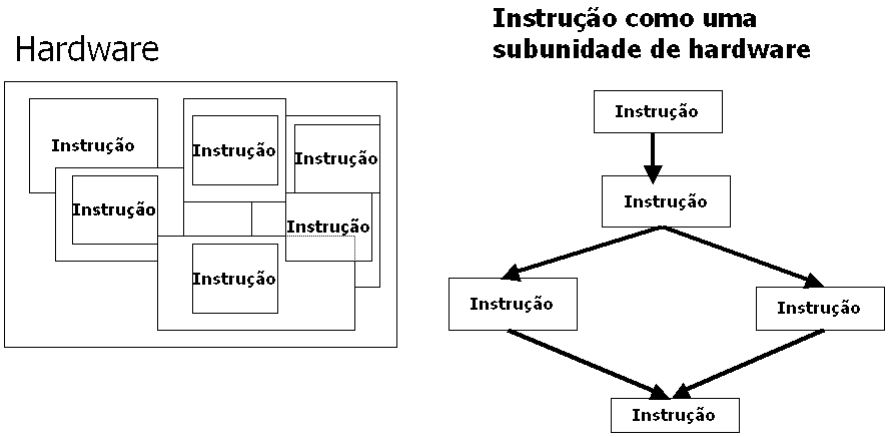

Figura 4.1: Interpretação da execução de um programa sob a óptica do consumo de energia

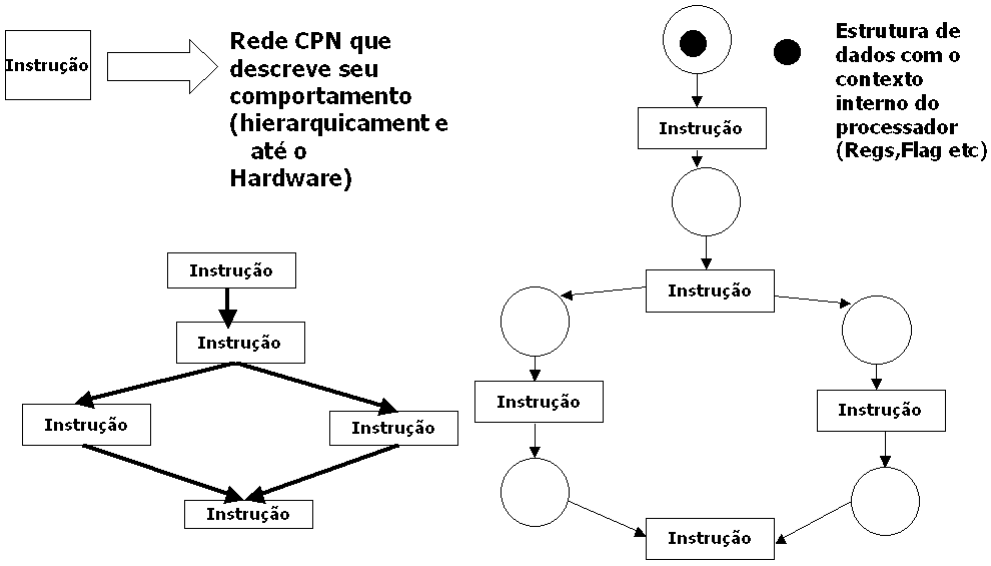

Figura 4.2: Modelamento do *software* em redes de Petri coloridas

Devido aos recursos de descrição e simulação presentes nas ferramentas que supor-

tam CPN, é possível simular integralmente o comportamento do *software*. Constrói-se com isso, não um simulador estruturado na arquitetura de *hardware* do processador, mas um mecanismo de simulação baseado no fluxo de execução do código de máquina. Os mecanismos de simulação e análise resultantes da aplicação de modelos CPNs não são específicos de uma arquitetura de *hardware* ou de um conjunto de instruções, mas sim de uma representação mais abstrata do sistema. O mecanismo de simulação não simula um processador, mas sim uma rede de Petri. Além do recurso de simulação, a modelagem em rede de Petri colorida estabelece um modelo estrutural do código, possibilitando mecanismos de pesquisa/reconhecimento de estruturas. Como será mostrado adiante, a geração de perfis de execução e consumo de energia é naturalmente realizada pelo modelo de execução, caracterizando as regiões do código mais executadas e suas contribuições para o consumo de energia.

Em contraste com modelos de processadores em CPN propostos em trabalhos como [19] e [18], baseados em descrições da arquitetura interna do processador, o modelo proposto nesta tese baseia-se unicamente no conjunto de instruções. Modelos para análise de potência baseados no *hardware* precisam ser carregados com detalhes do *hardware* e suas características de consumo, que não são livremente disponíveis para o projetista [60]. O modelo proposto, por outro lado, necessita apenas de um modelo de potência das instruções do processador em questão. Esse modelo pode ser obtido por medidas realizadas no dispositivo físico. Adicionalmente, os modelos CPN construídos podem ser interpretados como uma especificação executável do conjunto de instruções de um processador. Instruções podem ser inseridas ou retiradas caracterizando novas versões de processadores, dentro de uma mesma família. O simples *instanciamento* da rede[2] em uma ferramenta de simulação, gera automaticamente um simulador de instruções, o que faz o modelo comportar-se como uma linguagem de descrição de arquiteturas de conjunto de instruções.

A Figura 4.3 ilustra a metodologia de análise proposta. A metodologia consiste em converter o código executável (código binário) em um modelo CPN, simular a rede e analisar a massa de dados gerada por ela. Neste trabalho, a ferramenta CPNTools foi utilizada para a construção dos modelos de instrução e validação dos modelos de código. Um compilador Binário-CPN foi implementado para converter o código executável em uma rede de Petri colorida no formato de entrada da ferramenta CPNTools. O ambiente EZPetri[3] foi estendido de forma a processar os dados gerados pelo modelo, oferecendo

[2]Criação de uma rede referente a um programa executável.

[3]Ambiente de integração de modelos sobre redes de Petri, desenvolvido pelo Departamento de Sistemas Computacionais da Escola Politécnica de Pernambuco-UPE.

uma interface adequada à análise de consumo de energia. O compilador Binário-CPN é carregado com descrições dos recursos de *hardware* e com modelos CPN do conjunto de instruções da arquitetura alvo. Esses dois conjuntos definem o modelo CPN da arquitetura. Devido a auto-contenção do modelo de instrução, a validação do modelo de instrução pode ser realizada isoladamente. As funções de análise geram resultados referentes às estimativas de perfil de consumo, perfil de execução e acesso à memória.

Este trabalho formula uma metodologia de modelagem para arquiteturas sem-*pipeline*. Tendo sido usada como estudo de caso a arquitetura i8051, consagrada arquitetura de micro-controlador, presente em inúmeros sistemas embutidos. Adicionalmente, as bases do modelo para arquiteturas *simplescalar* são definidas. Objetivando formalizar o processo de análise, foi criado um conjunto de entidades de análise e definições que classificam os trechos de código de acordo com seu padrão de execução e consumo. A taxonomia proposta foi apresentada em [46]. A taxonomia, bem como o modelo do i8051 e o possível modelo para arquiteturas *simplescalar*, serão discutidos nas seções seguintes.

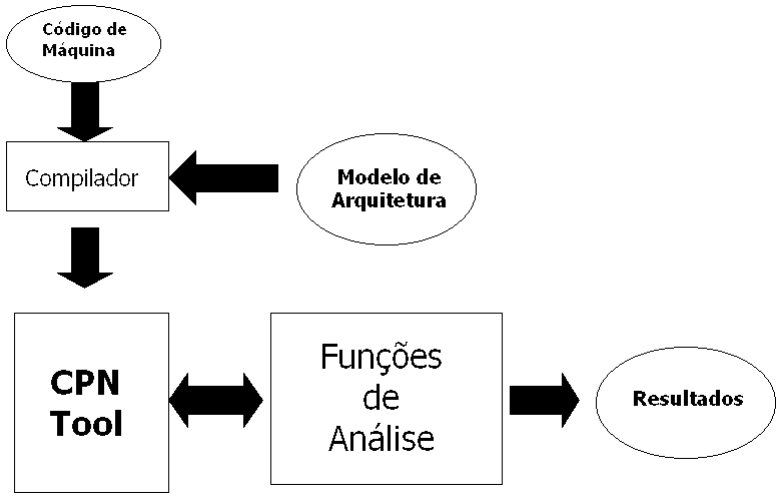

Figura 4.3: Metodologia Proposta

4.2 Modelo CPN de Descrição

O modelo CPN de arquitetura é definido por dois conjuntos: o conjunto de modelos CPN de instrução e o conjunto de recursos de *hardware*. O modelo CPN de uma instrução é uma rede de Petri colorida que opera como uma especificação executável da instrução, na qual lugares representam os estados anteriores, internos e posteriores à execução da instrução, e transições representam conjuntos de operações associadas à

instrução. Nessa abordagem, uma operação é definida como um conjunto de eventos em *hardware* que caracteriza total ou parcialmente uma instrução. O modelo CPN de instrução é construído de forma que, em ao menos uma transição, exista um código que descreve o comportamento da instrução por meio de chamadas aos recursos de *hardware*, ou a um modelo CPN de operação. Os recursos de *hardware* são funções que executam operações específicas solicitadas pelas instruções. Dependendo do nível de abstração do modelo CPN essas operações podem estar associadas à uma ou mais unidades funcionais do *hardware*, ou mesmo a sub-unidades. O modelo CPN de operação é uma rede que representa em seu conjunto de transições e lugares, as ações e os estados relacionados às operações realizadas pela instrução, respectivamente. Cada transição do modelo CPN de operação pode representar um outro modelo CPN de operação, que descreve, em menor nível de abstração, um sub-conjunto do eventos de *hardware* associados à instrução. Dessa forma, o modelo pode descrever a instrução em diversos níveis de abstração, podendo chegar até à modelagem dos circuitos lógicos afetados pela instrução, onde modelos CPN de operação modelam portas lógicas. O modelo de instrução descreve o comportamento da instrução em função das transformações que a instrução promove no contexto interno do processador. O contexto interno é o conjunto de elementos de armazenamento, tais como registradores, memórias e *flags* internos, acessíveis ou não pelo código (*storables*). O modelo CPN é formalmente definido a seguir.

Definição 4.1 (Contexto Interno-CI) .
Estrutura de dados que representa o conjunto de elementos de armazenamento, tais como registradores, memórias e flags internos, acessíveis ou não pelo código. O tipo dessa estrutura de dados define o tipo Context.

Definição 4.2 (Instrução) .
Conjunto finito de operações de hardware associado a um código numérico (Opcode).

Uma instrução define um padrão de mudanças no CI.

Definição 4.3 (Recurso de Hardware-RH) .
Algoritmo que modela um conjunto de operações de hardware.

cRH representa o conjunto de RHs de uma descrição.

Definição 4.4 (Código de Instrução-CodInst) .
Descrição de instrução por meio de algoritmo que invoca funções presentes no cRH. Um conjunto de códigos de instrução de uma arquitetura é definido por $cCodInst =$

$\{cCodInst_{Name}|Name \in cMnem\}$, *onde* $cMnem$ *é o conjunto de mnemônicos associados às instruções da arquitetura.*

Definição 4.5 (Aplicação de Código-ApCod) .
Código ou segmento de código assembly, definido como um conjunto ordenado de instruções.

Definição 4.6 (Função Interna-FI) .
Algoritmo que implementa ações internas ao modelo.

cFI representa o conjunto de FIs de uma descrição.

As funções internas são responsáveis por eventos internos ao modelo de simulação, não estando relacionadas às operações da aplicação modelada. As funções internas implementam a inicialização do modelo, verificação dos critérios de parada, marcação de sondas de análise e geração de arquivos.

O modelo CPN para a descrição de arquiteturas de processadores é construído com base em dois construtores de redes: o modelo de desvio condicional e o modelo ordinário. O construtor de desvio condicional é usado para as instruções de desvio condicional enquanto o construtor ordinário para todas as demais. Dessa forma, os modelos CPN de instruções são divididos em duas categorias: modelo CPN de desvio condicional e modelo CPN ordinário. Define-se $cInst$ e $cInstDesvCond$ como o conjunto de todas as instruções e das instruções de desvio da arquitetura alvo ($cInstDesvCond \subseteq cInst$),respectivamente.

Definição 4.7 (Modelo CPN de Instrução Ordinária-MCIOrd) .
Seja $Inst_n|Inst_n \notin cInstDesvCond$, *a rede de Petri colorida descrita pela tupla* $MCIOrd_n = (\Sigma, P, T, A, N, C, G, E, I)$, *define um modelo CPN de instrução de* $Inst_n$, *onde:*
(i) $\Sigma = \{Context\}$,
(ii) $P = \{S\_Input, S\_Output\}$,
(iii) $T = \{InsName_i|FCod(InsName_i) = cCodInst_{Name}\}$,
(iv) $A = \{a_0, a_1\}$,
(v) $N(a_0) = (S\_Input, InsName_i)$ *e* $N(a_1) = (InsName_i, S\_Output)$,
(vi) $\forall P_k \in P, C(P_k) = Context$,
(vii) $G(InsName_i) = true$,
(viii)$E(a_0) = K_i$, $E(a_1) = K_o|Type(K_o) = Type(K_i) = Context$ *e*
(ix) $I(S\_Input) = I(S\_Output) =$ *não definido.*

A Figura 4.4 ilustra a estrutura do modelo CPN de instrução ordinária. A rede é constituída por um lugar de entrada (S_Input), um lugar de saída (S_Output) e uma transição ($InsName_i$), nomeada de acordo com a instrução modelada ($Name$). O índice de instanciamento, i, identifica a instância de instrução no modelo CPN de aplicação, que será apresentado na Definição 4.17. A operação de atribuição de código, item (iii), estabelece que a transição está vinculada a um código de instrução ($CodInst$), descrevendo o comportamento da instrução em função do contexto interno, aqui representado pelo valor $Value$.

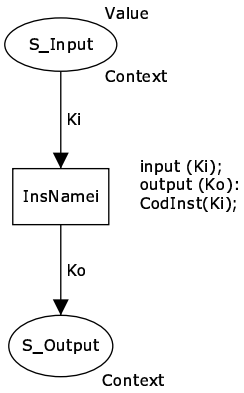

Figura 4.4: Estrutura do Modelo CPN de Instrução Ordinária

Definição 4.8 (Modelo CPN de Instrução de Desvio Condicional-MCICon)
.
Seja $Inst_n | Inst_n \in cInstDesvCond$*, a rede de Petri colorida descrita pela tupla* $MCICon_n = (\Sigma, P, T, A, N, C, G, E, I)$ *define um modelo CPN de instrução de* $Inst_n$*, onde:*
(i) $\Sigma = \{Context\}$*,*
(ii) $P = \{S\_Input, Inter\_State, S\_Output1, S\_Output2\}$*,*
(iii) $T = \{Jump, NotJump\} \cup \{InsName_i | FCod(InsName_i) = cCodInst_{Name}\}$*,*
(iv) $A = \{a_k | 0 \leq k \leq 5\}$*,*
(v) $N(a_0) = (S\_Input, InsName_i)$*,*
$N(a_1) = (InsName_i, Inter\_State)$*,* $N(a_2) = (Inter\_State, Jump)$*,*
$N(a_3) = (Inter\_State, NotJump)$*,* $N(a_4) = (Jump, S\_Output1)$*,*
$N(a_5) = (NotJump, S\_Output2)$*,*
(vi) $\forall P_k \in P, C(P_k) = Context$*,*
(vii) $G(InsName_i) = true$*,* $G(Jump) = CheckJump()$*,*$G(NotJump) = not(CheckJump())$*, onde* $CheckJump() \in cRH$*,*

(viii)$E(a_0) = K_i$ *e* $\forall a_j \neq a_0$, $E(a_j) = K_o$, *onde* $Type(K_o) = Type(K_i) = Context$ *e (ix)* $I(S\_Input) = I(Inter\_State) = I(S\_Output1) = I(S\_Output2) =$ *não definido.*

A Figura 4.5 ilustra a estrutura do modelo CPN de instrução de desvio condicional. A rede é constituída de um lugar de entrada (S_Input), um lugar interno ($Inter\_State$) e dois lugares de saída ($S\_Output1$ e $S\_Output2$). Cada lugar de saída descreve uma alternativa de fluxo de execução, definida pela condição avaliada pelo código de instrução ($CodInst$), associado à transição $InsName_i$ e validada pelas guardas das transições $Jump$ e $NotJump$. A validação de um desvio compreende a leitura de um $flag$ de $hardware$, sendo portanto definida por uma função predicado $CheckJump() \in cRH$.

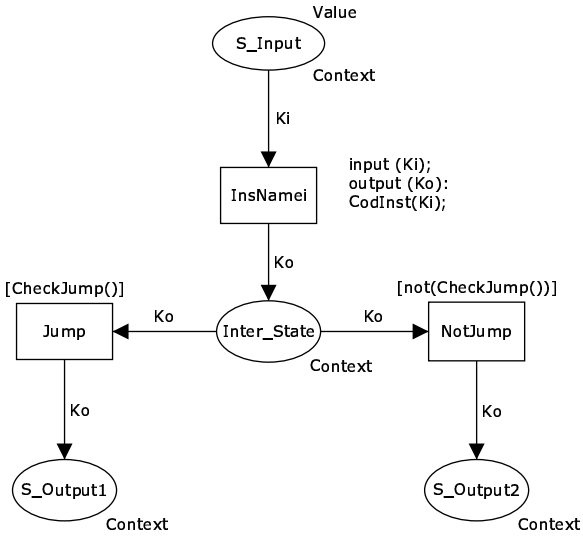

Figura 4.5: Estrutura do Modelo CPN de Instrução de Desvio Condicional

Definição 4.9 (Modelo CPN de Instrução-MCI) .
Seja uma instrução $Inst_n$ *e a rede de Petri colorida descrita pela tupla* $MCI_n = (\Sigma, P, T, A, N, C, G, E, I) | MCI_n \in (\{MCIOrd_k\} \cup \{MCICon_k\})$, MCI_n *é definida como modelo CPN de instrução de* $Inst_n$.

De forma a estabelecer um mecanismo hierárquico para construção do modelo de descrição, define-se um segundo construtor denominado procurador de instrução, $Proxy$. O $Proxy$ é uma CPN que representa um MCI no modelo HCPN de aplicação, que será apresentado na Definição 4.17. Dados os dois tipos de $MCIs$, define-se dois tipos de $Proxys$: ordinário e de desvio condicional.

Definição 4.10 (Proxy Ordinário-ProxyOrd) .
Seja $MCIOrd_n$ *um modelo CPN de instrução ordinária, a rede de Petri colorida*

descrita pela tupla $ProxyOrd_n = (\Sigma, P, T, A, N, C, G, E, I)$ *define seu procurador ordinário, onde:*

(i) $\Sigma = \{Context\}$,
(ii) $P = \{Input, Output\}$,
(iii) $T = \{InsPName_i\}$,
(iv) $A = \{a_0, a_1\}$,
(v) $N(a_0) = (Input, InsPName_i)$ *e* $N(a_1) = (InsPName_i, Output)$,
(vi) $\forall P_k \in P, C(P_k) = Context$,
(vii) $G(InsPName_i) = true$,
(viii)$E(a_0) = E(a_1) = i|Type(i) = Context$ *e*
(ix) $I(Input) = I(Output) =$ *não definido.*

Definição 4.11 (Proxy de Desvio Condicional-ProxyDesv) .
Seja $MCICon_n$ *um modelo CPN de instrução de desvio, a rede de Petri colorida descrita pela tupla* $ProxyDesv = (\Sigma, P, T, A, N, C, G, E, I)$ *define seu procurador de desvio, onde:*

(i) $\Sigma = \{Context\}$,
(ii) $P = \{Input, Output1, Output2\}$,
(iii) $T = \{InsPName_i\}$,
(iv)$A = \{a_0, a_1, a_2\}$,
(v)$N(a_0) = (Input, InsPName_i)$, $N(a_1) = (InsPName_i, Output1)$,
$N(a_2) = (InsPName_i, Output2)$,
(vi)$\forall P_k \in P, C(P_k) = Context$,
(vii)$G(InsPName_i) = true$,
(viii) $E(a_0) = E(a_1) = E(a_2) = i|Type(i) = Context$ *e*
(ix) $I(Input) = I(Output1) = I(Output2) =$ *não definido.*

Definição 4.12 (Proxy CPN de Instrução-Proxy) .
Seja uma instrução $Inst_n$ *e a rede de Petri colorida descrita pela tupla* $Proxy\_n = (\Sigma, P, T, A, N, C, G, E, I)|Proxy\_n \in (\{ProxyOrd_k\} \cup \{ProxyDesv_k\})$.
$Proxy\_n$ *é definido como procurador CPN de instrução de* $Inst_n$.

As Figuras 4.6 e 4.7 mostram as estruturas de redes representativas dos procuradores ordinários e de desvio condicional. O procurador de instrução estabelece o elemento básico de construção do modelo CPN do código. Para tanto, o primeiro passo é vincular as instâncias de instrução do código alvo em procuradores de instrução. De forma a

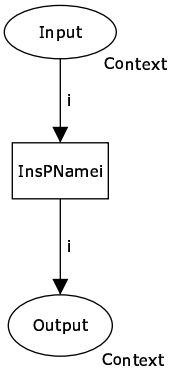

Figura 4.6: Estrutura do Proxy Ordinário.

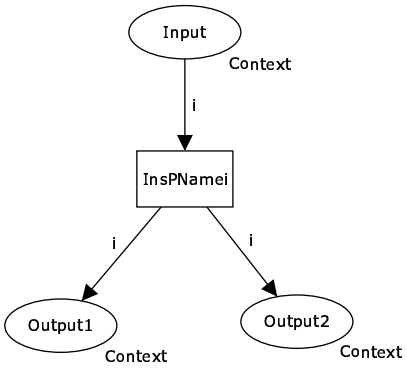

Figura 4.7: Estrutura do Proxy de Desvio Condicional.

gerar a seqüência de execução, inclusive os desvios condicionais, define-se uma função de fluxo. Essa função gera um conjunto ordenado de modelos procuradores em função das instruções presentes no código alvo. Os lugares dos *procuradores* são renomeados em função dos endereços associados à instrução, sendo eles: endereço da instrução ($InstMem_k^n$), endereço da próxima instrução ($InstMem_{k+1}^n$) e endereço alvo de desvio ($EndAlv_k$). A renomeação opera com quatro focos (vide Figuras 4.6 e 4.7).

1. Instruções de desvio incondicional.
   O lugar $Input$ é renomeado por $InstMem_k^n$ e $Output$ é renomeado por $EndAlv_k$.

2. Instruções de desvio incondicional *dinâmico*[4].
   O lugar $Input$ é renomeado por $InstMem_k^n$ e $Output$ é renomeado por P_{in}.

3. Instruções de desvio condicional.
   O lugar $Input$ é renomeado como $InstMem_k^n$, $Output1$ é renomeado $EndAlv_i$ e $Output2$ como $InstMem_{k+1}^n$,

[4]Aquelas cujo endereço alvo do desvio é definido em tempo de execução.

4. Demais instruções.
   O lugar *Input* é renomeado $InstMem^n_k$ e *Output* como $InstMem^n_{k+1}$.

Lugares de saída de instruções de desvio *dinâmico* são renomeados como P_{in}.

Definição 4.13 (Função de Renomeação-F_{Rn}) .
Seja $ApCod_n$ uma aplicação de código, $InstMem^n$ o conjunto de endereços de memória das instruções em $ApCod_n$, $F_{Ext}(I_i, End_i)$ uma função que fornece o endereço de desvio da instância de instrução I_i, que está alocada na posição de memória $End_i \in InstMem^n$. Seja ainda $P^{Proxy} = \{Input, Output, Output1, Output2\}$ o conjunto de possíveis lugares de um procurador de instrução e $cEnd = \{InstMem^n_k\} \cup \{P_{in}\}$ um conjunto de possíveis nomeações dos lugares das redes Proxy associadas à $ApCod_n$. $F_{Rn} : ApCod_n \times InstMem^n \times P^{Proxy} \rightarrow cEnd$ é denominada função de renomeação tal que

$$F_{Rn}(I_i, End_i, P^{Proxy}_k) = \begin{cases} End_i & \text{Se } P^{Proxy}_k = Input. \\ End_{i+1} & \text{1-Se } P^{Proxy}_k = Output \text{ e } I_i \notin cInstDesvCond \\ & \text{não sendo de desvio dinâmico ou incondicional.} \\ & \text{2-Se } P^{Proxy}_k = Output2 \text{ e } I_i \in cInstDesvCond. \\ P_{in} & \text{Se } P^{Proxy}_k = Output \text{ e } I_i \notin cInstDesvCond \\ & \text{sendo de desvio dinâmico.} \\ F_{Ext}(I_i, End_i) & \text{1-Se } P^{Proxy}_k = Output \text{ e } I_i \notin cInstDesvCond. \\ & \text{sendo de desvio incondicional.} \\ & \text{2- Se } P^{Proxy}_k = Output1 \text{ e } I_i \in cInstDesvCond. \end{cases}$$

Definição 4.14 (Função de Fluxo-F_{Flow}) .
Seja $ApCod_n$ uma aplicação de código, $InstMem^n$ o conjunto de endereços de memória das instruções em $ApCod_n$, $F_{Ext}(I_i, End_i)$ uma função que fornece o endereço de desvio da instância de instrução I_i, que está alocada na posição de memória $End_i \in InstMem^n$.

$$F_{Flow} : ApCod_n \times InstMem^n \rightarrow \left\{ cProxy\_k | \{P_i\}_{cProxy\_k} \subseteq cEnd, \{a_i\}_{cProxy\_k} \subseteq cArc \right\}.^5$$

Onde

$$cArc = \{(InstMem_k^n, InsPName_i), (InsPName_i, InstMem_k^n)\} \cup \{(InsPName_i, P_{in})\}$$

e cEnd é um conjunto de possíveis nomeações dos lugares das redes Proxy associadas a $ApCod_n$ (vide Definição 4.13). Dado I_i e End_i, F_{Flow} define o seguinte mapeamento:

$$F_{Flow}(I_i, End_i) = \begin{cases} ProxyDesv_i | \{P_k\}_{ProxyDesv_i} = ConjP1 & Se\ I_i \in cInstDesvCond \\ \quad \{a_k\}_{ProxyDesv_i} = ConjA1 & \\ \quad \{T_k\}_{ProxyDesv_i} = \{InsPName_i\} & \\ ProxyOrd_i | \{P_k\}_{ProxyDesv_i} = ConjP2 & Se\ I_i \notin cInstDesvCond, \\ \quad \{a_k\}_{ProxyDesv_i} = ConjA2 & não\ sendo\ de\ desvio\ dinâ- \\ \quad \{T_k\}_{ProxyDesv_i} = \{InsPName_i\} & mico. \\ ProxyOrd_i | \{P_k\}_{ProxyDesv_i} = ConjP2 & Se\ I_i \notin cInstDesvCond, \\ \quad \{a_k\}_{ProxyDesv_i} = ConjA2 & sendo\ de\ desvio\ dinâmico. \\ \quad \{T_k\}_{ProxyDesv_i} = \{InsPName_i\} & \end{cases}$$

Onde :

$ConjP1 =$

$$\{F_{Rn}(I_i, End_i, Input), F_{Rn}(I_i, End_i, Output1), F_{Rn}(I_i, End_i, Output2)\}$$

$ConjP2 =$

$$\{F_{Rn}(I_i, End_i, Input), F_{Rn}(I_i, End_i, Output))\}$$

$ConjA1 =$

$$\{(F_{Rn}(I_i, End_i, Input), InsPName_i), (InsPName_i, F_{Rn}(I_i, End_i, Output1), (InsPName_i, F_{Rn}(I_i, End_i, Output2))\}$$

$ConjA2 =$

$$\{(F_{Rn}(I_i, End_i, Input), InsPName_i), (InsPName_i, F_{Rn}(I_i, End_i, Output))\}$$

[5] A notação $\{A_k\}_B$ representa o conjunto de entidades A (lugares, transições ou arcos) pertencentes à rede B.

De posse dos procuradores, conectados pela nomeação de seus lugares e arcos, constroi-se um modelo CPN básico do fluxo de execução ($MCBF$) pela simples fusão de todas as redes contidas em $cProxy = \{cProxy\_k\}$. Um modelo básico de fluxo é definido por $MCBF = (\Sigma, cLE, cTI, cA, N, C, G, E, I)$[6], representando uma rede na qual $cLE = \bigcup_k \{P_i\}_{cProxy\_k}$, $cTI = \bigcup_k \{InsPName_i\}_{cProxy\_k}$ e $cA = \bigcup_k \{a_j\}_{cProxy\_k}$.

No $MCBF$, o conjunto de transições (cTI) representa as instruções e o conjunto de lugares (cLE) representa os endereços de memória onde as instruções estão alocadas. O $MCBF$ descreve o fluxo de execução onde os endereços de desvio são estáticos, sendo representados por arcos ligando transições (instruções) a lugares (endereços). Contudo, para instruções de desvio *dinâmico*, tais como JMP @A+DPTR e RETI, presentes no i8051, não é possível estabelecer um lugar de destino no modelo $MCBF$. Dada a nomeação realizada pela função F_{Flow}, sempre que uma instrução de desvio dinâmico for executada o *token* é encaminhado para o lugar P_{in}, que no $MCBF$ estará isolado. Para lidar com tal questão é definida uma estrutura complementar ao $MCBF$. Essa processa o destino do *token* enviado para P_{in}. Isso é feito em função do valor do registrador PC, presente no CI encapsulado pelo *token*. Essa estrutura é denominada *Distribuidor de Token*. O processamento do *token* no *Distribuidor de Token* permite, além dos desvios dinâmicos, a avaliação do critério de parada da simulação, tal avaliação é realizada por um segmento de código $f_{av} \in cFI$.

Definição 4.15 (Distribuidor de Token-DT) .
Seja a aplicação de código $Apcod_n$, e $MCBF_n$ seu modelo CPN básico de fluxo, no qual E_{max} é o endereço de sua última instrução e CI é o contexto interno do modelo de arquitetura. A rede de Petri colorida $DT_n = (\Sigma, dP, dT, dA, dN, C, G, E, I)$ é definida como distribuidor de token de $ApCod_n$, onde:
(i) $\Sigma = \{Context\}$,
(ii) $dP = \{P_{in}, P_{out}\} \cup \left\{cLE_k | cLE_k \in \{P_j\}_{MCBF_n}\right\}$,
(iii) $dT = \{t\_DT\} \cup \{Trans_k | 0 \leq k \leq E_{max}\}$,
(iv) $FCod(t\_DT) = f_{av}(CI)$, $FCod(dT_k)_{dT_k \neq t\_DT} =$ *não definido,*
(v) $dA = \left\{a_j^d | 0 \leq j \leq 2 * (E_{max} + 1)\right\}$,
(vi) $dN(a_0^d) = (P_{in}, t\_DT)$, $dN(a_1^d) = (t\_DT, P_{out})$, $dN(a_k^d)_{2 \leq k \leq E_{max}} = (P_{out}, Trans_k)$ *e* $dN(a_k^d)_{E_{max} < k \leq 2 * E_{max}} = \left(Trans_{(k-E_{max})}, cLE_{(k-E_{max})}\right)$,
(vii) $\forall dP_j \in dP, C(dP_j) = Context$,

[6]Os elementos Σ, N, C, G, E e I serão apresentados na Definição 4.17.

(viii) $G(d\_DT) = true$ *e* $G(Trans_k) = CheckPC(k, CI)|CheckPC() \in cFI$ [7]
(ix)$E(a_k^d) = i|Type(i) = Context,$
(x) $I(dP_k) =$ *não definido.*

A Figura 4.8 apresenta o distribuidor de *token*. Com o distribuidor de *token*, desvios dinâmicos, inclusive as interrupções de *hardware*, são modeladas adequadamente. Como a interrupção é uma ação que ocorre *fora* do código, ela é modelada como um *rapto* do *token* (Contexto Interno) e, subseqüentemente, seu encaminhamento para o endereço alvo - vetor de interrupção. Para realizar tal tarefa é definida a estrutura *Raptora de Token*. A Figura 4.9 apresenta a estrutura *Raptora de Token*.

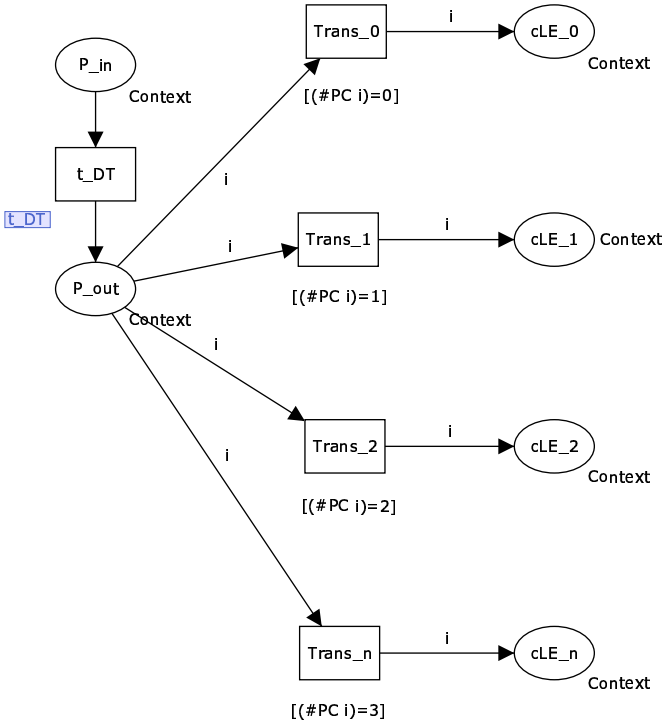

Figura 4.8: Estrutura *Distribuidor de Token*.

[7]Função interna que habilita a transição de transferência ($Trans_k$) em função do valor do PC.

$$CheckPC(k, CI) = \begin{cases} true & \text{Se } PC = k \\ false & \text{Se } PC \neq k \end{cases}$$

Definição 4.16 (Raptora de Token-RT) .

Seja $ApCod_n$ uma aplicação de código, $MCBF_n$ seu modelo CPN básico de fluxo, $IntVect$ um conjunto de endereços dos vetores de interrupção, $AlgInt()|AlgInt() \in cRH$ um segmento de código que transforma CI em função de $IntVect$, Int um flag de interrupção de hardware e $RT_n = (\Sigma, rP, rT, rA, rN, C, G, E, I)$ uma rede de Petri colorida. RT_n é definida como raptora de token de $ApCod_n$, onde:

(i) $\Sigma = \{Context\}$,

(ii) $rP = \{P_{in}\} \cup \left\{cLE_k | cLE_k \in \{P_j\}_{MCBF_n}\right\}$,

(iii) $rT = \{RT\}$,

(iv) $FCod(RT) = AlgInt(CI)$,

(v) $rA = \{a_k^r | 0 \leq k \leq E_{max} + 1\}$,

(vi) $rN(a_0^r) = (RT, P_{in})$ *e* $rN(a_k^r)_{0<k\leq E_{max}+1} = \left(cLE_{(k-1)}, RT\right)$,

(vii) $\forall rP_j \in rP, C(rP_j) = Context$,

(viii) $G(RT) = Int$,

(ix)$E(a_k^r) = i | Type(i) = Context$ *e*

(x) $I(rP_k) =$ *não definido.*

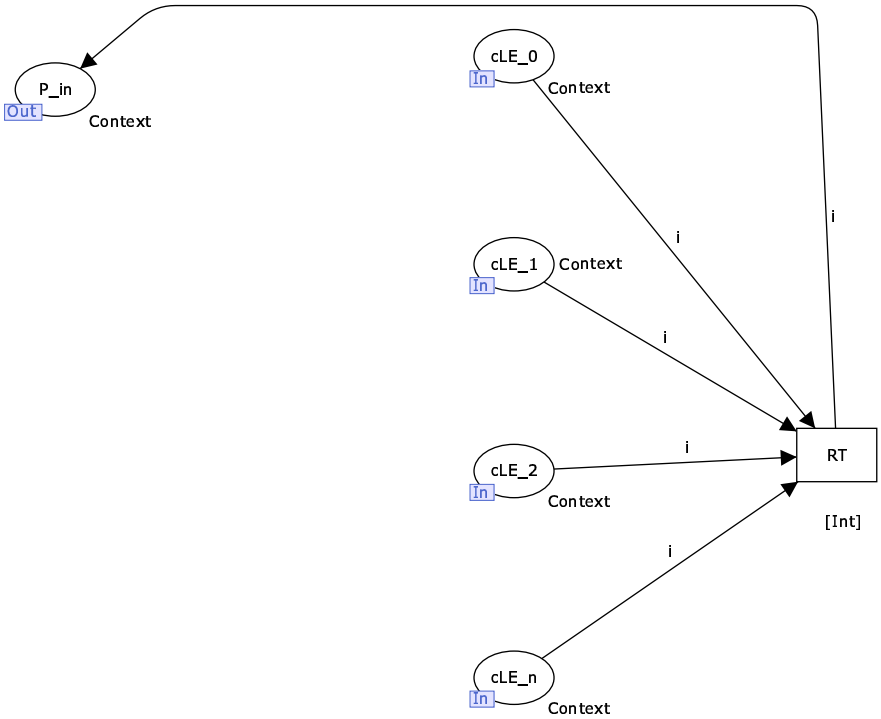

Figura 4.9: Estrutura *Raptora de Token*.

Dada uma aplicação $ApCod_n$, a fusão das redes $MCBF_n$, DT_n e RT_n define um modelo CPN estrutural (MCE). A agregação do cRH e do cFI ao MCE possibilita a definição de um modelo CPN de aplicação.

Definição 4.17 (Modelo CPN de Aplicação-MCA) *:*

Seja $ApCod_n$ *uma aplicação de código,* MCE_n*,* DT_n *e* TR_n *seu modelo CPN estrutural, seu distribuidor de token e seu raptor de token, respectivamente. Seja ainda* $cMCI_n^{Ap}$

$|cMCI_n^{Ap} \subseteq cMCI$ *o conjunto de* MCI*'s das instruções presentes em* $ApCod_n$ *e* $cProxy$ *seu conjunto de procuradores. O modelo CPN de aplicação é definido pela tupla* $MCA_n = (rMCA_n, cRH, cFI)$*. Onde* cRH *é o conjunto de recursos de hardware,* cFI *o conjunto de funções internas e* $rMCA_n$ *é a rede de Petri colorida hierárquica* $rMCA_n = (S, SN, SA, PN, PA, FS, FT, PP)$*, tal que:*

(i) $S = \{MCE_n\} \cup cMCI_n^{Ap}$*, onde*

$MCE_n = (\Sigma, cL, cT, cA, N, C, G, E, I)$*, tal que*

1-$\Sigma = \{Context\}$,

2-$cL = \{dP_j\}_{DT} \cup \{rP_j\}_{RT} \cup \left(\bigcup_k \{cLE_j\}_{cProxy\_k}\right)$,

3-$cT = \{dT_j\}_{DT} \cup \{rT_j\}_{RT} \cup \left(\bigcup_k \{cTI_j\}_{cProxy\_k}\right)$,

4-$cA = \left\{a_j^d\right\}_{DT} \cup \left\{a_j^r\right\}_{RT} \cup \left(\bigcup_k \{a_j\}_{cProxy\_k}\right)$,

5- a)$N(a_0^r) = (RT, P_{in})$ *e* $N(a_k^r)_{0<k\leq E_{max}+1} = \left(cLE_{(k-1)}, RT\right)$,

b)$N(a_0^d) = (P_{in}, t\_DT)$, $N(a_1^d) = (t\_DT, P_{out})$, $N(a_k^d)_{2\leq k\leq E_{max}} = (P_{out}, Trans_k)$ *e* $N(a_k^d)_{E_{max}<k\leq 2*E_{max}} = \left(Trans_{(k-E_{max})}, cLE_{(k-E_{max})}\right)$,

c)$N(a_k) = a_k$[8],

6-$\forall cL_j \in cL, C(cL_j) = Context$,

7— a) $G(cT_j)_{cT_j=d\_DT} = true$ *e* $G(cT_j)_{cT_j=Trans_k} = CheckPC(CI)$

b)$G(cT_j)_{cT_j=RT} = Int$,

c)$G(cT_j)_{cT_j\notin(\{T_v\}_{DT}\cup\{T_v\}_{RT})} = not(Int)$,

8-$E(a_j) = i|Type(i) = Context$

9-$I(cL_j)_{cL_j\neq cL_{Reset}} =$ *não definido,* $I(cL_j)_{cL_j=cL_{reset}} = Value|Type(Value) = Context$*, onde* cL_{reset} *é o lugar que representa o endereço de reset da arquitetura modelada e* $Value \equiv$ *valor do contexto interno.*

(ii) $SN = \left\{cT_k|cT_k \in \bigcup_k \{cTI_j\}_{cProxy\_k}\right\}$,

(iii) $SA(cT_j)_{cT_j\in SN} = MCI_j|MCI_j \in cMCI_n^{Ap}$,

(iv) $PN = \{cL_j|cL_j \notin \{P_{in}, P_{out}\}_{DT}\}$,

(v) $PA(cT_j) = \left\{\left(X(cT_j)_k, PN_k^{MCI_j\in cMCI_n^{Ap}}\right)\right\}$*, onde* $PN^{MCI_j\in cMCI_n^{Ap}}$ *representa o conjunto de portas presentes em um* $MCI_j \in cMCI_n^{Ap}$*. O índice* k *denota um elemento tanto de* $X(cT_j)$ *quanto de* $PN^{MCI_j\in cMCI_n^{Ap}}$*.*

[8]Observe na Definição 4.14, que os arcos de um Proxy são nomeados como tuplas, identificando seus nós de ligação, ou seja, $N(a_k) = a_k$

(vi) $FS \supseteq \left\{ \left\{ X(cT_j)_k, PN_k^{MCI_j \in cMCI_n^{Ap}} \right\}_h \right\}^9$,
(vii) $FT(FS) = global$,
(viii) $PP = 1'MCE_n$

O modelo CPN de aplicação estabelece um modelo formal de simulação e análise do código de interesse. Uma vez descrito o $cMCI$ de uma arquitetura, o modelo CPN simulável é automaticamente construído pelo instanciamento do MCA do código alvo. Dessa forma, as redes de Petri coloridas são aplicadas como uma linguagem de descrição de arquitetura. Os conjuntos $cMCI$, cRH e cFI, estabelecem um mecanismo adequado de descrição da arquitetura alvo, definindo um modelo CPN de descrição.

Definição 4.18 (Modelo CPN de Descrição-MCD) .
Seja $cMCI$ o conjunto de modelos CPN de instrução de uma arquitetura, cRH o conjunto de seus recursos de hardware e cFI um conjunto de funções internas. O modelo CPN de descrição dessa arquitetura é definido pela tupla $MCD = (cMCI, cRH, cFI)$.

Adicionalmente, o modelo incorpora um conjunto de variáveis que operam como sondas de observação dentro do modelo da arquitetura alvo. Essas variáveis são denominadas *Sondas de Análise*, tendo como função registrar parâmetros de interesse durante a simulação. Os parâmetros de interesse variam de acordo com a finalidade específica do modelo. As *Sondas de Análise* foram construídas em função do consumo de energia, tempo de execução e número de acessos à memória de dados e de código.

Definição 4.19 (Sondas de Análise-SA) .
Define-se o conjunto de sondas de análise como $cSA = \{cSA_k | cSA_k \in cRH\}$, no qual cSA é um conjunto de variáveis presentes em um modelo de arquitetura. Os elementos de cSA recebem atribuições por meio de um conjunto de funções $cSAtrib \in (cMCI \cup cRH)$.

A Figura 4.10 apresenta a estrutura de um modelo de aplicação para um código e arquitetura hipotéticos. O modelo CPN tem seu mecanismo de simulação centrado no fluxo de código descrito pelo modelo de aplicação. O contexto interno é encapsulado no *token* da rede, caracterizando uma entidade que percorre a rede. Em última análise, o *token*, em deslocamento pela rede, representa o processador explorando os estados internos gerados pela execução do código, em que a estrutura da rede descreve os possíveis fluxos de execução. O mecanismo hierárquico das redes de Petri coloridas é aplicado de forma que cada transição do modelo de aplicação encapsula um modelo de

[9] O conjunto fusões específico depende do código alvo.

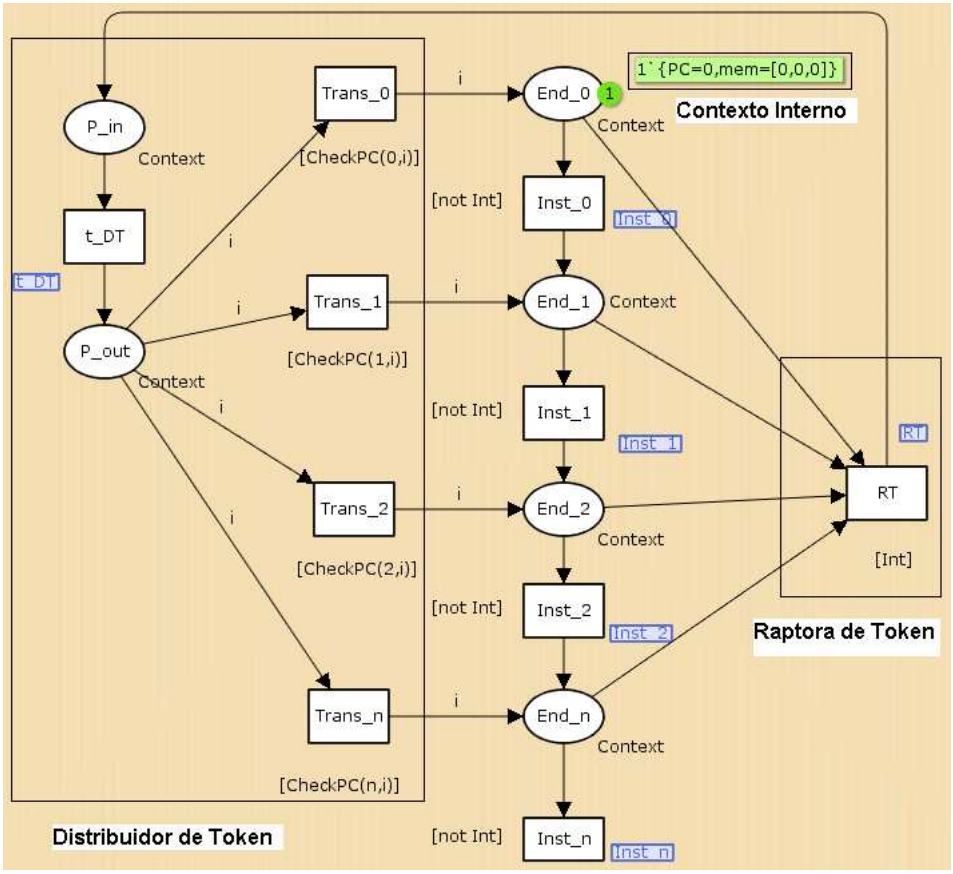

Figura 4.10: Estrutura do Modelo CPN de Aplicação

instrução. Por sua vez, cada transição do modelo de instrução é capaz de encapsular um modelo de operação. Esse último pode encapsular modelos de operação de menor nível de abstração, recursivamente até atingir a abstração mínima de operação desejada. Para um modelo CPN descrito apenas em nível comportamental, as transições do modelo de instrução executam códigos que evocam os recursos de *hardware* presentes no *cRH*. Os modelos CPN são plena e independentemente executáveis por meio de um simulador CPN. Adicionalmente, um mesmo modelo pode ser descrito com vários níveis de abstração, possibilitando a implementação de modelos otimizados para análises específicas.

4.2.1 Taxonomia de Análise

As aplicações de código em sistemas embutidos envolvem normalmente dois tipos de funcionalidades: controle e processamento de sinais. Em uma definição simples, as aplicações de controle monitoram o ambiente e reagem às mudanças que nele ocorrem. Sistemas de controle podem ser caracterizadas por operarem em um laço contínuo de varredura ou em estado de espera de uma interrupção de *hardware*, o que estabelece

tempo de execução infinito. O modelo de análise proposto neste trabalho assume a avaliação dessas aplicações como rotinas de tempo de execução finito. As aplicações de controle são avaliadas com o seu laço principal aberto, avaliando-se o comportamento do código em uma varredura. As aplicações de processamento de sinais são avaliadas como simples rotinas, evocadas e analisadas após seu término. Para a melhor caracterização das regiões de maior consumo do código faz-se necessária a criação de uma terminologia, associando trechos de código a padrões específicos, permitindo assim uma descrição formal.

Definição 4.20 (Vetor de Execução (*Execution Vector*)) .
É um vetor onde cada componente está associado, ordenadamente em função da posição de memória, à uma instância de instrução da $ApCod_n$. O valor do componente representa o número de vezes que a instrução foi executada. Dado um conjunto de instâncias de instruções da aplicação ($ApCod_n$), $EV_n : N^{|InstMem^n|}$ define o Vetor de Execução de $ApCod_n$, onde $EV_n[I_i]$ representa o número de execuções da instância de instrução I_i.

Definição 4.21 (Vetor de Consumo (*Consumption Vector*)) .
É um vetor onde cada componente está associado, ordenadamente em função da posição de memória, à uma instância de instrução da $ApCod_n$. O valor do componente representa o consumo de energia da instrução. $CV_n : R^{+|InstMem^n|}$ define o Vetor de Consumo de $ApCod_n$, onde $CV_n[I_k] = B_k + O_{k,k-1}$ representa o consumo de energia associado à instância de instrução I_k, para $\alpha < k \leq \beta$ e $k \in N$, onde $[\alpha, \beta]$ é um intervalo de índices de endereços da memória de código. B_k e $O_{k,k-1}$ são, respectivamente, o custo básico e o custo inter-instrução associado à instância de instrução I_k. $O_{k,k-1}$ representa o custo inter-instrução associado ao par de instâncias de instruções de índices $k-1$ e k.

O termo *Perfil de Execução* define a representação gráfica do *Vetor de Execução*. A Figura 4.11 mostra o *Perfil de Execução* de um programa de ordenamento em bolha (*BubbleSort*), onde as instruções estão identificadas por *números* referentes ao índice k. A análise desse perfil permite a identificação de certos padrões tais como *Patches*.

Definição 4.22 (Patch) .
Patch é um conjunto de instâncias de instruções que estão localizadas em endereços consecutivos e são executadas um mesmo número de vezes. Seja uma $ApCod_n$ avaliada, um Patch é definito pelo conjunto $Ptch = \{I_i | \alpha \leq i < \beta, i \in N, EV[I_i] = EV[I_{i+1}] \neq 0\}$ $|Ptch \subseteq ApCod_n$, onde $[\alpha, \beta]$ é um intervalo de índices de endereços da memória de código.

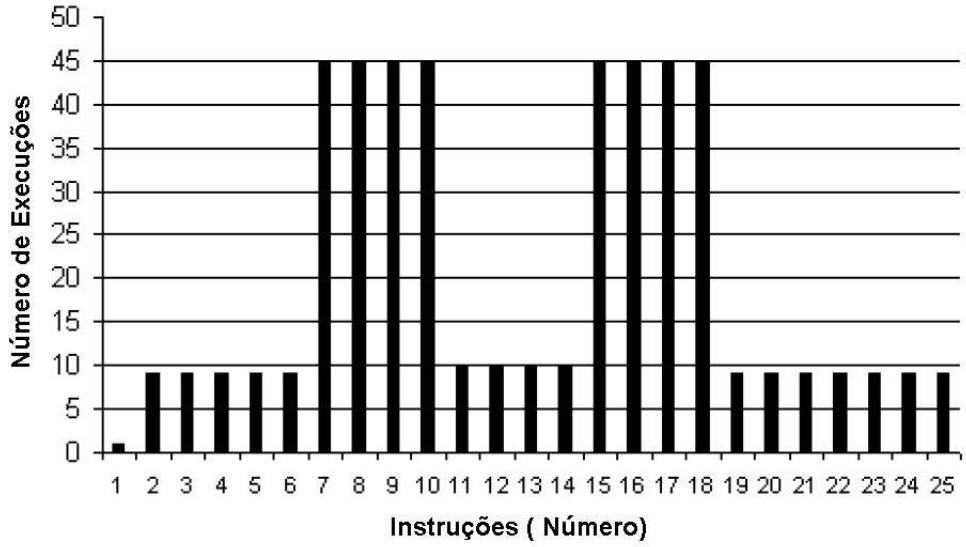

Figura 4.11: Exemplo de *Perfil de Execução*

Na Figura 4.11, cinco *Patches* são identificados: de I_2 até I_6 ($Ptch_1$), de I_7 até I_{10} ($Ptch_2$), de I_{11} até I_{14} ($Ptch_3$), de I_{15} até I_{18} ($Ptch_4$) e de I_{19} até I_{25}($Ptch_4$). O *Patch* é uma renomeação da entidade denominada *bloco básico* (*Basic Block*) [29].

Definição 4.23 (Loop-Patch) .
Loop-Patch representa um laço (loop) único (sem laços internos). O Loop-Patch é um Patch que quando seguido e precedido por uma instância de instrução, essas são executadas apenas uma vez. Seja $Ptch_i \subseteq ApCod_n$ *um Patch, sejam* h *e* v *o maior e menor endereço de instrução em* $Ptch_i$*. *$Ptch_i$* é definido como Loop-Patch se e somente se* $EV_n[I_{v-1}] = EV_n[I_{h+1}] = 1$.

O *Loop-Patch* identifica trechos que executam laços estruturalmente independentes, ou seja, laços que não fazem parte de uma estrutura de laços aninhados. Esse conceito é particularmente útil para a identificação de trechos que representam laços isolados, passíveis de migração *software-hardware*.

Definição 4.24 (Cluster) .
Cluster é um conjunto de Patches agregados em endereços consecutivos, que quando seguido e/ou precedido por uma instrução, essas são executadas apenas uma vez. Seja $Cter = \bigcup_{i=1}^{m} \{Ptch_i\}$ *um conjunto de Patches consecutivos,* h *e* v *o maior e menor índice de endereço de instância de instrução associado ao* $Cter$*, *$Cter$* é definido como Cluster, se e somente se,* $EV_n[I_{v-1}] = EV_n[I_{h+1}] = 1$.

O *Cluster* identifica trechos que executam conjuntos de laços, sendo esses indicadores de centros de consumo de energia no código [92]. Na Figura 4.11 existe apenas um *Cluster*: $\bigcup(Ptch_1, Ptch_2, Ptch_3, Ptch_4, Ptch_5)$.

Definição 4.25 (Bound-Patch Set) .
Bound-Patch Set é um conjunto não unitário de Patches executados um mesmo número de vezes e pertencentes ao mesmo Cluster. Seja um Cluster $Cter_j$ *e um conjunto de Patches* $BPS = \{Ptch_i\} | Ptch_i \subset Cter_j$, BPS_m *é definido como Bound-Patch Set de* $Cter_j$, *se e somente se,* $BPS_m = \{Ptch_i | \forall I_k \in Ptch_i, EV_n[I_k] = const\}$.

O *Bound-Patch Set* identifica os *Patches* que podem possuir forte interdependência dentro de um *Cluster*. Na Figura 4.11 existem dois *Bound-Patch Set*: $\{Ptch_1, Ptch_5\}$, $\{Ptch_2, Ptch_4\}$.

Definição 4.26 (Free-Patch) .
Free-Patch é um Patch contido em um Cluster mas não incluso em nenhum Bound-Patch Set desse Cluster. Seja $Ptch_i \subseteq Cter_j$, $Ptch_i$ *é um Free-Patch, se e somente se,* $\nexists BPS$ *de* $Cter_j | Ptch_i \in BPS_j$.

O *Free-Patch* identifica trechos de código (*Patches*) que possuem número de execução possivelmente independente dos demais. Na Figura 4.11 existe apenas um *Free-Patch*: $Ptch_3$.

Avaliando-se o vetor de consumo e execução, pode-se definir métricas específicas tais como:

1. *Consumo de Instância de Instrução*

$$E_n^{Inst}(I_k) = EV_n[I_k] \times CV_n[I_k]$$

Onde $E_n^{Inst}(I_k)$ é o consumo total associado à instância de instrução I_k, $EV_n[I_k]$ é o número de execuções da instância de instrução I_k e $CV_n[I_k]$ é seu custo energético (custo básico + custo inter-instrução).

2. *Consumo de Instrução*

$$E_n^{Instr}(Name) = \sum_{\forall k | I = Name} EV_n[I_k] \times CV_n[I_k]$$

Onde $E_n^{Instr}(Name)$ é o consumo total associado a instrução I, cujo mnemônico é $Name$ (vide Definição 4.17), $EV_n[I_k]$ é o número de execuções da instância de instrução I_k e $CV_n[I_k]$ é seu custo energético (custo básico + custo inter-instrução).

3. *Consumo de Patch*

$$E_n^{Patch}(Ptch_h) = \sum_{\forall k | I_k \in Ptch_h} EV_n\left[I_k\right] \times CV_n\left[I_k\right]$$

Onde $E_n^{Patch}(Ptch_h)$ é o consumo de energia do *Patch* $Ptch_h$, $EV_n\left[I_k\right]$ é o número de execuções do *Patch* $Ptch_h$, e $CV_n\left[I_k\right]$ é o custo energético associado à instância de instrução $I_k \in Ptch_h$.

4. *Consumo de Cluster*

$$E_n^{Cluster}(Cter_h) = \sum_{\forall k | Ptch_k \in Cter_h} E_n^{Patch}(Ptch_k)$$

5. *Vetor Consumo de Aplicação*

$$Cp_n = (Ev_n \bullet Cv_n)$$

Onde Ev_n é o vetor de execução da aplicação $ApCod_n$ e Cv_n é seu vetor de consumo. O termo *Perfil de Consumo* define a representação gráfica do *Vetor Consumo de Aplicação*.

A identificação das entidades propostas, e a estimativa das métricas apresentadas, ajudam o projetista na visualização das estruturas do código e seus consumos de energia. Por exemplo, um *Loop-Patch* representa um laço isolado dentro do código, um *Cluster* representa um trecho de código no qual ocorre concentração de consumo de energia e tempo de execução, um *Cluster* com *Bound-Patch Sets*, cujos *Patches* apresentam simetria no *Perfil de Execução*, pode representar um laço aninhado (vide Figura 4.11). Inspecionando os gráficos de *Perfil de Execução* e *Perfil de Consumo*, o projetista pode mapear consumo de energia nas estruturas do código. De fato, entidades como *Patches*, *Clusters* e *Bound-Patch Sets* estabelecem indícios do fluxo característico do programa. Com base nesses indícios, o projetista pode verificar a estrutura do trecho em questão em uma representação gráfica do fluxo de programa, ou seja, avaliar o $MCBF$ contido no modelo CPN de aplicação. Dessa forma, identificando estruturas de código que possam ser otimizadas ou migradas para *hardware*.

4.3 Modelo CPN-i8051

Nessa seção será demonstrada a aplicação do modelo de descrição apresentado, através da modelagem da arquitetura i8051. Para tal, o ambiente CPNTools foi utilizado

para a construção e validação do conjunto $cMCI$, bem como validação e execução dos $MCAs$ de teste, estabelecendo assim um modelo CPN de descrição do i8051 - $MCD-i8051$. Para possibilitar a implementação de $MCAs$ automaticamente, a partir do código executável, um compilador Binário-CPN foi desenvolvido, esse compilador será descrito em seções posteriores. O $MCD - i8051$ foi especificado completamente por descrições comportamentais das instruções.

Na atual implementação foram estabelecidas as seguintes características do modelo: (i) as operações são modeladas de forma comportamental, por meio de RHs, (ii) as instruções são descritas por códigos que acessam os RHs (operações), (iii) as interrupções de *hardware* não são modeladas e (iv) existe um conjunto de atribuições em SAs, sendo executadas no modelo de instrução ou no conjunto de recurso de *hardware*.

4.3.1 Definindo o Contexto Interno

O primeiro passo na descrição de uma arquitetura consiste em definir o contexto interno (CI). No ambiente CPNTools, isso é feito pela criação do tipo *Context*, que estabelece a estrutura de dados do CI, definindo assim o *tipo* (*color set*) de todos os lugares e variáveis presentes no MCA. Como *Context* é um *tipo* composto, faz-se necessário definir os *tipos* secundários que caracterizam os elementos de armazenamento da arquitetura. O *Context* e os seus componentes são definidos na área de declarações do ambiente. O *tipo Context* da arquitetura i8051 é definido como:

```
colset Context =
                record cycle:inte*power:instpad*Ex_Pro:instpad*
                patch:instpad*mem:Memory*Upmem:Memory*
                PC:inte*ext_mem:Memory*program:instpad;.
```

No qual `cycle`, `power`, `Ex_Pro` e `patch` são SAs. O `cycle` acumula o número de ciclos de relógio executados, operando como um marcador de temporização interna do modelo. A sonda `power` registra de forma acumulativa a energia consumida durante a execução, disponibilizando sub-totais de consumo a qualquer momento da simulação. O `Ex_Pro` é uma lista carregada durante a simulação de forma a construir o vetor de execução da aplicação. Identicamente, o `patch` constrói o vetor de consumo da aplicação. Os elementos `mem`, `Upmem`, `ext_mem` e `program` são listas que modelam respectivamente a memória RAM interna, a porção alta da RAM interna, a memória RAM externa e a memória de código. Tais elementos representam o real contexto interno, descrevendo a arquitetura em termos de elementos de armazenamento. O instanciamento do CI no modelo é feito pela definição da variável valor de contexto (*value*)

que é inserida no *token*. No ambiente CPNtools *value* é sintaticamente uma constante, contudo essa constante é modificada a cada disparo de transições Proxy em função da instrução executada, o que faz com que ela tenha comportamento de variável. O construtor de *value* é definido como sendo:

```
val value={cycle=0,power=[0,0,0,0],Ex_Pro=inEx_Pro,patch=Inst_Preg,
           mem=write_pr(inic,129,7),Upmem=Up_inic,PC=0,
           ext_mem=ext_m_inic,program=programinic};.
```

Como pode ser visto, para cada elemento do CI há um construtor específico definido como uma função CPN-ML na área de declarações do ambiente CPNTools. Por exemplo, o registrador de pilha (SP), localizado na posição 129 da memória RAM interna é inicializado pela função `write_pr(inic,129,7)`, ($write\_pr() \in cFI$).

4.3.2 Construindo $MCIs$

O processo de construção dos modelos CPN de instrução consiste na criação de redes com dois padrões: (i) redes de um lugar de entrada e dois de saída, e (ii) redes de um lugar de entrada e um lugar de saída. As redes do tipo (i) definem os $MCIs$ das instruções de desvio condicional, no qual o *token* possui duas alternativas de fluxo de execução. As redes do tipo (ii) definem todas as demais instruções - $MCIs$ ordinários. O processo de concepção do MCI implica na natural criação de um MCA para instrução. Entenda-se esse MCA como um modelo CPN de aplicação onde o código de aplicação possui apenas uma instrução, sendo caracterizado por $rMCA = MCI$, ou seja, uma rede não hierárquica definindo o modelo CPN da instrução alvo. Um ponto particularmente importante é que o MCA de uma instrução é plenamente executável no ambiente CPNTools. O RH é definido na área de declaração do ambiente, de acordo com a descrição da instrução. Dessa forma, os modelos CPN de instrução são construídos encapsulados como modelos de aplicação, permitindo validação e verificação do modelo de instrução durante o processo de descrição. A medida que o $cMCI$ vai sendo criado, as instruções vão acumulando recursos de *hardware* nas áreas de declaração de seus respectivos $MCAs$, construindo assim o cRH do modelo CPN de descrição. Após a concepção de todo o $cMCI$, as áreas de declaração de todos os $MCAs$ de instrução são atualizadas com o cRH mais recente. Isto é feito pela manipulação dos arquivos do CPNTools. Por armazenar os modelo no padrão XML o CPNTools possibilita que seus arquivos sejam facilmente manipuláveis. Durante a implementação do $MCD - i8051$ foram criadas aplicações em Java para a execução de tais manipulações. De forma similar cria-se um arquivo separado contendo o cRH no

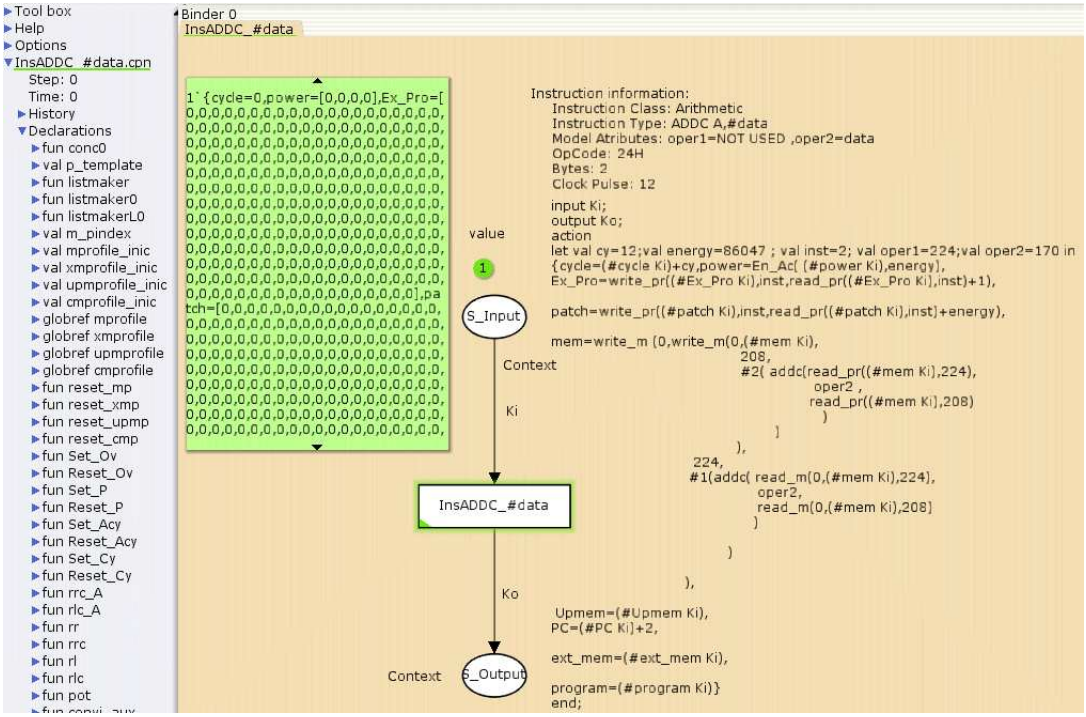

Figura 4.12: Modelo CPN da instrução ADDC A,#dado

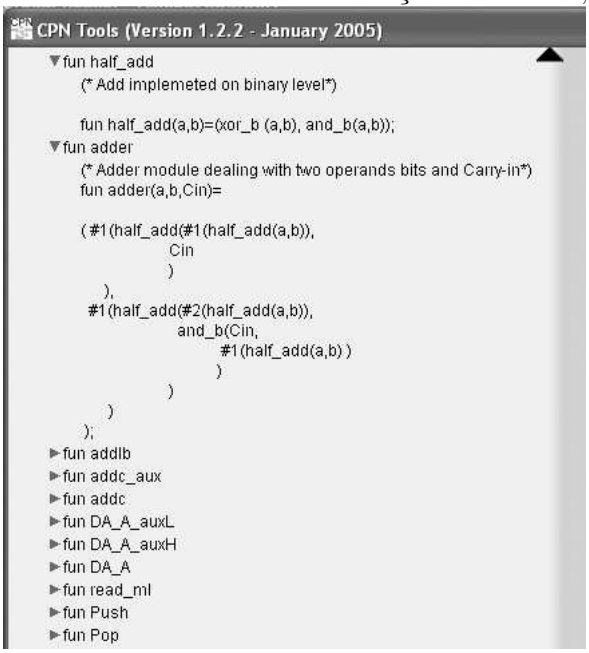

Figura 4.13: Recurso de *hardware* referente a ULA módulo somador

formato da área de declaração do CPNTools. Esse arquivo é denominado `Globox.cpn` e compreende a primeira parte do arquivo XML que descreve um MCA. Sendo assim, a tupla $MCD = (cMCI, cRH, cFI)$ tem seu primeiro elemento representado pelo conjunto de arquivos dos modelos CPN de instrução criados (encapsulados como $MCAs$) e o segundo e terceiro elemento pelo arquivo `Globox.cpn`.

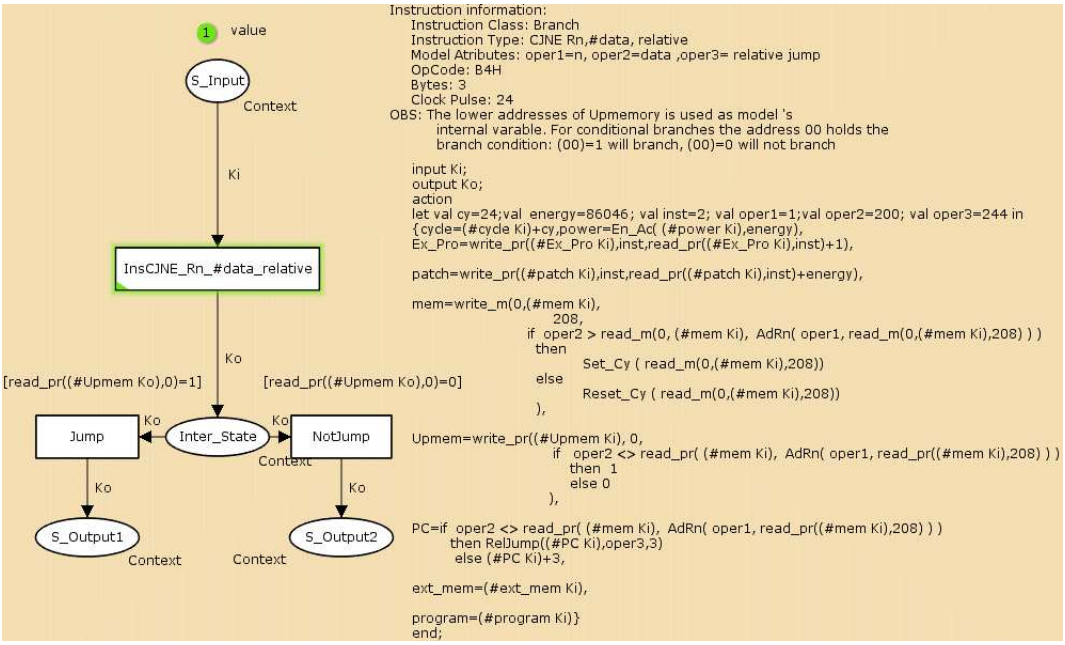

Figura 4.14: Modelo de uma instrução de desvio condicional

Devido ao nível de abstração da descrição $MCD - i8051$, cada MCI possui apenas uma transição capaz de modificar o CI, essa transição opera por meio de um $CodInst$ associado. Para que o $CodInst$ acesse a estrutura do CI ($token$), variáveis de entrada (K_i) e saída (K_o) são avaliadas como expressões nos arcos de entrada e saída respectivamente. O $CodInst$ é descrito em CPN-ML de forma a modificar os elementos do CI afetados pela instrução. A Figura 4.12 mostra o MCI da instrução ADDC A,#dado, no qual evidencia-se tais elementos de descrição. Os recursos de *hardware* são descritos por funções CPN-ML (vide lado esquerdo da Figura 4.12) que descrevem as operação em diferentes níveis de abstração. Por exemplo, `write_m()` e `read_m()` promovem o acesso aleatório a qualquer posição do modelo de memória, emulando assim operações de escrita e leitura. Como um exemplo de descrição de baixo nível, a Figura 4.13 mostra a descrição de uma unidade somadora com operação bit a bit. Em uma análise mais detida do modelo, observa-se ainda que o CI é transportado como um *valor* e não como uma *referência*, isso foi feito em favor da transparência referencial nos $CodInst$. Embora seja possível criar artifícios no CPNTools de forma a descrever o CI usando referências, nesse primeiro modelo $MCD - i8051$ optou-se pelo formalismo, em detrimento do desempenho. Vale salientar que tal escolha é um dilema comum quando da construção da semântica interna em linguagens de descrição de arquitetura [83].

Nas instruções de desvio condicional, a rede modela o fluxo de execução, de forma que faz-se necessário o uso de expressões de guarda em algumas transições do MCI. Essas expressões avaliam uma função predicado, o resultado dessa função determina

o fluxo de execução, mediante o disparo da transição `Jump` ou `NotJump`, presentes nos $MCIs$ de instruções de desvio condicional (vide Definição 4.8). Na Figura 4.14 é apresentado o MCI da instrução CJNE Rn,#dado, no qual observa-se que o fluxo de execução depende do resultado da avaliação das expressões `read_pr((#Upmem Ko),0)=0` e `read_pr((#Upmem Ko),0)=1` - substituindo, nessa implementação, chamadas à função $CheckJump() \in cRH$ - Definição 4.8-, em que `read_pr((#Upmem Ko),0)` retorna o valor armazenado na posição 0 da memória alta. O processo de modelagem da memória RAM interna alta (`Upmem`) faz com que exista um total de 128 posições não utilizadas pelas instruções, pois elas não existem no dispositivo real. O modelo $MCD - i8051$ faz uso desses endereços para guardar sinalizadores (*flags*) internos que servirão em tempo de simulação. A posição 0 é um desses sinalizadores, sinalizando se haverá ou não desvio. A atribuição desse sinalizador pode ser observada na linha de atribuição da `Upmem` (vide Figura 4.14).

O processo de acesso aos bancos de registradores é executado pela função/recurso de *hardware* `AdRn(n,psw)` ($AdRn(n, psw) \in cRH$). `AdRn(n,psw)` recebe o número do registrador e o *psw*, retornando o endereço do registrador. Pelo uso de funções como essas, os $CodInsts$ reproduzem fielmente o comportamento da instrução modelada, a simulação da instrução é exata em seus efeitos nos registradores e *flags* internos do processador.

4.3.3 Modelo de Consumo de Energia

O $MCD - i8051$ foi construído de forma que todos os $CodInsts$ possuem campos de constantes (`energy` e `cy`)[10], nos quais são atribuídos os valores característicos de consumo de energia e tempo de execução da instrução. A mudança de tais parâmetros pode ser feita com facilidade, acessando-se os arquivos XML do $cMCI$, uma pequena aplicação Java permite que isso seja feito automaticamente. Dessa forma, uma vez de posse do modelo de consumo por instrução, o $MCD - i8051$ pode ser ajustado para qualquer dispositivo da arquitetura i8051. Similarmente, todo $CodInst$ está munido com comandos de registro nas SA. Antes da execução da instrução, o impacto de sua execução é registrado nos vetores de execução e de consumo da aplicação.

O modelo de consumo por instrução foi construído por meio de medidas realizadas sobre um dispositivo baseado na arquitetura i8051, o AT89S8252. Para tal, um sistema de caracterização automática foi implementado. Detalhes sobre o modelo de energia resultante, o processo de caracterização e as considerações metrológicas são relatados

[10]Sendo `cy` em ciclos de relógio e `energy` em pJ

no *Apêndice A*.

4.3.4 Sondas de Análise

O *MCD* pode ser definido com inúmeras sondas de análise. Como uma primeira abordagem, foram definidas duas classes de sondas: as da camada de operação e as da camada de instrução. As que estão na camada de operação registram parâmetros referentes a elementos e operações do *hardware*, enquanto as da camada de instrução o fazem para parâmetros referentes à instrução e ao código como um todo. A Figura 4.15 apresenta a construção das sondas na área de declarações do CPNTools.

Figura 4.15: Conjunto de definição e construtores de sondas.

O $MCD - i8051$ implementa as sondas de instrução `cycle, power, Ex_Pro` e `patch`, como elementos do CI, sendo portanto transportadas como valores. Como sondas de operação, foram introduzidas estruturas de dados que são alimentadas com o padrão de atividade de escrita e leitura capturado das operações de escrita e leitura. Essas estruturas são listas de listas, em que a lista interna possui três posições, nas quais são armazenados: a identificação do endereço acessado, o número de operações de escrita, e o número de operações de leitura. Cada lista interna representa uma posição de memória. Cada tipo de memória da arquitetura possui uma sonda. A sondagem da operação é realiza pela inserção de chamadas à uma função marcadora de

sonda, dentro da operação alvo. A Figura 4.16 apresenta esse processo, implementado sobre a operação de escrita em memória. A operação é descrita pela função `write_m()` ($write\_m() \in cRH$) e a sondagem pela função `markw_access()`,($markw\_access() \in cFI$).

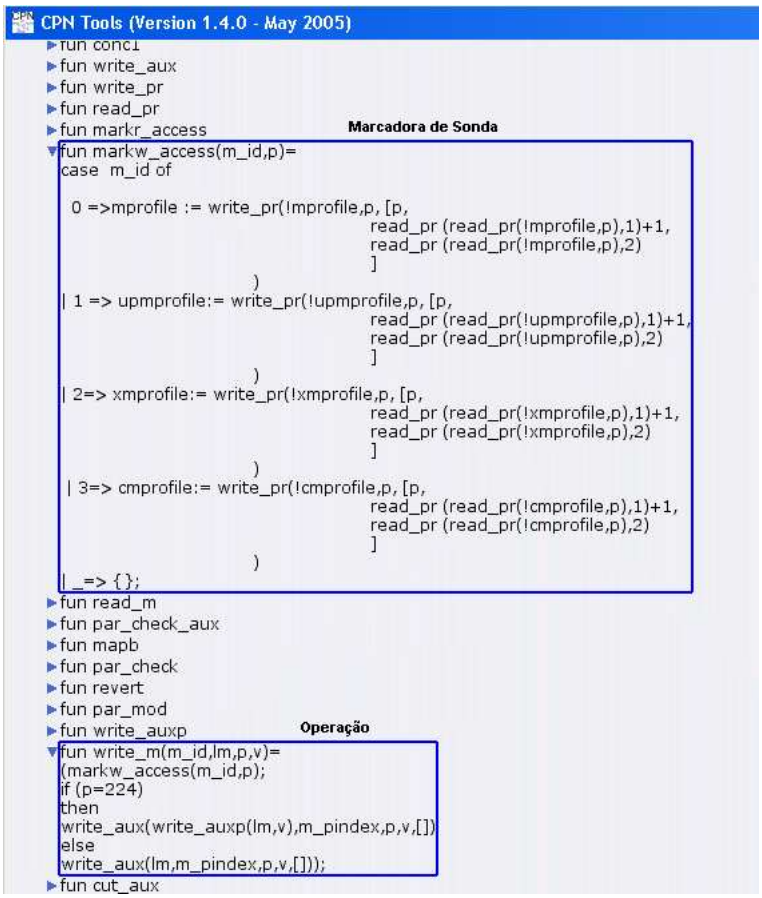

Figura 4.16: Mecanismo de registro de acesso à memória implementado sobre a operação de escrita.

4.3.5 Distribuidor de *Token* e Geração de Arquivos

Como toda instrução modifica o PC, o MCA poderia ser construído baseado apenas na estrutura do distribuidor, dispensando a função de fluxo. Cada MCI enviaria o *token* para o distribuidor que o despacharia para a próxima instrução a ser executada. Isso, embora torne a estrutura da rede mais padronizada, implicaria em duas desvantagens: (i) aumenta o número de disparos da rede por instrução, impactando no desempenho e (ii) faz com que a rede deixe de representar a estrutura de fluxo do programa ($ApCod$) modelado. Devido a isso, o distribuidor só é utilizado por instruções que modificam o fluxo de execução dinamicamente. O mecanismo de distribuição, implementado desta

forma, garante a identificação estrutural de sub-rotinas. Trechos da rede que partem do distribuidor e retornam ao distribuidor caracterizam sub-rotinas. Uma análise do arquivo que descreve o MCA pode informar estaticamente a localização das sub-rotinas, o cruzamento desses dados com o conteúdo do vetor de execução identifica seções de códigos elegíveis para a migração *software-hardware*, com resolução de sub-rotinas (funcionalidades). Cabe ressaltar que como o mecanismo analisaria a descrição de uma rede de Petri colorida, ele seria independente da arquitetura de processador modelada pelo MCD.

No $MCD - i8051$ a estrutura distribuidora de *token* acumula mais duas funções, além da resolução das descontinuidades da função de fluxo (Definição 4.14). Ela avalia o critério de final de execução e gera arquivos de saída com a massa de dados coletada durante a simulação. O critério de final de execução é um artifício usado para detectar o final de simulação de um MCA. Para tal, o código sob análise é instrumentado com uma instrução RET isolada após sua última instrução. Tal procedimento é realizado manualmente para códigos cujas fontes estão em linguagem de montagem (*assembly*), sendo desnecessário para fontes em C, uma vez que o compilador define todo código como uma sub-rotina, inserindo automaticamente um RET ao final do código. A existência de uma instrução RET isolada, sem uma CALL precedente, causa uma exceção durante a simulação do MCA, decorrente de um acesso a uma posição ilegal da pilha de endereços de sub-rotinas. Como já descrito, o distribuidor de *token* avalia o CI sempre que uma instrução de desvio dinâmico for executada. Para detectar dessa exceção, os $CodInsts$ do tipo CALL incrementam um contador de sub-rotina interno ao MCD, contador esse mapeado na posição 2 do modelo de memória alta. Em complementação, os $CodInsts$ do tipo RET decrementam o contador, de forma que a execução de um RET, sem CALL precedente, implica em um valor negativo no contador. Esse mecanismo pode ser observado nas Figuras 4.17 e 4.18. O mesmo código CPN-ML que detecta o final de execução gera arquivos no formato ASCII com os dados armazenados nas sondas de análise. A Figura 4.19 ilustra a estrutura do distribuidor de *token* (a) e o código CPN-ML associado à sua transição principal (T'_{calc})(b)[11]. Dessa forma, ao final de uma simulação do MCA são gerados os seguintes arquivos:

- Cons_Pro.dpb, referente ao vetor consumo de aplicação;
- Ex_Pro.dpb, referente ao vetor de execução;
- IRAM_Profile.dpb, referente ao perfil de acesso a RAM interna;

[11]Nessa implementação a função $f_{av}()$ - Definição 4.15- é implementada como um segmento de código associado à uma sub-página de T'_{calc}.

- XRAM_Profile.dpb, referente ao perfil de acesso a RAM externa;
- UPRAM_Profile.dpb, referente ao perfil de acesso a RAM interna parte alta;
- CodRAM_Profile.dpb, referente ao perfil de acesso a memória de código, especificamente a acessos feitos pela aplicação, não relacionados à busca de instruções.

Esses arquivos constituem os dados brutos da simulação. Para que a aplicação seja analisada sob a óptica das entidades propostas na Seção 4.2.1, faz-se necessário o processamento desses dados. Os algoritmos de processamento definem as *funções de análise*. Na atual estrutura de análise, tais funções são implementadas externamente ao ambiente CPNTools, em um ambiente de integração denominado EZPetri, que será descrito no Capítulo 6.

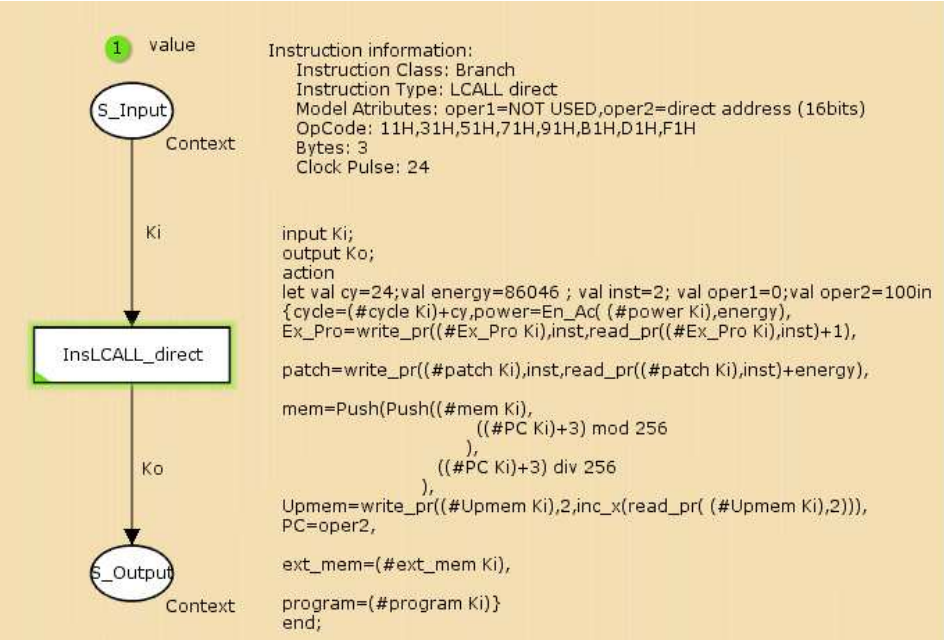

Figura 4.17: Modelo CPN de Instrução da instrução LCALL.

4.3.6 Funções de Análise

Como explicado, as funções de análise reconhecem os padrões referentes às entidades de análise dentro dos arquivos de sonda. A seguir, tais funções são listadas e suas ações básicas são descritas. Adicionalmente, são discutidas algumas considerações sobre possíveis interpretações dos resultados.

- *Get_Eprofile()* retorna o valor da sonda *vetor de execução*, mostrando graficamente o padrão de execução por instância de instrução, ou seja, gerando o *Perfil*

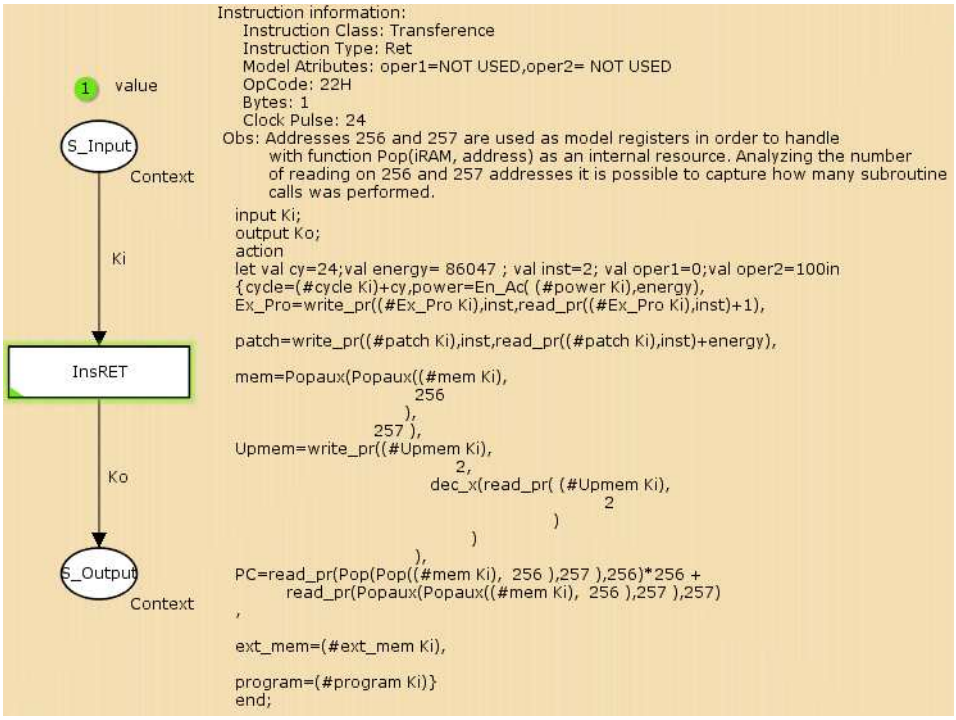

Figura 4.18: Modelo CPN de Instrução da instrução RET.

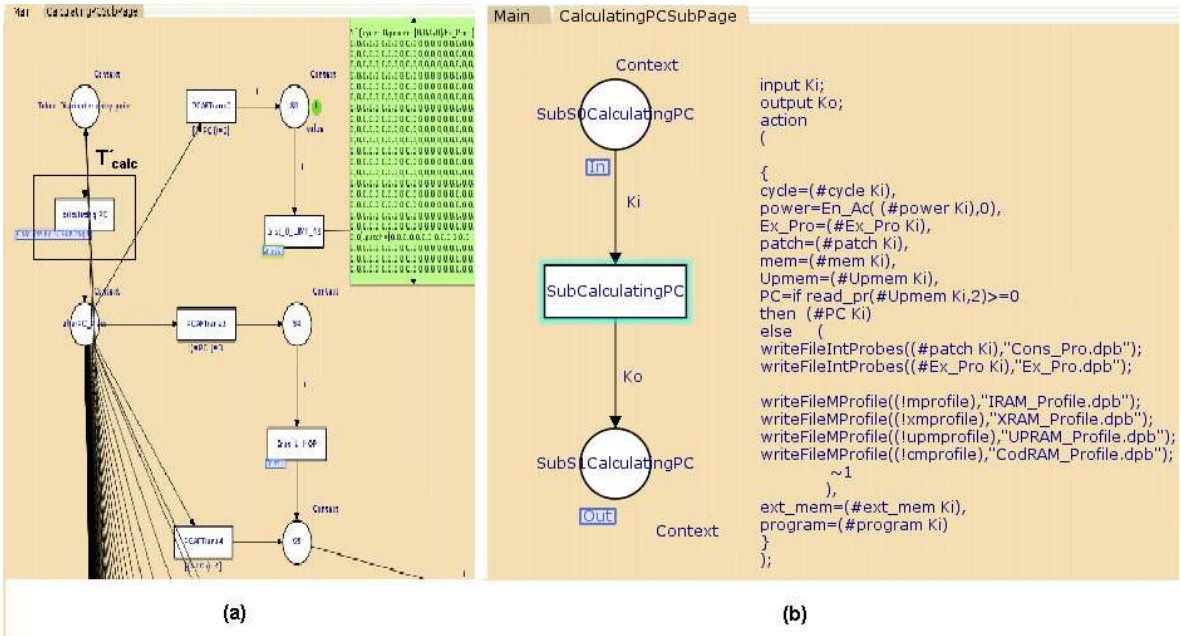

Figura 4.19: (a) Distribuidor de *token* (b) Analisador de final de execução e gerador de arquivos.

de Execução. Esse perfil elucida o padrão de acesso da memória de código e fornece subsídios para o dimensionamento de bancos de memória visando menor consumo. Bancos de memória com tamanhos diferentes podem ser construídos

de forma que se garanta um particionamento eficaz da memória em circuitos alimentados sob demanda [111]. Dessa forma, pode-se estabelecer estratégias de gerenciamento da alimentação da memória, particularmente interessante para aplicações em que o processador/microcontrolador é um *Core*.

- *Get_Dprofile()* retorna a sonda de perfil de acesso a memória de dados, mostrando graficamente o padrão de acesso para cada posição de memória. Identicamente à *Get_Eprofile()*, os dados apresentados por essa função fornecem informações sobre o melhor planejamento de bancos de memória. Uma vez que muitas arquiteturas de microcontroladores mapeiam seus registradores em memória (o 8051 é um exemplo), o perfil de dados permite otimização do consumo com base no acesso aos registradores.

- *Get_Cprofile()* retorna o *Perfil de Consumo*, mostrando graficamente o padrão de consumo por instância de instrução. Esse padrão elucida a distribuição de consumo ao longo do código, podendo assim serem identificados trechos críticos quanto ao consumo.

- *Get_Patches()* retorna os intervalos de trechos da memória de código, nos quais encontram-se *Patches*, e seus números de execuções. A identificação dos *Patches* possibilita a localização de trechos de códigos passíveis de otimização ou de migração para *hardware*.

- *Get_FreePatches()* retorna os intervalos de trechos da memória de código, nos quais encontram-se *Free-Patches*, e seus números de execuções. A identificação dos *Free-Patches* possibilita a localização de laços isolados, passíveis de otimização ou de migração para *hardware*.

- *Get_BoundPatches()* retorna os conjuntos de *Patches* ligados. Os *BoundPatches* fornecem indícios de laços aninhados.

- *Get_CPatches()* retorna o consumo dos *Patches*. Possibilita a identificação dos elementos de consumo por trecho de código.

- *Sort_CPatches(setPatch)* classifica em ordem decrescente de consumo os *Patches* fornecidos na lista de *Patches*, setPatch.

- *Get_CBoundPatches(setBoundPatch)* retorna o consumo dos *Patches* pertencentes ao conjunto de *Patches* ligados, *setBoundPatch*.

- *Get_Clusters()*, retorna os intervalos de trechos da memória de código, nos quais encontram-se *Clusters*. A identificação dos *Clusters* possibilita a identificação de trechos de código que possuem estrutura com vários *Patches* de consumo de energia distintos, ou seja, trechos passíveis de análise para otimização ou de migração para *hardware*.

- *Get_CCluster(Cluster)*, retorna o consumo total de um *Cluster*. Possibilita a identificação dos elementos de consumo por trecho de código associado ao *Cluster*.

A depender do cSA presente no MCD, outras funções podem ser definidas. Funções adicionais devem ser acrescentadas de acordo com as particularidades da arquitetura alvo.

4.4 Compilador Binário-CPN

Com o fim de automatizar a geração de $MCAs$, a partir de qualquer código executável, um compilador Binário-CPN foi desenvolvido para o i8051. Dado um código alvo, o compilador Binário-CPN opera em três etapas: (i) constrói o $cMCI^{Ap}$ da aplicação, (ii) instancia um conjunto de Proxys de acordo com a Definição 4.14 e (iii) instancia um MCA de acordo com a Definição 4.17. Adicionalmente, o compilador foi construído de forma que permitirá o suporte a outras arquiteturas descritas sob o paradigma de modelagem proposto. A Figura 4.20 ilustra a estrutura básica do compilador. O código executável, após ser processado por um *parser*, é representado internamente como um conjunto de objetos. Com base no $cMCI$ e no cRH [12] da arquitetura alvo, um MCA é criado, por meio de um mecanismo gerador de modelos (*ModelFactory*). Este modelo compreende uma rede de Petri colorida no formato interno do compilador. Essa representação interna do MCA é convertida para um arquivo de saída. Esse arquivo pode ser gerado em qualquer formato de representação CPN, a partir da inserção de novas classes conversoras. A Figura 4.21 apresenta o diagrama de classes do compilador. Existem duas classes geradoras de modelos internos: a *Instruction* e a *CPNet*. A classe *Instruction* define o modelo interno das instruções. A definição de cada instrução é feita criando novas classes que herdam suas propriedades básicas da classe *Instruction*. A *CPNet* é a classe geradora das representações internas da rede. A leitura do código binário é realizada pelo *InstructionReader* que gera uma representação do código em termos de objetos do tipo *Instruction*. O *NetInstancer* opera conjuntamente com o *ModelFactory* instanciando os objetos tipo *CPNet*, de forma a criar uma representação

[12] Armazenados como arquivos XML, com denominações Ins_*IDinst*.cpn e Globox.cpn

interna da rede que descreve o *MCA* do código de entrada. Essa representação é convertida para o formato CPNTools gerando uma rede simulável e analisável pelo ambiente CPNTools.

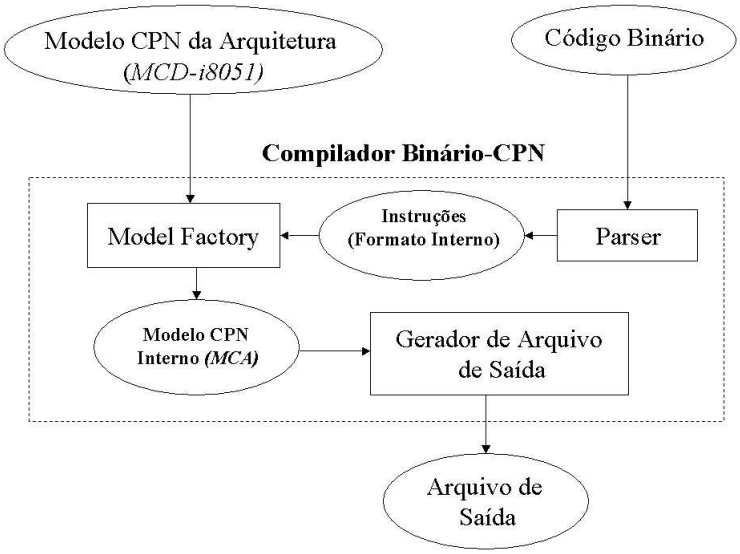

Figura 4.20: Estrutura básica do compilador

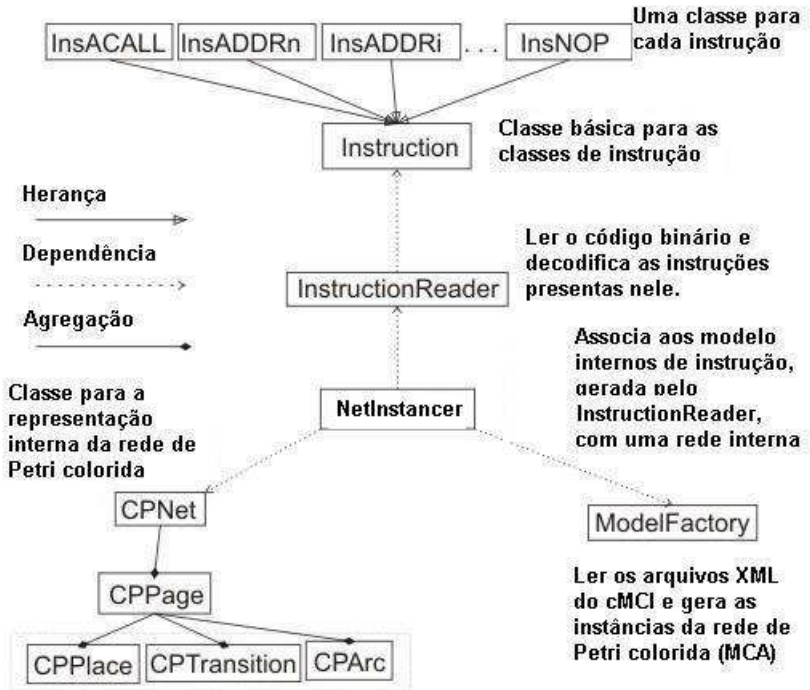

Figura 4.21: Diagrama de classes

4.5 Considerações Finais

Foi apresentada neste capítulo uma metodologia de modelagem e análise de consumo de energia devido ao *software*. O modelo proposto compreende a modelagem do *software* como uma cadeia de fontes de consumo (instruções), cujo acionamento é encadeado no tempo em função dos padrões de fluxo de execução. Esta cadeia é representada por um modelo computacional baseado em *tokens*, expresso em redes de Petri coloridas. Essa metodologia, estabelece a aplicação das redes de Petri coloridas como uma linguagem de descrição de arquitetura (*ADL-Architecture Description Language*), de forma a permitir a reconfiguração do ambiente de análise para diferentes arquiteturas de processadores. Entidades específicas de análise foram propostas, tais entidades mapeiam padrões de execução/consumo em trechos de código, permitindo a identificação de regiões passíveis de otimização ou migração *software-hardware*. A descrição da arquitetura é realizada por meio de um modelo formal denominado modelo CPN de descrição (MCD).

O modelo MCD estabelece uma descrição formal baseado em vinte e seis definições constitutivas. Com base nessas definições, arquiteturas de processadores podem ser descritas em diferentes níveis de abstração, no formato de redes de Petri coloridas. Embora seja centrado na descrição do conjunto de instruções da arquitetura alvo, o MCD permite que a descrição de cada instrução possa ser desdobrada em níveis mais baixos até o modelo de operações de *hardware* associado à instrução, aplicando as propriedades hierárquicas das redes de Petri coloridas. Uma particularidade do MCD é sua natureza "descentralizada", não há descrição específica do *hardware*, mas sim das instruções e suas operações no *hardware*. A análise do consumo de energia devido ao *software* pode ser realizada em diferentes níveis de abstração sem que um modelo da micro-arquitetura do *hardware* seja construído. Sondas de análise podem ser facilmente inseridas dada a regularidade resultante do formalismo das redes de Petri. O modelo de instrução pode ser construído como uma estrutura hierárquica, com cada nível hierárquico descrevendo as operações de *hardware* associadas à instrução. Cada transição do modelo pode encapsular um macro modelo de potência, referente a diferentes resoluções desejadas para a estimativa.

O processo de descrição não é centrado na construção de um simulador de processador, mas sim em um mecanismo formal de simulação do fluxo de execução de uma aplicação, bem como das transformações resultantes nos estados internos do processador (CI). Isso constrói um mecanismo funcionalmente idêntico ao simulador, uma vez que reproduz o exato comportamento do processador, mas cuja regra de construção

é baseada em redes de Petri. Diferentemente de outras abordagens voltadas à descrição de arquiteturas, a descrição e simulação da arquitetura alvo não é baseada em um mecanismo proprietário, concebido de forma *ad hoc*, ou guardando um formalismo especificamente proposto para o qual não existe ferramentas de análise. A aplicação do formalismo de redes de Petri estabelece o pressuposto do emprego dos recursos de análise descritos no Capítulo 3, recursos aceitos e amadurecidos pela comunidade científica. A aplicação de tais recursos no modelo apresentado, constitui por si só, um conjunto de interessantes trabalhos futuros.

Adicionalmente, a metodologia foi validada pela construção da descrição da arquitetura i8051 e criação de um compilador Binário-CPN, objetivando a geração automática de modelos de aplicação. Um conjunto de funções de análise foi integrado em um ambiente de análise protótipo, permitindo avaliações experimentais. Os resultados de tais avaliações são apresentados no Capítulo 6.

Capítulo 5

Modelo CPN-Probabilístico

5.1 Introdução

A estimativa do consumo de energia é fortemente influenciada pelo contexto de entrada da simulação, ou seja, pelo conjunto de cenários de execução gerados pelas variáveis do *software*. Particularmente em sistemas embutidos que operam de forma reativa, o consumo do sistema pode ser caracterizado pela simulação de vários cenários de operação do sistema. Esse conjunto de cenários é, contudo, difícil de delinear em termos de vetores de teste, sendo imensamente numerosos para a grande maioria dos *softwares*. Para abordar essa questão, o modelo CPN determinístico foi estendido de forma a suportar a *simulação probabilística* da aplicação. A extensão probabilística do MCD implementa os preceitos lançados por Burch *et al* em [16], centrando-os na avaliação do consumo devido ao *software*.

Nessa extensão, o modelo possibilita um modo de simulação no qual o *token-game* é gerido por escolhas probabilísticas presentes no disparo das transições de desvio condicional. As transições não mais executam instruções realizando uma simulação comportamental, mas unicamente computam os gastos de energia e tempos de execução. As escolhas de fluxo de execução são determinadas pela avaliação de variáveis aleatórias. Por exemplo, na arquitetura i8051 uma instrução JNC (*Jump if Not Carry*) determina uma escolha entre dois fluxos de execução, f_1 e f_2, com probabilidades, $p(f_1) = 1-p(f_2)$. As probabilidades $p(f_1)$ ou $p(f_2)$ podem ser inferidas *ad hoc* na especificação do sistema, com base em um histórico de execuções do algoritmo descrito pelo código. Funções parametrizáveis, de geração de números aleatórios, definem dinâmicamente a ocorrência do fluxo f_1 ou f_2, com base nos valores de $p(f_1)$ e $p(f_2)$. Dessa forma, o modelo é capaz de reproduzir o comportamento gerado por vetores de teste aleatórios. A cada simulação, as SAs são avaliadas segundo um critério de parada. Ao final de um con-

junto finito de simulações, pode-se postular, baseado no teorema dos grandes números, que a melhor estimativa será a média aritmética das N simulações.

O *teorema dos grandes números*, contudo, estabelece a identidade entre valor médio e melhor estimativa na condição $N \rightarrow \infty$. Para que se realize uma estimativa consistente com um erro máximo pré-fixado, dentro de um valor finito de N, faz-se necessário aplicar um critério de parada adequado. Esse critério de parada é determinado pelo processo *Monte Carlo* [111] de inferência da adequação estatística dos resultados. Uma descrição sucinta do processo *Monte Carlo* é apresentada no Apêndice B. Nesta primeira abordagem é assumido que a função de distribuição de potência é uma *gaussiana*, por não haver um estudo que relacione funções de distribuição ao fenômeno de consumo devido ao *software*. Reforça essa escolha o fato da distribuição *gaussiana* ser intensamente aplicada em estimavas estatísticas de consumo de energia [111]. As seções seguintes descrevem a metodologia e o arcabouço de modelamento e análise propostos.

5.2 Modelo CPN-Probabilístico de Descrição

O modelo CPN-probabilístico de descrição amplia o MCD adicionando um segundo modo de simulação do modelo CPN da aplicação. Esse segundo modo é implementado pela definição de uma nova entidade no modelo, denominada *código probabilístico de instrução* ($CodProbInst$) que assume o lugar do $CodInst$.

Definição 5.1 (Código Probabilístico de Instrução-CodProbInst) .

Descrição da instrução em função da probabilidade da mesma gerar mudança no fluxo de execução da aplicação.

$cCodProbInst$ representa o conjunto de $CodProbInst$ do modelo.

A criação do $cCodProbInst$ estabelece a divisão das instruções em duas categorias de comportamento: as instruções determinísticas e as probabilísticas.

Definição 5.2 (Instrução Determinística-ID) .

Seja a instância de instrução I_i e $Prob_i$ a sua probabilidade de gerar mudança no fluxo de execução da aplicação, I_i é uma instância de instrução determinística, se e somente se, $Prob_i \in \{0, 1\}$.

Definição 5.3 (Instrução Probabilistica-IP) .

Seja a instância de instrução I_i e $Prob_i$ a sua probabilidade de gerar mudança no fluxo de execução da aplicação, I_i é uma instância de instrução probabilística, se e somente se, $Prob_i \in [0, 1]$.

Seja $cProb$ o conjunto de probabilidades associado às instruções de uma aplicação. Identicamente, seja $cProb_{ID}$ e $cProb_{IP}$ os conjuntos de probabilidades associadas, respectivamente, às instruções determinísticas e probabilísticas dessa aplicação. Observa-se, como conseqüência das Definições 5.2 e 5.3, que

$$cProb = cProb_{ID} \cup cProb_{IP} | cProb_{ID} \subset cProb_{IP}.$$

Em outras palavras, o modelo permite que instruções probabilísticas apresentem comportamento determinístico, mas não o contrário. O que é coerente com a percepção de eventos determinísticos como casos particulares de uma realidade probabilística. Com base nessas novas definições, estabelece-se um modelo CPN-probabilístico de instrução e aplicação. Os modelos probabilísticos de instrução e aplicação são extensões dos modelos definidos no Capítulo 4, sendo portanto definidos a partir de agora em função das definições anteriores.

Definição 5.4 (Modelo CPN-Probabilístico de Instrução Ordinária-MCPIO) .
Seja $MCIOrd_n$ um modelo CPN de instrução ordinária associado à instrução $Inst_n$, a rede $MCPIO_n$, de estrutura idêntica à $MCIOrd_n$ (Definição 4.7), tal que,

$$\{T_k\}_{MCPIO_n} = \{InsName_i | FCod(InsName_i) = cCodProbInst_{Name}\},$$

define o modelo CPN-Probabilístico de instrução ordinária de $Inst_n$.

O modelo $MCPIO$ é estruturalmente idêntico ao modelo $MCIOrd$ (Definição 4.7), representando, portanto, apenas mudanças determinísticas no fluxo de execução, via distribuidor de *token*. Um $MCPIO$, portanto, representa uma instrução de comportamento determinístico.

Definição 5.5 (Modelo CPN-Probabilístico de Instrução Condicional-MCPIC) .
Seja $MCICon_n$ um modelo CPN de instrução de desvio condicional associado à instrução $Inst_n$, a rede $MCPIC_n$, de estrutura idêntica à $MCICon_n$ (Definição 4.8), tal que,

$$\{T\}_{MCPIC_n} = \{Jump, NotJump\} \cup \{InsName_i | FCod(InsName_i) = cCodProbInst_{Name}\},$$

define o modelo CPN-Probabilístico de instrução de desvio condicional da instrução $Inst_n$.

Definição 5.6 (Modelo CPN-Probabilístico de Instrução-MCPI) .
*Seja uma instrução $Inst_n$ e a rede de Petri colorida descrita pela tupla $MCPI_n = (\Sigma, P, T, A, N, C, G, E, I) | MCPI_n \in (\{MCPIO_k\} \cup \{MCICon_k\})$.
$MCPI_n$ é definida como modelo CPN-Probabilístico de instrução de $Inst_n$.*

Dadas as Definições 5.4 e 5.5, observa-se que o modelo CPN-Probabilístico de aplicação pode ser construído aplicando a mesma função de fluxo do modelo determinístico (vide Definição 4.14), preservando assim o mesmo conceito de procuradores de instrução (vide Definições 4.10, 4.11 e 4.12). Um modelo CPN-Probabilístico de aplicação ($MCPA$) é estruturalmente idêntico a um MCA. A natureza probabilística do modelo é estabelecida pela associação dos procuradores de instrução aos modelos CPN-Probabilísticos de instrução ($MCPI$)

Definição 5.7 (Modelo CPN-Probabilístico de Aplicação-MCPA) .

Seja $ApCod_n$ uma aplicação de código e $MCPA_n$ uma rede de estrutura idêntica e construída com os mesmos mecanismos do MCA_n (Definição 4.17), tal que, para $MCPA_n$ tem-se: $SA(cT_j)_{cT_j \in SN} = MCPI_j | MCPI_j \in cMCPI_n^{Ap}$. $MCPA_n$ define o modelo CPN-Probabilístico de aplicação de $ApCod_n$, onde $cMCPI_n^{Ap}$ é o conjunto de $MCPIs$ associado à $ApCod_n$.

Em uma apresentação não formal da Definição 5.7, pode-se afirmar que um $MCPA$ é um MCA no qual: (i) os $MCIs$ ordinários não descrevem o comportamento da instrução - apenas seu consumo e tempo de execução - e (ii) os $MCIs$ de desvio condicional definem a ocorrência de desvio na instrução em função da avaliação de uma variável aleatória.

Definição 5.8 (Modelo CPN-Probabilístico de Descrição-MCPD) .

Seja $cMCPI$ o conjunto de modelos CPN-Probabilístico de instrução de uma arquitetura, cRH o conjunto de seus recursos de hardware e cFI um conjunto de funções internas. O modelo CPN-Probabilístico de descrição dessa arquitetura é definido pela tupla $MCPD = (cMCPI, cRH, cFI)$.

A entidade CI continua encapsulada pelo *token*, sofrendo alteração apenas nas SAs de instrução e no registrador PC. O cRH é mantido de forma a permitir que sejam introduzidas SAs de operação. Isso estabelece a possibilidade de uma medida estatística do uso do *hardware*. O distribuidor e a transição raptora de *token* preservam a mesma funcionalidade básica presentes no MCD. As mudanças limitam-se às funções internas invocadas por essas entidades. A simulação probabilística da rede compreende a simulação de um $MCPA$ ciclicamente, até que o padrão estatístico dos resultados estejam adequados ao erro e ao nível de confiança admissíveis.

Definição 5.9 (Simulação Probabilística) .

Processo de simulação cíclica do $MCPA$ em que o critério de parada é baseado no mecanismo Monte Carlo de avaliação estatística dos resultados.

O critério de parada é construído como uma conjunção de dois mecanismos. Primeiramente, é verificado se a simulação da aplicação chegou ao fim, usando o mesmo mecanismo presente em um MCA, instrumentação do código por inserção de uma instrução RET isolada. Uma vez que a simulação chegue ao fim, funções de avaliação do critério de *Monte Carlo* são invocadas. Caso os resultados parciais estejam dentro dos valores especificados para erro e intervalo de confiança, os ciclos de simulação são encerrados e arquivos de saída são gerados, caso contrário, um novo ciclo de simulação se inicia. O distribuidor de *token* continua responsável pela avaliação do critério de parada. A raptora de *token* opera agora, não avaliando *flags* de interrupção, mas sim uma variável aleatória construída de acordo com a probabilidade de ocorrência de interrupções. Introduzidas essas novas entidades, o modelo CPN-Probabilístico de descrição ($MCPD$) mantém a mesma regra de formação do MCD.

O $MCPD$ pode ser descrito como um MCD no qual o $cMCI$ é substituído por um $cMCPI$, e funções de avaliação estatística do critério de parada são introduzidas no cFI. De fato, a grande diferença entre o MCD e o $MCPD$ está no comportamento do $MCPI$ e na avaliação do critério de parada. Na descrição do $MCPI$ de uma instrução probabilística o $CodProbInst$ associado avalia uma variável aleatória, o desvio é realizado em função do intervalo de ocorrência dessa variável. A geração do valor dessa variável é realizada seguindo uma distribuição constante de probabilidade. Uma vez estabelecidas as devidas modificações no compilador Binário-CPN, o modelo CPN-Probabilístico de descrição pode gerar automaticamente um modelo CPN-Probabilístico de aplicação.

5.2.1 Definindo o Elemento de Análise

O elemento básico de análise na simulação de um $MCPA$ é o cenário. O cenário é um conceito abstrato que define as condições de operação da aplicação. Na abordagem apresentada neste trabalho, o termo cenário identifica um conjunto não enumerável de possíveis comportamentos da aplicação. Para poder especificar tal conjunto, os comportamentos são descritos em termos da combinação de prováveis fluxos de execução da aplicação. Como já exposto, o $MCPA$ descreve tais fluxos por meio das probabilidades de desvio associadas às instruções de desvio condicional. Com base nisso, o cenário é concretizado no modelo como um conjunto de probabilidades associadas às instruções probabilísticas presentes no $MCPA$. Uma vez que esse conjunto de probabilidades estabelece uma condição a ser simulada, ele será, a partir de agora, denominado *cenário de simulação.*

Definição 5.10 (Cenário de Simulação-CS) .
Seja $MCPA_n$ o modelo CPN-Probabilístico de aplicação da aplicação $ApCod_n$, formado a partir do $cCodProbInst_n$. Seja ainda, f_{ext_prob} uma função extratora do conjunto de probabilidades cProb, tal que, $f_{ext\_prob} : cCodProbInst_n \rightarrow cProb_n$. O conjunto $cProb_n$ é definido como o cenário de simulação de $MCPA_n$.

Uma vez que o cenário de simulação define uma condição de operação, o modelo pode explorar tais condições pela variação do CS. A exploração livre do espaço de probabilidades dos fluxos de execução gera, literalmente, um espaço infinito de CSs a serem analisados. Esse espaço infinito pode ser limitado, embora continue com cardinalidade infinita, eliminando-se os CSs incoerentes com o sistema durante a descrição. Para torná-lo finito, contudo, faz-se necessário definir o espaço de probabilidades de fluxo como um conjunto discreto e finito, ou seja, associar a cada instrução de desvio condicional um conjunto discreto e finito de probabilidades, em que cada elemento é a probabilidade daquela instrução em um CS. O processo pelo qual se define os CSs de uma aplicação, eliminando os CSs inverossímeis e aplicando um conjunto discreto e finito de probabilidades, é aqui definido como *modelagem probabilística*. A questão que aparece em seguida é: como realizar uma modelagem probabilística da aplicação, de forma que o modelo de simulação realize uma exploração eficiente do espaço de probabilidades de fluxo? Uma vez que, para cada instrução existe um conjunto de probabilidades, o espaço a ser explorado explode em tamanho quando avaliadas todas as combinações dos valores de probabilidades de fluxo, ou seja, todos os CSs implementáveis. Adicionalmente, avaliar se tais combinações são coerentes na real operação do sistema é um grande desafio. Para tratar essa questão, o modelo de simulação opera de forma a variar apenas a probabilidade de uma instrução por vez. A modelagem probabilística deve então definir a aplicação nesses termos. Um CS de referência (rCS) é estabelecido pela escolha de um cenário típico ao sistema, e os demais cenários de simulação são criados a partir do CS de referência, por variação da probabilidade de uma instrução por vez. Em outras palavras, conjuntos de cenários são gerados em função de uma instrução, havendo portanto uma associação entre uma instrução e um conjunto de cenários. Cabe notar que um desvio condicional é um indicador de evento no sistema, seja decorrente de um estímulo externo ou interno. Sendo assim, os CSs gerados estão diretamente relacionados aos cenários de eventos do sistema. A limitação dessa abordagem reflete-se no fato do modelo não poder explorar a interdependência probabilística dos eventos automaticamente[1]. Tal exploração, contudo, pode ser reali-

[1]Equivale a questionar se uma instrução opera efetivamente com uma probabilidade p_x, dado que uma outra está operando com probabilidade p_y.

zada pela criação e simulação manual de CSs.

5.2.2 Modelagem Probabilística

A descrição do comportamento probabilístico da instrução é realizada por um parâmetro denominado *anotação de instrução*. A anotação de instrução inclui informações que definem o comportamento da instrução em tempo de simulação probabilística. A anotação de instrução encapsula: (i) o conjunto de probabilidades que definem os CSs gerados pela instrução,(ii) o modo de simulação da instrução, (iii) o valor da probabilidade da instrução no CS de referência, e (iv) o número máximo de desvios consecutivos permitidos para a instrução. O conjunto de probabilidades que definem os CSs gerados pela instrução, nominados a partir de agora *conjunto de probabilidades de instrução*. O conjunto de probabilidades de instrução é um conjunto discreto, finito e ordenado, de valores de probabilidades da instrução, descritos na forma de intervalo de valores, no qual a natureza discreta do intervalo é estabelecida por uma constante de passo.

Definição 5.11 (Conjunto de Probabilidades de Instrução- cPI) .
Seja I_i uma instância de instrução, p_{in} a probabilidade que define o cenário inicial gerado por ela, p_{fin} o cenário final e $st|st \in (R^+ - \{0\})$, a constante de passo. O conjunto de probabilidades de instrução é definido pela tupla $cPI_i = (IntProb, st)|IntProb = [p_{in}, p_{fin}]$.

O identificador do modo de simulação probabilística ($ModProb$) da instância de instrução é um valor do tipo booleano que sinaliza se ela gera CSs. Caso a instância de instrução não gere CSs, o cPI não é levado em consideração em tempo de simulação. Cabe esclarecer que uma instância de instrução que não gera cenários é aquela cuja probabilidade de desvio pode ser modelada como fixa, e em assim sendo, descrita no cenário de referência. Um modelo probabilístico de aplicação em que todas as instâncias de instrução são modeladas com probabilidade fixa($Probfix$), define apenas um cenário de simulação, ou seja, o cenário de referência (rCS).

Definição 5.12 (Anotação de Instrução-AI) .
Parâmetro que identifica os padrões de comportamento probabilísticos da instância de instrução. Seja uma instância de instrução I_i, cPI_i seu conjunto de probabilidades, $ModProb_i$ seu modo de simulação probabilística, $Probfix_i$ sua probabilidade no CS de referência, $Wloop_i$ o maior número de desvios consecutivos gerados pela instrução e rCS o cenário de simulação de referência. A anotação de instrução de I_i é definida pela tupla:
$AI_i = (cPI_i, ModProb_i, Probfix_i, Wloop_i)$, *onde* $ModProb_i = true \Rightarrow \nexists CS \neq rCS$.

O parâmetro $Wloop_i$ estabelece dois importantes recursos em tempo de simulação: (i) permite a atribuição de comportamento determinístico em instruções probabilísticas e (ii) permite mapear o limite de iteração dos laços de execução. Como já apresentado, $cProb = cProb_{ID} \cup cProb_{IP} | cProb_{ID} \subset cProb_{IP}$, o que permite que uma instrução probabilística possa assumir comportamento determinístico. Para tal, basta associar a ela uma AI com a estrutura : $AI_i = (cPI_i, true, 1.0, Wloop_i)$ ou $AI_i = (cPI_i, true, 0.0, Wloop_i)$. Uma vez que $ModProb_i = true$, a instrução não gera cenários particulares, apenas contribui com seu $Probfix_i$ para o rCS. Simultaneamente, probabilidades de desvio 1.0 ($Probfix_i = 1.0$) ou 0.0 ($Probfix_i = 0.0$) estabelecem comportamentos determinísticos, mesmo em instruções conceitualmente probabilísticas, como instruções de desvio condicional. Nos casos em que $Probfix_i = 1.0$ e a instrução implementa uma estrutura de laço, faz-se necessário a limitação do número de iterações, essa limitação é estabelecida pelo valor de $Wloop_i$. Similarmente, em instruções de comportamento probabilístico que implementam estruturas de controle do tipo *while* ou *for-next*, o número de iterações precisa ser limitado a um máximo valor admissível para o sistema, o parâmetro $Wloop_i$ permite modelar o limite de iterações possíveis no sistema, evitando que a natureza aleatória da simulação introduza mais iterações do que dita o contexto funcional do sistema.

Apresentados tais conceitos, a modelagem probabilística de um código pode ser definida como um processo de duas etapas: (i) mapeamento de especificações probabilísticas do sistema em AIs e (ii) mapeamento das AIs no $cMCPI_n^{Ap}$ da $ApCod_n$ alvo. A primeira etapa caracteriza o sistema em termos de probabilidades de comportamentos, eliminando os padrões de comportamentos inverossímeis. A segunda instancia um subconjunto de $cMCPI$ com as especificações probabilísticas do sistema. O conjunto de especificações probabilísticas do sistema ($cEPS$) não é proposto formalmente neste trabalho. O $cEPS$ pode ser, contudo, informalmente compreendido como um conjunto de tuplas cujos elementos identificam os eventos do sistema, suas probabilidades - cenários de simulação - e sua relação com as instruções probabilísticas da aplicação. Podendo ser construído mediante uma descrição *ad hoc* do sistema, ou por meio de uma base de dados referente à estatística de ocorrência dos eventos do sistema.

Definição 5.13 (Modelagem Probabilística-MP) .

Seja $ApCod_n$ uma aplicação de código, $cEPS_n$ um conjunto de especificações probabilísticas do sistemas no qual $ApCod_n$ está inserido, e cAI_n um conjunto de anotações de instrução, tal que, $\exists f_{Desc} | f_{Desc} : cEPS_n \rightarrow cAI_n$. Define-se como modelagem probabilística de $ApCod_n$ o processo descrito pela função $f_{mp} : ApCod_n \times cAI_n \times cMCPI \rightarrow cMCPI_n^{Ap}$

Conceito de Evento e Modelagem Probabilística

Um *evento* é definido como um ente primitivo associado à ação de transição de estados em um sistema [20]. A depender da definição de estado, o conceito de evento pode ser associado a um ato específico no sistema, relacionado a um significado abstrato humano, como o pressionar de uma tecla para atender uma ligação ao celular, ou a uma operação atômica no *hardware*, como a transição de nível lógico em uma via de controle. Dada a possibilidade de escalonamento dos níveis de abstração de descrição do evento, um evento de maior nível de abstração pode ser descrito como um conjunto de eventos de menor nível de abstração.

O $cEPS$ pode ser descrito em termos de eventos associados à operações de sentido mais abstrato no sistema, por exemplo, acessar uma rede de comunicação. Tais eventos são definidos aqui como *eventos primários*.

Definição 5.14 (Evento Primário) .
Ação sistêmica abstrata, representando uma ação de interação com o ambiente físico ou outros sistemas.

O $cEPS$ pode ser refinado em função do conjunto de *eventos primários* e suas probabilidades de ocorrência. Abordando o contexto da aplicação de *software*, os eventos primários habilitam um conjunto de possíveis fluxos de execução da aplicação, cada fluxo caracteriza uma possível ação dentro da aplicação, definindo uma *ação primária*.

Definição 5.15 (Ação Primária) .
Execução de um fluxo específico do código da aplicação.

Cada fluxo de execução é definido por um conjunto de desvios implementados pelas instruções de desvio. Cada desvio caracteriza um *evento secundário*.

Definição 5.16 (Evento Secundário) .
Variação do fluxo de execução devido a uma instrução de desvio condicional.

Conceitualmente, a modelagem probabilística de uma aplicação deve então partir de um conjunto de probabilidades de eventos primários do sistema e descrever o conjunto de probabilidades dos eventos secundários. Uma vez definido o conjunto de probabilidade dos eventos secundários, esse deve ser descrito em um cAI e mapeado no $cMCPI^{Ap}$, conforme Definição 5.13. Cabe observar que cada evento primário deve ser mapeado em um conjunto de eventos secundários decorrentes. No atual estágio da pesquisa foram escolhidos exemplos onde houvesse um mapeamento biunívoco entre

eventos primários e secundários. O presente trabalho centra-se em um processo *ad hoc*, no qual o projetista constrói o $cEPS$ de acordo com sua percepção do sistema, perfazendo a função f_{Desc} ao descreve-lo na forma de cAI. As AIs presentes no código fonte são automaticamente mapeadas no $cMCPI^{Ap}$ pelo compilador Binário-CPN em modo probabilístico.

Uma vez definido o $cMCPI_n^{Ap}$, o $MCPA_n$ é formado aplicando a Definição 5.7. Contudo, para que o $MCPA_n$ possa ser objeto de uma simulação probabilística, faz-se necessário que os parâmetros que definem os critérios de parada sejam inseridos nas respectivas variáveis do cFI. Esse processo é abordado na próxima seção.

5.2.3 Parâmetros de Simulação

Os parâmetros de simulação definem os termos em que a simulação probabilística ocorrerá. Tais parâmetros são encapsulados como um parâmetro composto denominado *anotação de cabeçalho*. A anotação de cabeçalho encapsula: (i) o nível de confiança da estimativa (ii) o erro máximo admitido, (iii) a métrica a ser avaliada pelo critério de parada (energia, potência ou tempo de execução), e (iv) a quantidade mínima de ciclos de simulação do $MCPA$.

Definição 5.17 (Anotação de Cabeçalho-AC) .
Seja $ApCod_n$ uma aplicação de código a ser avaliada, em que IC_n, Er_n Mt_n e Qs_n são, respectivamente: o nível de confiança desejado, o erro máximo, a métrica avaliada pelo critério de parada, e a quantidade mínima de ciclos de simulação almejados. Define-se como anotação de cabeçalho de $ApCod_n$ a tupla $AC_n = (IC_n, Er_n, Mt_n, Qs_n)$.

O parâmetro Qs_n estabelece o volume de dados por simulação probabilística, visto que este volume é proporcional ao número de ciclos de simulação. Isso permite que os resultados possam ser analisados pela construção da distribuição de probabilidade dos intervalos de valores encontrados para as métricas. A importância desse parâmetro ficará clara na Seção 5.5, na qual serão discutidos os mecanismos de avaliação de resultados.

A massa de dados para a avaliação dos padrões de consumo da aplicação é gerada mediante a simulação probabilística do seu $MCPA$. O presente modelo de simulação probabilística permite dois modos de operação: o modo normal e o modo varredura (*sweep*). O modo normal executa a simulação probabilística do $MCPA$ explorando apenas o CS de referência (rCS). No modo de varredura são executadas simulações probabilísticas do $MCPA$ para cada CS definido no cAI, varrendo os $cPIs$ das instruções habilitadas para esse modo ($ModProb_i = false$). Adicionalmente, cenários

específicos podem ser simulados como rCS, sendo carregados no modelo mediante alteração dos $Probfixs$ de todas as instruções.

5.3 Modelo CPN-Probabilístico-i8051

Os conceitos apresentados nas seções anteriores serão demonstrados na criação de um $MCPD - i8051$. Identicamente ao modelo $MCD - i8051$, o ambiente CPNTools foi utilizado para a construção e validação do conjunto $cMCPI$, bem como validação e execução dos $MCPAs$ de teste. A implementação de $MCPAs$ automaticamente, a partir do código executável, foi realizada pela modificação do compilador Binário-CPN e implementação de mecanismo de suporte à modelagem probabilística da aplicação. O modelo CPN-Probablístico do i8051 foi especificado com base em um conjunto pré-estabelecido de requisitos. O conjunto de requisitos reproduz essencialmente os mesmos parâmetros de modelagem do $MCD - i8051$. Observa-se as seguintes características: (i) O cRH do $MCD - i8051$ é preservado de forma a permitir que o modelo acesse operações no *hardware*, (ii) as funções internas são ampliadas, comportando a análise de critérios estatísticos, (iii) as instruções são descritas em termos de sua probabilidade de gerar desvios no fluxo de execução da aplicação, (iv) as interrupções de *hardware* não são modeladas e (v) existe um conjunto de atribuições em SAs, sendo executadas no modelo de instrução ou no conjunto de recursos de *hardware*.

5.3.1 Definindo o Contexto Interno

O $MCPD$ mantém a descrição comportamental para três tipos de instruções: as de chamadas de subrotinas, as de retorno e as de desvio incondicional. Isso é feito de forma a garantir os desvios determinísticos, tais como chamadas e retornos de subrotinas. Para tal, o $MCPD$ deve possuir um modelo de memória de forma a permitir a construção da pilha. Por outro lado, o transporte desse modelo no CI penaliza o desempenho da simulação. Para resolver esse problema de desempenho, um mecanismo de passagem por referência foi "sintetizado" no CPNTools[2]. O mecanismo constrói um identificador de memória (meta-referência), que é avaliado pelas funções `read_aux` e `write_aux` ($\{read\_aux\ , write\_aux\} \subset cFI$). A memória é modelada como um *array* e acessada de acordo com o valor do identificador. A Figura 5.1 apresenta a função de acesso aos modelos de memória por meio desse identificador.

[2]Uma vez que a linguagem CPN-ML não suporta a manipulação de estruturas como listas e *arrays* por referência, ou mesmo o encapsulamento do tipo *array* no *token*.

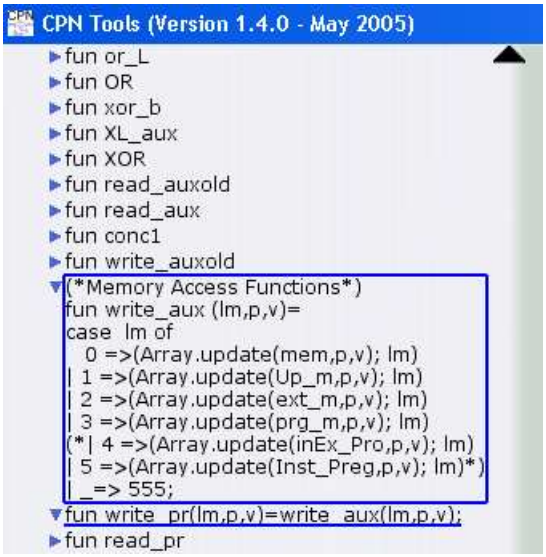

Figura 5.1: Mecanismo de passagem por referência implementado no CPN-ML.

Isso implica em reformulação do tipo (*colorset*) *Context*, com uma considerável simplificação de seus tipos componentes. Sendo esse definido agora como:

```
colset Memory = inte;
colset Context =
  record mem:Memory*Upmem:Memory*PC:inte*ext_mem:Memory*program:instpad;
```

Como se pode notar, o cSA foi retirado do CI, o que torna o *token* mais simples. As sondas no *MCPD-i8051* são definidas como variáveis globais do ambiente e marcadas por meio da função `record()` ($record() \in cFI$). Identicamente ao $MCD - i8051$, o instanciamento do CI no modelo $MCPD - i8051$ é feito pela definição da variável valor de contexto (*value*), que é inserida no *token*. O construtor de *value* é definido como sendo:

```
val value= {mem=write_pr(0,129,7),Upmem=1,PC=0,ext_mem=2,program=3};  .
```

No qual, as variáveis de memória não carregam listas, mas sim inteiros que operam como identificadores (meta-referências) dos *arrays*, que modelam a memória. Uma operação em memória é realizada por uma função de escrita ou leitura que opera sobre um *array* - não mais listas - e retorna o identificador (meta-referências) da memória alvo. Por exemplo, a operação `mem=write_pr(0,129,7)`, descrita acima, opera sobre a memória RAM do i8051, identificada pelo código 0, escrevendo na posição 129 o valor 7.[3] As diferenças estruturais do CI no $MCPD - i8051$, em relação ao $MCD - i8051$,

[3]Na arquitetura i8051 a posição 129 de memória contém o registrador SP (*Stack Pointer*) que tem por *default* valor inicial igual a 7.

garantem maior velocidade de simulação e simplicidade na construção dos $MCPIs$.

5.3.2 Construíndo $MCPIs$

Uma vez que não há comportamento específico para ser descrito, a construção do $cMCPI$ de um $MCPD$ é extremamente simples. O primeiro passo é identificar na arquitetura alvo as instruções probabilísticas e determinísticas. Em seguida identificar aquelas cuja descrição de comportamento deve ser mantida como no MCD. Instrução do tipo CALL, RET, RETI e JMP, são mantidas em uma descrição determinística, uma vez que sua funcionalidade é o desvio determinístico do fluxo de execução. Cada $MCPI$[4] é construído associando-se dois gabaritos de $CodProbInst$, um para instrução determinística e outro para probabilísticas. As Figuras 5.2 e 5.3 apresentam, respectivamente, os $MCPI$'s de uma instrução determinística (instrução ANL A,direto) e probabilística (instrução CJNE A,#dado). O $CodProbInst$ determinístico, apenas marca as SAs e atualiza o PC, não havendo novos parâmetros além daqueles já conhecidos do modelo determinístico.

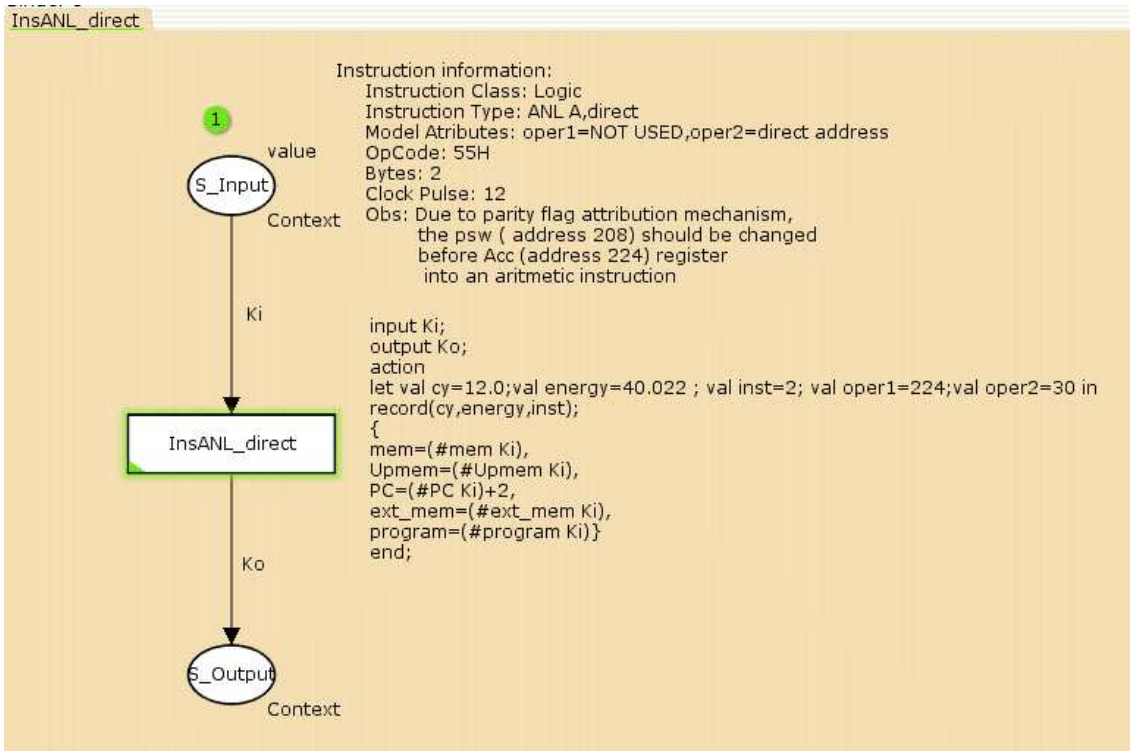

Figura 5.2: Modelo CPN-Probabilístico da instrução ANL A,direto.

Por sua vez, o $CodProbInst$ probabilístico carrega um novo conjunto de atributos, referentes: (i) aos dados que devem ser carregados a partir da AI e (ii) ao controle de

[4]Identicamente ao modelo determinístico, os $MCPI$ são construídos como modelos de aplicação.

modo de simulação da instrução. Em relação à AI, o $CodProbInst$ define $fixProb$ e $wloop$, referentes, respectivamente, aos parâmetros $Probfix$ e $Wloop$ da AI. O cPI e o $ModProb$ são carregados como elementos, respectivamente, das listas internas `ProbList` e DetermList ($\{$`ProbList`, `DetermList`$\} \subset cFI$)(vide Figura 5.4). O processo de carga da probabilidade da instrução, em tempo de simulação, é promovido pela atribuição:

```
val prob=if
(!InstSweep= thisInstSweep) andalso !SweepEnabled
then (!globprob)
else fixProb;.
```

A função de atribuição verifica se o modo varredura está ativo (`!SweepEnabled`) e se a instrução está habilitada na instância de varredura, (`!InstSweep=thisInstSweep`), de forma que `SweepEnabled` e `InstSweep` são variáveis globais do modelo [5] e `thisInst Sweep` um parâmetro local, inerente à posição do $MCPI$ no $MCPA$, que estabelece em que instância de varredura o cPI da instrução modelada é varrido. O valor 0 no `thisInstSweep` caracteriza a presença de $ProbMod = true$ no cAI, ou seja, a instrução não deve ser habilitada em nenhuma instância da varredura. Esse mecanismo de restrição é garantido pelo fato da variável `InstSweep` ter valor inicial igual a 1 (vide Figura 5.4). Quando ativa no modo de varredura, a instrução tem sua probabilidade (`prob`) definida pela variável `globprob`, que é sistematicamente carregada com os valores do cPI da instrução, caso contrário, a instrução opera com probabilidade `fixProb`.

O comportamento probabilístico do $MCPI$ é definido pela atribuição:

```
Upmem=write_pr((#Upmem Ki), 0,
 if(uniform(0.0,1.0)<=prob) andalso (Array.sub(LoopCounter,inst)<wloop-1)
  then( (jumpflg:=1;
    Array.update(LoopCounter,inst,Array.sub(LoopCounter,inst)+1)); 1)
  else( (jumpflg:=0;
    Array.update(LoopCounter,inst,0)); 0)                          ).
```

A função `uniform(0.0,1.0)` (`uniform(0.0,1.0)` $\notin cFI$) é uma geradora de números aleatórios no intervalo $[0.0, 1.0]$, cuja função densidade de probabilidade é constante. Como o predicado `(uniform(0.0,1.0)<=prob)` avalia se o valor aleatório gerado é menor ou igual à `prob`, a probabilidade de seu valor ser *true* é descrito por $P(0 \leq X \leq$

[5] $\{$`SweepEnabled`, `InstSweep`$\} \subset cFI$

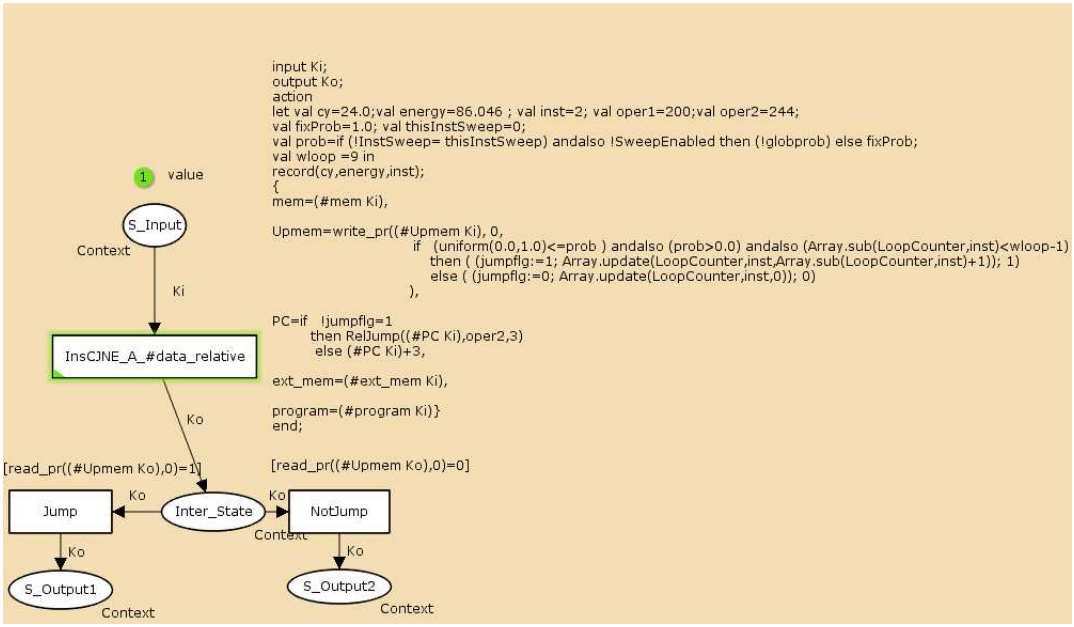

Figura 5.3: Modelo CPN-Probabilístico da instrução CJNE A,#dado.

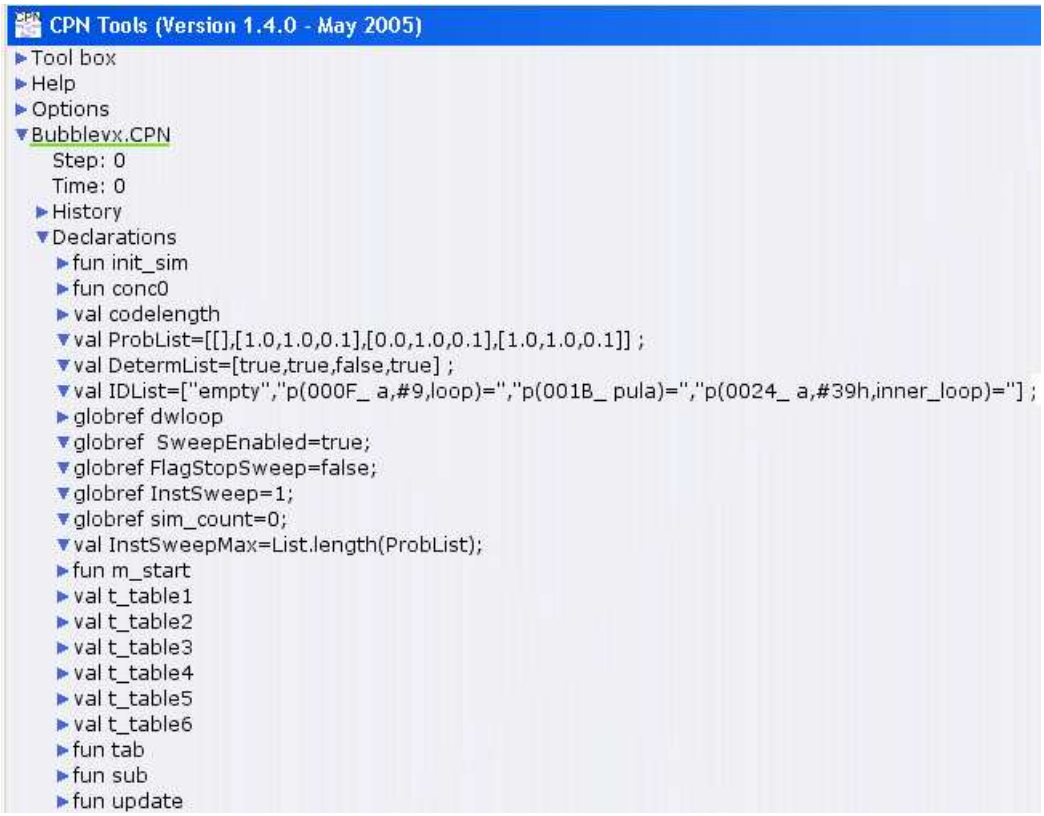

Figura 5.4: Variáveis globais que descrevem cPI e o modo de varredura.

prob), onde X é uma variável aleatória gerada por `uniform(0.0,1.0)`. Sendo assim,

aplicando-se a definição básica da função densidade de probabilidade, tem-se:

$$P(a \leq X \leq b) = \int_a^b f(x)dx = 1.$$

Onde $\forall x \in [a,b] \Rightarrow f(x) = k | k = \frac{1}{b-a}$, logo $[a,b] = [0.0, 1.0] \Rightarrow k = 1$. Conseqüentemente:

$$P(0 \leq X \leq prob) = \int_0^{prob} kdx = prob.$$

Dessa forma, o predicado `(uniform(0.0,1.0)<=prob)` é *true* aleatoriamente com probabilidade `prob`, o que garante que os sinalizadores de desvios, representados por `jumpflg`, operando como sinalizador de carga do PC (vide Figura 5.3) e pelo conteúdo da posição 0 do modelo de memória alta[6], também o sejam. Esse mecanismo apresenta um real processo de escolha probabilística em um único passo de simulação da rede, contrapondo-se à abordagem freqüencista apresentada em [40].

O mecanismo que limita o número de desvios consecutivos a $Wloop$ é definido pela conjunção do predicado `(Array.sub(LoopCounter,inst)<wloop-1)` com as linhas:

```
(a)  Array.update(LoopCounter,inst,Array.sub(LoopCounter,inst)+1)); 1)
```

e

```
(b)  Array.update(LoopCounter,inst,0)); 0).
```

O *array* `LoopCounter` é uma variável interna do modelo (`LoopCounter` $\in cFI$), na qual cada posição acumula o número de desvios consecutivos de uma instrução. A instância da instrução no $MCPA$ opera como indexador de `LoopCounter`. Sendo assim, a linha `(a)` incrementa um contador de desvio, armazenado em `LoopCounter`, e `(b)` o zera. Enquanto o predicado `(Array.sub(LoopCounter,inst)<wloop-1)` for igual a *true* `(a)` é avaliada, do contrário, avalia-se `(b)`.

Uma vez apresentados os gabaritos de formação do $cMCPI$, cabe ressaltar que tais gabaritos homogenizam as instruções em dois grupos: as determinísticas e as probabilísticas. Uma vez que não há comportamento específico, as instruções possuem comportamento probabilístico definido por sua AI, que traz informação da modelagem probabilística da aplicação, e não da arquitetura. Seus únicos vínculos com a arquitetura do processador são os parâmetros de consumo de energia, tempo de execução e incremento do PC. Descrever o $cMCPI$ de uma arquitetura resume-se a instanciar o gabarito adequado de $CodProbInst$ no $MCPI$ e carregar os atributos referentes ao consumo de energia, tempo de execução e incremento do PC. Dessa forma, o modelo $MCPD$, sem sondas de *hardware*, é conceitualmente independente dos aspectos funcionais da arquitetura alvo.

[6]Identicamente ao modelo determinístico.

5.3.3 Definindo o Mecanismo de Parada

Similarmente ao modelo determinístico, no $MCPD - i8051$ avalia-se o critério de parada em um código CPN-ML associado à transição T'_{calc} do distribuidor de *token*. No $MCPD - i8051$, como no $MCD - i8051$, a transição T'_{calc} é denominada *CalculantingPC*. A Figura 5.5 mostra a estrutura CPN associada à transição *CalculantingPC*. Primeiramente é verificado o término da simulação do $MCPA$ - Cada simulação é um ciclo dentro da simulação probabilística do $MCPA$ -, aplicando-se o mesmo mecanismo descrito na Seção 4.15. Após cada simulação do $MCPA$, a função `StopChecker()` é invocado para avaliar os resultados parciais. Como definido pelo modelo, a parada da simulação probabilística do $MCPA$ é estabelecida por dois critérios: número mínimo de simulações e adequação estatística dos resultados. O número mínimo de simulações é a concretização do parâmetro Qs_n presente na definição dos parâmetros de simulação (Definição 5.17), sendo representado no $MCPD-i8051$ pela constante `N_iter_Min`(vide Figura 5.5). Caso o número mínimo de simulações não tenha sido atingido, os valores de consumo de energia, tempo de execução e potência são concatenados ao final de uma lista e uma nova simulação é iniciada. Quando o total de simulações atingir o mínimo especificado em `N_iter_Min`, a lista é avaliada pelas funções `CalcAvPlog()` e `getSD()` que calculam, respectivamente, o valor médio e o desvio padrão dos resultados parciais concatenados na lista. Uma vez calculados os parâmetros estatísticos dos resultados parciais, o critério de Monte Carlo é avaliado a partir da inequação:

$$N\_Iteration \leq \left(\frac{t * getSD(plog)}{ErrorMax * CalcAvPlog(plog)} \right)^2 .$$

Na qual *plog* é a lista de resultados parciais, $ErrorMax$ é a concretização do parâmetro Er_n da AC, e t é um fator de normalização da função de distribuição da probabilidade assumida. O fator t é definido em função do número de simulações (amostras) realizadas e do nível de confiança almejado, estando mapeado em tabelas acessíveis através da função `getVtab()` (vide Figura 5.5). Se a inequação acima for verdadeira, uma nova simulação é iniciada, caso contrário, a simulação probabilística é encerrada. No modo varredura, para cada CS, especificado em tempo de descrição probabilística da aplicação, é efetuada uma simulação probabilística. No modo normal apenas uma simulação probabilística é efetuada, referente ao CS de referência (rCS). Ao final de cada simulação probabilística, os resultados parciais são salvos em arquivos específicos, um arquivo para cada cenário, por meio da função `SampleFileOut()`. Adicionalmente, ao fechar o conjunto de cenários de uma instrução, os valores médios e os níveis de dispersão (desvio padrão), em função dos cenários, são salvos em arquivos por meio da função `Av_SDFileOUt()`. Dessa forma, uma ferramenta de análise pode

avaliar os cenários a medida que suas massas de dados são geradas.

A construção do $MCPD - i8051$ no ambiente CPNTools permitiu não apenas a validação e verificação da metodologia de modelagem proposta, mas também a criação de um arcabouço de análise completo. A seção seguinte descreve esse arcabouço.

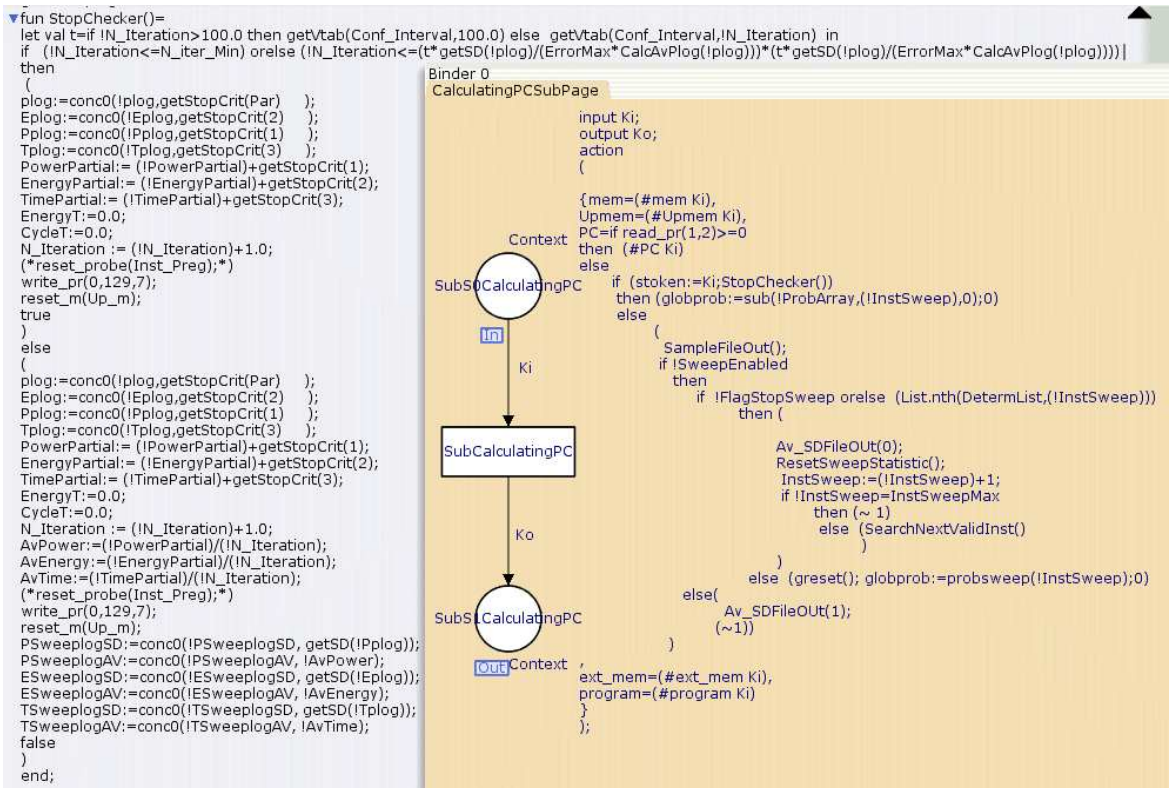

Figura 5.5: Estrutura do *CalculantingPC* e a função `StopChecker()`.

5.4 Arcabouço de Modelagem de Aplicação

O arcabouço de modelagem de aplicação consiste de um conjunto de ferramentas e processos que permitem: (i) gerar o modelo probabilístico de uma aplicação, representando-o por meio de anotações no código fonte, (ii) construir um $MCPA$ da aplicação, (iii) executar a simulação probabilística do $MCPA$, gerando uma massa de dados brutos e (iv) analisar essa massa de dados, compilando os resultados na forma de tabelas e histogramas. O processo definido como *modelagem probabilística de aplicação* é distribuído entre os itens (i) e (ii). O item (i) representa o processo de descrição associado à função f_{Desc} da Definição 5.13. Como já exposto, no atual estágio da pesquisa a função f_{Desc} é implementada manualmente de forma *ad hoc*. A Figura 5.7 ilustra esse processo. Com base em uma percepção abstrata do conjunto de especificações probabilísticas do sistema ($cEPS$), gera-se um mapeamento entre eventos primários e

secundários, inferindo-se as probabilidades dos eventos secundários com base nas probabilidades dos eventos primários. Cabe ressaltar que o $cEPS$ ainda não existe como um formato de descrição, sendo isso objeto para trabalhos futuros. O $cEPS$ deve ser entendido como a visão abstrata que o projetista tem dos eventos primários do sistema e suas probabilidades. O desmembramento dessa percepção, em termos de eventos secundários, conduz à especificação das anotações de instrução, perfazendo portanto a função f_{Desc}. Como resultado, o arcabouço gera no escopo do item (i), um código *assembly* anotado, no qual um conjunto de AIs é descrito explicitamente nas linhas de código. Adicionalmente, esse código anotado contém uma anotação de cabeçalho que encerra o parâmetro AC. A segunda etapa do modelamento probabilístico, definido pela função f_{mp}, é implementada pelo compilador Binário-CPN probabilístico apoiado por um processador de código anotado, denominado LST-Parser. O LST-Parser cumpre a função de vincular concretamente uma AI à instrução, de forma que o compilador Binário-CPN probabilístico possa construir adequadamente o $MCPA$. Para que o compilador possa carregar os parâmetros inluídos nas AIs nos respectivos $MCPIs$, ele precisa de um indexador confiável. Esse indexador é a posição de memória ocupada pela instrução. Essa informação, contudo, não está disponível no código anotado. Porém, ela é inserida nos arquivos extensão .LST naturalmente, como conseqüência do processo de montagem realizada pelo programa montador. Uma vez que os arquivos .LST preservam as anotações do código fonte, o LST-Parser opera capturando, instrução a instrução, o par posição-anotação $(EndMem, AI)$. O LST-Parser gera como saída um arquivo de mapeamento no formato XML no qual as AIs são desmembradas em elementos XML, esse arquivo é denominado XML-Prob (vide Figura 5.8).

A Figura 5.6 ilustra todo o arcabouço, descrevendo do fluxo de modelagem e análise do código anotado à geração dos resultados finais. Um aplicativo Java foi implementado de forma a prover um ambiente para edição de códigos anotados, compilação, simulação (acionando o CPNTools) e análise de resultados. A descrição desse ambiente protótipo será apresentada no Capítulo 6.

5.4.1 Construção Sintática da AC e da AI

Tanto a AC quanto a AI são introduzidas no código na condição de comentários, de forma a serem completamente transparentes para o montador, no processo de geração do código binário. A anotação de cabeçalho é colocada nas primeiras linhas do código, sendo definida pela seguinte estrutura sintática:

$$< \$IC|Er|Mt|Qs\$ > .$$

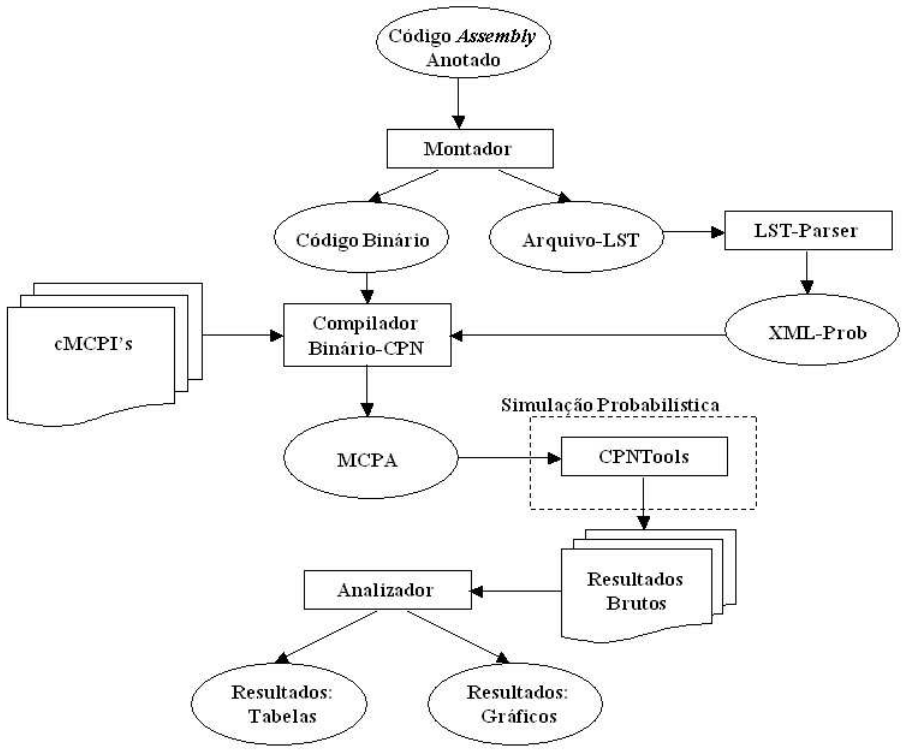

Figura 5.6: Arcabouço de modelagem, simulação e análise de códigos.

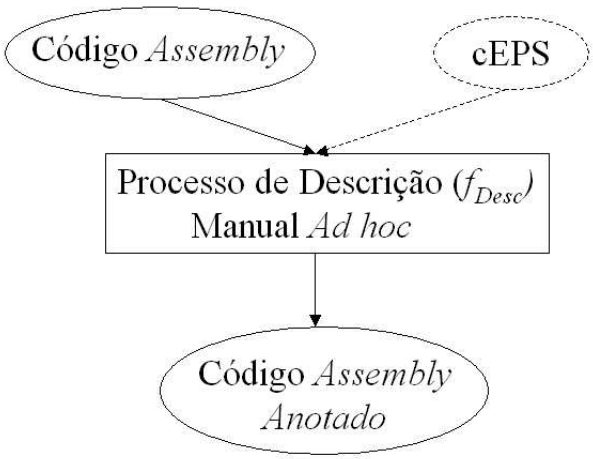

Figura 5.7: Mecanismo *ad hoc* de geração de código anotado.

Na qual IC,Er,Mt,Qs são, respectivamente: nível de confiança, erro máximo, métrica alvo e quantidade mínima de ciclos de simulação. Os parâmetros IC e Mt representam códigos identificadores dos níveis de confiança e das métricas disponíveis no modelo. A Tabela 5.1 apresenta essa codificação.

A introdução de anotações de instrução no código fonte é realizada aplicando-se a

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<probabilidades>
<Header Conf_Interval="1" ErrorMax="0.05"
N_iter_Min="1000" Par="3" />
     <elemento probabilidade="[1.0,1.0,0.1]"
instrucao="a,#9,loop" Endereco="000F" Iteracoes="9"
FixProb="1.0" DeterInst="true" />
     <elemento probabilidade="[0.0,1.0,0.1]"
instrucao="pula" Endereco="001B" Iteracoes="256000"
FixProb="0.5" DeterInst="false" />
     <elemento probabilidade="[1.0,1.0,0.1]"
instrucao="a,#39h,inner_loop" Endereco="0024"
Iteracoes="5" FixProb="1.0" DeterInst="true" />
</probabilidades>
```

Figura 5.8: Formato XML do mapeamento instrução → probabilidade.

Tabela 5.1: Identificadores dos níveis de confiança e métricas

Valor do Parâmetro		Nível	Métrica
IC	Mt	de Confiança(%)	Avaliada
1	1	90,0	Potência
2	2	95,0	Energia
3	3	97,5	Tempo de execução
4	-	99,0	-
5	-	99,5	-
6	-	99,9	-

seguinte estrutura sintática:

$$< @[p_i, p_f, st]|ModProb|Probfix|Wloop@ > .$$

Na qual $[p_i, p_f, st]$ representa o conjunto de probabilidades de instrução (cPI) descrito em termos da probabilidade inicial (p_i), final (p_f) e do passo(st), enquanto $ModProb, Probfix$ e $Wloop$ representam o modo de simulação, a probabilidade no rCS e o número limite de desvios consecutivos da instrução, respectivamente. A Figura 5.9 exemplifica anotações feitas sobre um código CRC-16 (*cycle redudance checking 16 bits*). O código é avaliado considerando nível de confiança de 90%, erro máximo de 5%, tendo o tempo de execução como métrica de parada e realizando no mínimo 1000 simulações por cenário avaliado.

5.4.2 Compilador Binário-CPN Probabilístico

O compilador Binário-CPN probabilístico é uma extensão do compilador Binário-CPN no qual é introduzido o mapeamento referente à função f_{mp}, definida no modelo pro-

```
;************Definição de Parâmetros de Simulação Probabilistica*********************************
                              ;<$1|0.05|3|1000$>

                lcall CRC16
                ret             ; Linha de instrumentaçao -> Condição de parada

CRC16:
                push    0                                       ; Save R0
                push    acc                                     ; Save Acc
                mov     r0,#8                                   ; 8 Bits In A Byte
                xrl     07,a                                    ; HI ^= Data
loop1:  clr     c                                               ; 0 Into Low Bit
                mov     a,06                                    ; D_CRC << 1
                rlc     a                                       ; Shift Left
                mov     06,a                                    ; Store Back
                mov     a,07                                    ; Get High Byte
                rlc     a                                       ; Shift Left
                mov     07,a                                    ; Store Back
                jnc     loop2   ;<@[0.0,1.0,0.1]|false|0.9|20@> ; Skip If Bit 15 Wasn't Set
                xrl     06,#10h                                 ; XOR In Polynomial High
                xrl     07,#21h                                 ; XOR In Polynomial Low
loop2:          djnz    r0,loop1;<@[0.0,1.0,0.1]|true|1.0|16@>  ; Repeat R0 More Times
                pop     acc                                     ; Recover Acc
                pop     0                                       ; Recover R0
                ret                                             ; Return To Caller

end
```

Figura 5.9: Exemplo de código anotado.

babilístico de descrição. Para tal, o compilador lê o arquivo XML-Prob e constrói uma *hashtable* com as AI, cuja chave é o endereço associado à AI. Quando o $MCPI$ de instruções probabilísticas ($MCPIC$) é instanciado, o endereço da instrução é usado para acessar a AI correspondente na *hashtable*, carregando os atributos da $MCPI$ em função dos parâmetros da AI. Adicionalmente, é interessante que o compilador permita um modo de operação em que a rede seja gerada em tamanho reduzido. A redução da extensão da rede implica em imediata redução do número de transições a serem disparadas, e conseqüentemente, no incremento da velocidade de simulação do $MCPA$.

Processos de Redução da Rede

O processo de redução de transições é possível graças à ausência de comportamento específico por instrução. Sendo assim, trechos de código podem ser modelados por uma única transição associada a um $MCPI$ equivalente. Para tal, faz-se necessário o estabelecimento de operações algébricas sobre os $MCPI$ possibilitando a construção de um $MCPI$ equivalente. Tais operações tornam-se possíveis pela representação dos $MCPI$ como vetores, em que os componentes representam os atributos numéricos do $MCPI$ - energia consumida e número de ciclos de relógio.

Definição 5.18 (Vetor Instrução -VI) .
Seja o plano cartesiano Energia×Ciclos-de-Clock, o vetor[7] *instrução associado à* $MCPI_n$ *é definido como* $\overrightarrow{MCPI_n} = (cy_n, energy_n)_n$, *onde* cy_n *e* $energy_n$ *são, respectivamente,*

[7] O termo "vetor" aqui representa a *entidade matemática vetor*, sujeita à álgebra vetorial, e não um *array* unidimensional.

o número de ciclos e consumo de energia da instrução - $energy_n = CV\,[I_n]$, vide Definição 4.21.

Axioma 5.1 .
O vetor instrução representa, no plano Energia×Ciclos-de-Clock, a contribuição da instrução ao desempenho da aplicação em termos de tempo de execução e consumo de energia.

Axioma 5.2 .
O plano Energia×Ciclos-de-Clock define um espaço vetorial linearmente independente.

Definição 5.19 (Segmento de *Patch*-sP) .
Seja $InstMem^n = \{e_k | k \in N\}$ um conjunto ordenado de posições de memória associadas à $ApCod_n$, um segmento de Patch sP_z, associado a um Patch δ, é definido como um conjunto ordenado de vetores instrução, representando uma seqüência de instruções ordinárias presentes no Patch δ. Um segmento de Patch é, portanto, definido como

$$sP_z = \left\{ (cy_k, energy_k)_k | ((cy_k, energy_k)_k \in \overrightarrow{cMCPIO_n^{Ap}}) \wedge (i \leq k \leq j) \right\},$$

onde $\overrightarrow{cMCPIO_n^{Ap}}$, i e j são, respectivamente, o conjunto de vetores instrução ordinárias da $ApCod_n$, o menor e o maior índice de memória associados ao sP_z.

Axioma 5.3 .
O tempo de execução, em um trecho de código seqüencial, é igual ao somatório dos tempos de execução de suas instruções componentes.

Teorema 5.1 (Teorema da Equivalência) .
Seja $sP_z = \{(cy_k, energy_k)_k | i \leq k \leq j\}$ um segmento de Patch, o vetor instrução $\overrightarrow{MCPI_z^{Eq}}$ representa o modelo equivalente CPN-Probabilístico de instrução do sP_z, se e somente se,

$$\overrightarrow{MCPI_z^{Eq}} = (\sum_{k=i}^{j} cy_k, \sum_{k=i}^{j} energy_k)_z.$$

Prova 5.1 .
Seja um segmento de Patch $sP_z = \{(cy_k, energy_k)_k | i \leq k \leq j\}$, sP_z está associado a um trecho de código cujo consumo de energia é definido pela Equação 2.5 , que pode ser re-escrita como

$$E_{sP_z} = \sum_{i=0} (E_i + O_{i,i-1}) | (O_{i,-1} = 0).$$

Observando-se o parâmetro energy na Definição 5.18, tem-se

$$E_{sP_z} = \sum_{k=i}^{j} energy_k.$$

Observa-se que a contribuição de sP_z sobre o consumo de energia da aplicação é dada por $\sum_{k=i}^{j} energy_k$, enquanto sua contribuição sobre o tempo de execução é dada por $\sum_{k=i}^{j} cy_k$ (Axioma 5.3). Representando na forma de vetor instrução a contribuição de sP_z ao desempenho da aplicação, tem-se

$$\overrightarrow{D_{sP_z}} = (\sum_{k=i}^{j} cy_k, \sum_{k=i}^{j} energy_k)_z.$$

De acordo com o Axioma 5.1, cada elemento de sP_z estabelece uma contribuição no plano Energia×Ciclos-de-Clock, representada por um vetor instrução. Com base no Axioma 5.2, existe um vetor instrução resultante associado aos elementos de sP_z, tal que

$$\overrightarrow{Resultante_z} = \sum_{k=i}^{j} \overrightarrow{MCPIO_k^z} = (\sum_{k=i}^{j} cy_k, \sum_{k=i}^{j} energy_k)_z.$$

Por observação direta tem-se

$$\overrightarrow{Resultante_z} = \overrightarrow{D_{sP_z}}.$$

Esse vetor representa uma instrução hipotética que apresenta contribuição idêntica a do sP_z no plano Energia×Ciclos-de-Clock, ou seja, uma instrução equivalente a sP_z. Com base na Definição 5.18, esse vetor é definido como

$$\overrightarrow{MCPI_z^{Eq}} = (\sum_{k=i}^{j} cy_k, \sum_{k=i}^{j} energy_k)_z.$$

Com base no Teorema da Equivalência, o compilador Binário-CPN pode ser configurado para reduzir a rede aplicando três processos: (1) conversão de segmentos de *Patches* em $\overrightarrow{MCPI^{Eq}}$,(2) conversão de laços determinísticos[8] em $\overrightarrow{MCPI^{Eq}}$ de laço e (3) conversão de auto-laços determinísticos[9] em $\overrightarrow{MCPI^{Eq}}$ de auto-laço.

- **Processo 1.**
  Seja $c\_sP^n = \{sP_k\}$ o conjunto de segmentos de *Patch* de uma aplicação $ApCod_n$, o conjunto $\left\{\overrightarrow{MCPI_k^{Eq}}\right\}$ é gerado segundo:

  $$f_{Crit-red1} : c\_sP^n \rightarrow \left\{\overrightarrow{MCPI_k^{Eq}} | \#(sP_k) > 1 \wedge \forall sP_k^j \in sP_k \Rightarrow sP_k^j \in c\overrightarrow{MCPIO_n^{Ap}}\right\}.$$

[8]Desvio em que a instrução desvia para um endereço anterior uma quantidade fixa de vezes.
[9]Desvio em que a instrução desvia para si mesma uma quantidade fixa de vezes.

- **Processo 2.**
  Seja $MCPIC_n$ um $MCPI$ de desvio condicional, sendo a tupla $(e_{ent}, e_{said1}, e_{said2})_n$ a representação de seus endereços: endereço de instrução (e_{ent}) e endereços de destino (e_{said1} e e_{said2}). Seja $ModProb_n$, $Probfix_n$ e $Wloop_n$ parâmetros de sua AI. Seja ainda, $sP_z = \{(cy_k, energy_k)_k | i \leq k < j\}$ um segmento de $Patch$ e $\overrightarrow{MCPI^{Eq}_{(n,z)}}$ o $\overrightarrow{MCPI_{Eq}}$ do segmento de rede que contém sP_z e $MCPIC_n$, definido pela tupla $(MCPIC_n, sP_z)$. Seja ainda $Prd^{Loop}_{form}(n,z)$ uma função predicado que verifica a formação de laço devido à conjunção de $MCPIC_n$ e sP_z, sendo $Prd^{Loop}_{form}(n,z) = (((e_i = e_{said1}) \vee (e_i = e_{said2})) \wedge (e_j = e_{ent})) \wedge (ModProb_n \wedge (Probfix_n = 1.0))$. O conjunto $\left\{\overrightarrow{MCPI^{Eq}_{(n,z)}}\right\}$ de uma aplicação é gerado segundo a função

$$f_{Crit-red2} : \left\{(MCPIC_n, sP_z) | Prd^{Loop}_{form}(n,z)\right\} \rightarrow \left\{\overrightarrow{MCPI^{Eq}_{(n,z)}}\right\},$$

  na qual

$$\overrightarrow{MCPI^{Eq}_{(n,z)}} = Wloop_n \cdot \left(\overrightarrow{MCPI^{Eq}_{z}} + \overrightarrow{MCPIC_n}\right).$$

- **Processo 3.**
  Seja $MCPIC_n$ um $MCPI$ de desvio condicional, tal que a tupla $(e_{ent}, e_{said1}, e_{said2})_n$ representa seus endereços. O conjunto $\left\{\overrightarrow{MCPI^{Eq}_{n}}\right\}$ é gerado segundo a função

$$f_{Crit-red3} : \left\{MCPIC_n | Prd^{Auto-Loop}_{form}(n)\right\} \rightarrow \left\{\overrightarrow{MCPI^{Eq}_{n}}\right\},$$

  na qual

$$Prd^{Auto-Loop}_{form}(n) = ((e_{said1} = e_{ent} \vee e_{said2} = e_{ent}) \wedge (ModProb_n \wedge (Probfix_n = 1.0))$$

  e

$$\overrightarrow{MCPI^{Eq}_{n}} = Wloop_n \cdot \overrightarrow{MCPIC_n}$$

O preceito de todos os processos é a substituição das estruturas determinísticas do $MCPA$, particularmente os laços, por uma única transição - $MCPI$ equivalente -, estabelecendo reduções dos tempos de simulação. O Processo 1 compacta seqüências de instruções que não possuem instruções de desvio, daí a sua definição em termos de segmentos de *Patches*, uma vez que um *Patch* inteiro pode conter uma instrução de desvio como sua última instrução. A Figura 5.10 ilustra esse processo de redução. O Processo 2 compacta os laços determinísticos, substituindo a estrutura de laço por um $MCPI$ equivalente definido em termos do número de iterações modelada ($Wloop$), tal mecanismo está ilustrado na Figura 5.11. O Processo 3 compacta os auto-laços

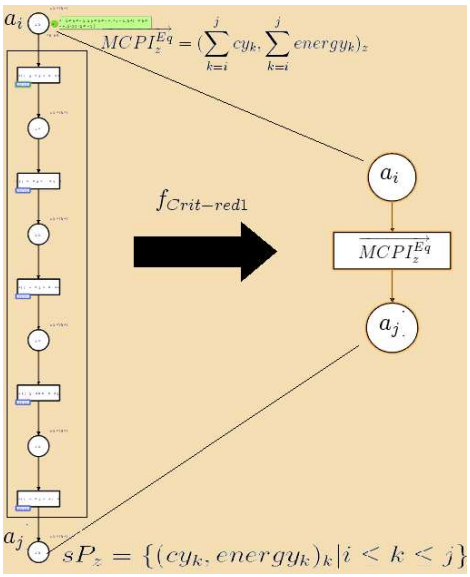

Figura 5.10: Redução da rede $MCPA$ em função dos segmentos de *Patches*.

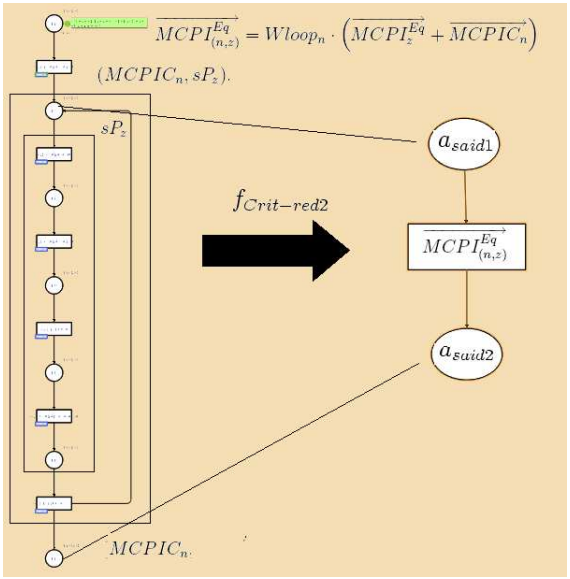

Figura 5.11: Redução da rede $MCPA$ em função dos laços determinísticos.

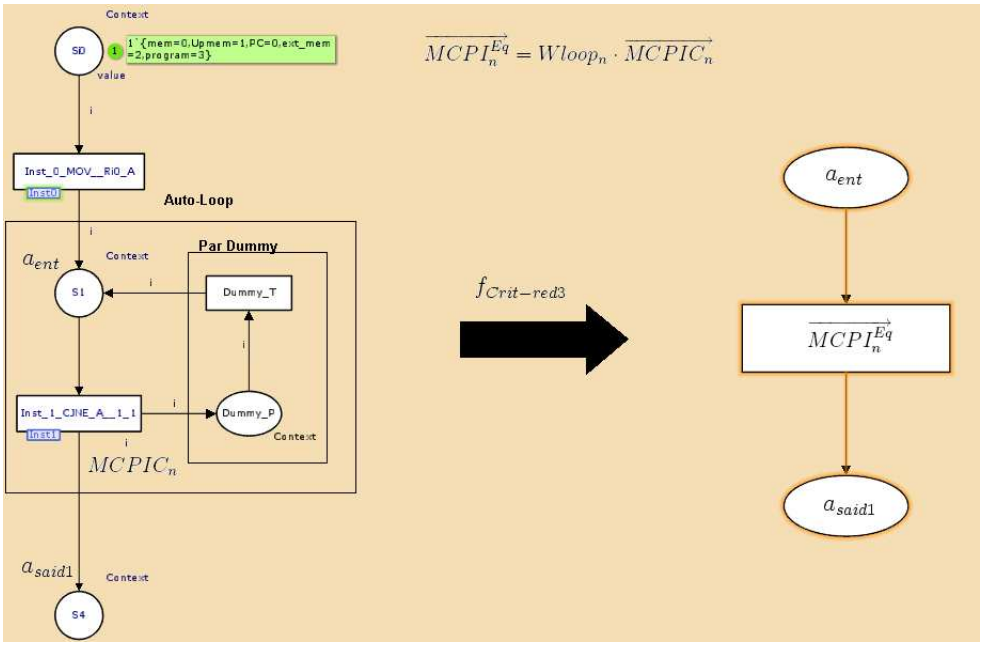

Figura 5.12: Redução da rede $MCPA$ em função dos auto-laços determinísticos.

de instrução, presentes no código para implementar atrasos intencionais na execução da aplicação, por exemplo: a espera de sinal de reconhecimento (*acknowlegde*) de um dispositivo periférico. Por estabelecerem auto-laços na rede, os compiladores, tanto o determinístico como o probabilístico no modo não reduzido, precisam inserir pares *Dummy* [58] para garantir a "corretude" da rede[10], o que faz com que um auto-laço

[10]Do contrário, o modelo não passa na verificação sintática do CPNTools.

tenha um custo de execução proporcionalmente dobrado[11]. A Figura 5.12 ilustra esse processo.

O mecanismo de redução deve ser executado em dois passos básicos: (i) identificação de segmento de *Patch* e implementação do Processo 1, e (ii) identificação de laços determinísticos e implementação dos Processos 2 e 3. Ao final da redução da rede $MCPA$, todos os elementos determinísticos estão compactados, de forma que a $MCPA$ compactada guarda as mesmas características comportamentais da não compactada. Embora tal mecanismo reduza muito o tempo de simulação, ele reduz o escopo de análise. Uma $MCPA$ reduzida elimina as instruções determinísticas como entidades de rede, impedindo o monitoramento estatístico das operações de *hardware* acessadas pelas instruções. Perde-se também a resolução de análise em termos de perfis de consumo e execução por instrução, tais perfis são agora gerados em termos dos $MCPI$ equivalentes de segmento ($MCPI_z^{Eq}$), de laço ($MCPI_{(n,z)}^{Eq}$) e de auto-laço ($MCPI_n^{Eq}$). Em contrapartida, isso permite a identificação automática de trechos candidatos para a migração *software-hardware*, caracterizados pelos $MCPI_z^{Eq}$, $MCPI_{(n,z)}^{Eq}$ e $MCPI_n^{Eq}$.

5.5 Mecanismos de Avaliação

O mecanismo de avaliação compreende a formatação dos dados gerados pela simulação do $MCPA$. Dada uma aplicação alvo, os dados gerados a partir da simulação probabilística de seu $MCPA$ são avaliados com base em três mecanismos básicos: (i) geração do histograma da distribuição de probabilidades dos valores das métricas, (ii) geração de estatísticas das métricas em função dos cenários gerados por instrução e (iii) geração de perfil de probabilidades em função dos *Patches*.

5.5.1 Distribuição de Probabilidades das Métricas

Dada a natureza probabilística do modelo proposto, não faz sentido buscar por um valor associado à métrica de interesse, deve-se buscar, no entanto, a probabilidade dessa métrica assumir uma determinada faixa de valores. O processo de avaliação da métrica segundo essa premissa compreende a representação da probabilidade da métrica de interesse assumir um valor dentro de determinado intervalo de valores, para um dado cenário de simulação. Para tanto, é definido um conjunto de intervalos de valores da métrica, dos quais se deseja estimar as probabilidades, em que cada intervalo é uma *classe* de valores a ser avaliada. Esse conjunto será, a partir da agora, denominado

[11] Duas transições são disparadas quando conceitualmente deveria ser uma.

conjunto de classes.

Definição 5.20 (Intervalo de Classe-$IntClass$) *:*
Seja $cClss_{Mt} = \{[MetrVal_n^{min}, MetrVal_n^{max}]_n | [MetrVal_n^{min}, MetrVal_n^{max}]_n \subseteq \mathbf{R}^+\}$ *o conjunto de intervalos de valores de interesse da métrica* Mt *(conjunto de classe). O valor* $IntClass_{Mt}$ *é dito intervalo de classe da métrica* Mt*, tal que* $IntClass_{Mt} = MetrVal_n^{max} - MetrVal_n^{min} = const, \forall [MetrVal_n^{min}, MetrVal_n^{max}]_n \in cClss_{Mt}$.

A distribuição de probabilidades das classes é calculada avaliando-se a freqüência de incidência dos valores, resultantes da simulação probabilística do cenário alvo, dentro do conjunto de classes. Seja CS um cenário de simulação, Mt uma métrica de interesse, $IntClass_{Mt}$ um intervalo de classe e $BagVal_{Mt}^{CS}$ o multiconjunto de valores capturados durante a simulação probabilística referente a CS, a distribuição de probabilidades de classe é calculada em duas etapas.

1. Determina-se um conjunto de classes em função do $IntClass_{Mt}$ e do $BagVal_{Mt}^{CS}$. Esse processo é definido pela função

$$F_{cClssConstr} : \{IntClass_{Mt}\} \times \left\{BagVal_{Mt}^{CS}\right\} \rightarrow cClss_{Mt}$$

2. Constrói-se o conjunto de pares probabilidade-classe

$$\left\{(Prob_k, Clss_k^{Mt}) | Clss_k^{Mt} \in cClss_{Mt}\right\}$$

através da função:

$$F_{dist} : cClss_{Mt} \times BagVal_{Mt}^{CS} \rightarrow \left\{(Prob_k, Clss_k^{Mt}) | Prob_k = \frac{\#(BagVal_{Mt}^{CS} \cap Clss_k^{Mt})}{\#BagVal_{Mt}^{CS}}\right\}$$

O conjunto de tuplas gerado por F_{dist} define a distribuição de probabilidades de ocorrência da métrica alvo dentro das classes, podendo ser visualizada em um histograma. A Figura 5.13 apresenta a distribuição de probabilidades gerada pela avaliação do consumo de energia de um filtro de convolução[12] operando no cenário $p(0024) = 0.8$, ou seja, a instrução localizada no endereço $0024H$ operando com probabilidade de desvio igual a 0.8. Cabe ressaltar que, dada a natureza freqüencista do conceito de probabilidade empregado, a exatidão dessa estimativa será tanto maior quanto maior for o número de amostras analisadas, ou seja, o valor do parâmetro Qs na anotação de cabeçalho (AC).

[12] A imagem representa uma interface de saída do protótipo de ambiente de análise desenvolvido, que será apresentado no Capítulo 6.

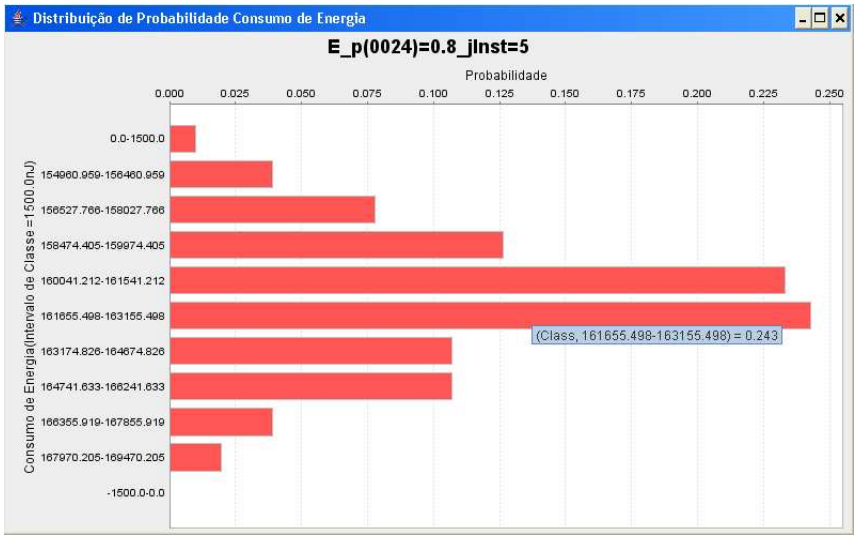

Figura 5.13: Exemplo de avaliação do consumo em termos da distribuição de probabilidades.

A avaliação também apresenta o valor médio e a dispersão (desvio padrão) dos valores da métrica alvo, em função do conjunto de cenário gerados por uma instrução de desvio. Tais estatísticas são geradas em tempo de simulação e armazenadas ao final de uma varredura de simulações probabilísticas. O ambiente de análise lê tais arquivos e constrói histogramas representativos da evolução do valor médio e do desvio padrão em função do cenário. A Figura 5.14 apresenta os resultados gerados pelo ambiente de análise com relação aos cenários gerados pela instrução situada no endereço $0024H$ do experimento *filtro de convolução*.

5.5.2 Perfil de Probabilidades dos *Patches*

Para um $MCPA$ instanciado sem reduções, o vetor de execução é gerado identicamente ao modelo determinístico. Sendo assim, a probabilidade da aplicação estar executando um *Patch ptch* em um momento aleatoriamente escolhido pode ser inferida como sendo:

$$Prob_{ptch} = \frac{\sum_{n=ip}^{fp} EV\left[i_n\right]}{\sum_{n=ia}^{fa} EV\left[i_n\right]}.$$

Em que ip e fp representam os índices do endereço inicial e final do *Patch* de interesse e ia e fa os índices dos endereços inicial e final da aplicação. Desta forma, é possível visualizar as probabilidades de acesso associadas às diferentes porções de memória ($Patch$). A avaliação de $Prob_{ptch}$ permite inferir melhor a distribuição de bancos de memória com gerenciamento de consumo, bem como, a estratégia de relocação de código como proposto em [11]. Enquanto no modelo determinístico a identificação de

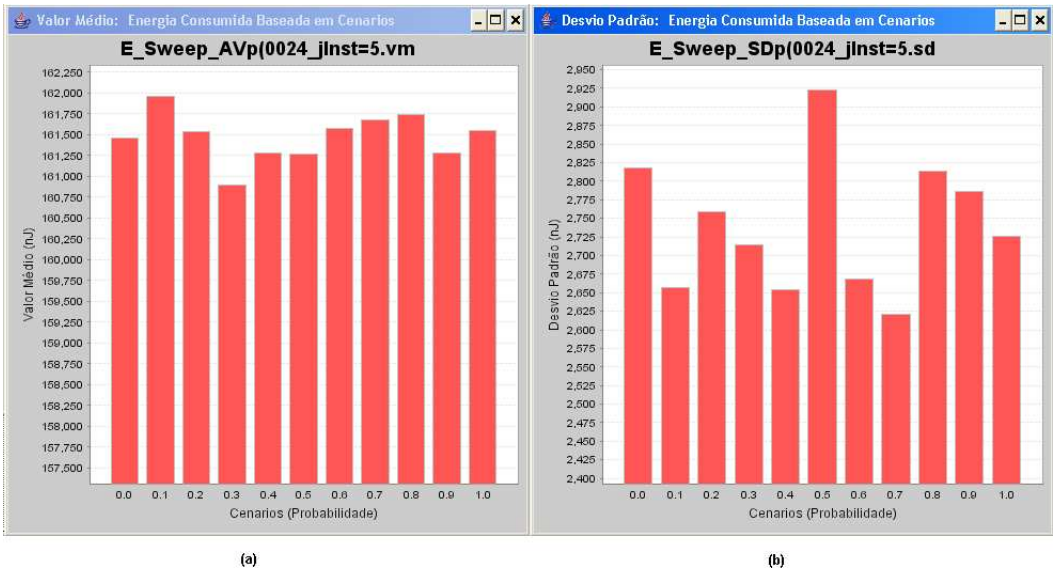

Figura 5.14: Exemplo de avaliação da métrica energia em função dos cenários: (a) valor médio, (b) desvio padrão.

Patches, *Clusters* e quantificação de seus efeitos é focada em uma condição pontual, no modelo probabilístico esse processo de identificação assume carácter mais genérico, estimando o comportamento do sistema em termos de cenários. O conceito de cenário permite uma visão centrada nos contextos de operação do sistema, o que pode resultar em uma análise mais adequada das políticas de otimização, sejam elas referentes ao particionamento *software-hardware*, ao escalonamento de bancos de memória, ou à relocação de trechos de código [11].

Os mecanismos de avaliação apresentados compreendem apenas um exemplo das possibilidades de análises do modelo probabilísticos. O arcabouço de modelagem e análise implementado, embora tenha sido desenvolvido para a arquitetura *i*8051, estabelece processos básicos para a avaliação de outras arquiteturas. Para arquiteturas *simplescalar*, serão necessários processos de redução da rede diferentes, uma vez que os atuais destroem o conceito de *pipeline* virtual. Não considerando o processo de redução, extensões *simplescalar* devem seguir os mesmos preceitos da extensão determinística proposta na Seção 7.2.1 como trabalho futuro.

5.6 Considerações Finais

Foi apresentada neste capítulo uma extensão do modelo CPN de descrição de arquiteturas (MCD), extensão essa centrada na modelagem do comportamento do *software* por meio das probabilidades de desvio do fluxo de execução, sendo denominada modelo

CPN-probabilistico de descrição ($MCPD$). Essa abordagem define o comportamento do *software* com base em eventos do sistemas, por meio de dois níveis hierárquicos construídos como categorias de eventos: eventos primários e secundários. Os eventos primários identificam uma ação macro no sistema, podendo ter sentido abstrato. Um evento primário inclui um conjunto de ações primárias, definidas com fluxos de execução no código. Cada ação primária, por sua vez, pode ser caracterizada por um conjunto de eventos de desvio no fluxo de execução do código, sendo esses definidos como eventos secundários. O modelo CPN-Probabilístico de descrição descreve uma aplicação de *software* segundo o conjunto de probabilidades associado à ocorrência dos eventos secundários da aplicação. Para tal, o modelo CPN de instrução é estendido de forma a comportar uma descrição da probabilidade de desvio associada à instrução. Sob essa óptica, o modelo classifica dois tipos básicos de instruções: as determinísticas e as probabilísticas. São classificadas como determinísticas as instruções que guardam comportamento uniforme quanto à execução de um desvio, seja sempre executando-o, como as instruções de desvio incondicional, seja nunca o fazendo, como as instrução que não são de desvio. As instruções classificadas como probabilísticas são aquelas cuja ação de desviar pode ser descrita em termos de probabilidades inferidas a partir de uma percepção do cenário de operação do sistema, cuja ação de desviar (evento secundário) está associada, direta ou indiretamente, a um evento primário. Nesse contexto, as instruções de desvio condicional são por natureza instruções probabilísticas. Com base nesses preceitos, modelos CPN-Probabilísticos de instrução ($MCPI$) foram criados para a arquitetura $i8051$, de forma a prover a geração de $MCPAs$, mantendo o mesmo formalismo de construção proposto no modelo CPN determinístico de descrição.

Identicamente ao MCA, o $MCPA$ é criado diretamente do código fonte *assembly* da aplicação. Dado o $MCPA$, ele é simulado segundo a técnica Monte Carlo, que compreende simular o $MCPA$ ciclicamente até que a avaliação das estatísticas dos resultados parciais mostrem que o valor médio estimado está dentro de um erro estipulado e de acordo com um dado nível de confiança. Tal simulação cíclica foi denominda *simulação probabilística*, por descrever melhor a natureza aleatória do processo de simulação. Diferentemente de outras aplicações da técnica de Monte Carlo para estimativa de consumo de energia, como em Burch *et al* [16][13], o comportamento da simulação é determinada em tempo de simulação com base em um modelo de inferência das probabilidades de ocorrência dos eventos secundários na aplicação, ou seja, um modelo probabilístico de comportamento do sistema. O modelo de compor-

[13]No trabalho de Burch *et al* o vetor de teste é gerado aleatóriamente sem maiores restrições, podendo em tese alimentar a simulação com valores inverossímeis ao sistema.

tamento probabilístico e os parâmetros de simulação da aplicação são inseridos por meio de anotações, definindo duas categorias sintáticas: as anotações de instrução e as anotações de cabeçalho. A anotações de instruções (AI) são inseridas em todas as instruções probabilísticas e modelam o comportamento probabilístico da instrução (evento secundário). A anotação de cabeçalho (AC) é um conjunto de parâmetros de simulação do $MCPA$, definindo fatores como erro máximo e nível de confiança, existindo apenas uma AC por código. O processo pelo qual se captura as percepções abstratas do comportamento do sistema, convertendo-as em anotações de instruções, foi denominado *modelagem probabilística*. No presente trabalho, a *modelagem probabilística* foi tratada de forma *ad hoc*, a sistematização da *modelagem probabilística* compreende um desafio a parte.

Um ambiente de modelagem, simulação e análise foi implementado permitindo a automatização da geração de $MCPAs$ e a análise da massa de dados gerada pela simulação probabilística. Como resultado da simulação probabilística do $MCPA$, as métricas são avaliadas de forma estatística. O valor médio das métricas é interpretado *a priori*, como o valor de maior probabilidade dentro do cenário de operação proposto. Para cada instrução probabilística, o ambiente de análise gera um histograma representativo do valor médio e do desvio padrão da métrica, com o fim de estabelecer uma visão específica, cenário a cenário. O modelo de análise representa o conjunto de valores de métricas capturado em classes de valores, com base nessa representação constroem-se histogramas da distribuição de probabilidades de ocorrências dessas classes no cenário de interesse. Em outras palavras, dada uma métrica e um cenário de interesse, o ambiente de análise informa a probabilidade do valor da métrica encontrar-se em determinados intervalos (classes). Adicionalmente, os *Patches* podem ser identificados em função das suas probabilidades de execução, estabelecendo um recurso para a escolha de trechos candidatos à migração *software-hardware*.

O $MCPD$ implementa os preceitos lançados por Burch *et al* em [16], originalmente aplicados à estimativa em *hardware*, focando a avaliação da aplicação de *software*. A metodologia proposta estabelece as seguintes contribuições: (i) aplicação específica para processadores, (ii) avaliação centrada no modelo de consumo do *software*, (iii) exploração dos cenários de execução por fluxos de execução aleatórios parametrizáveis a partir de inferências de probabilidade dos eventos do sistema, (iv) capacidade de investigação de cenários específicos de execução, (v) formalismo decorrente de ser um MCD e (vi) grande reconfigurabilidade quanto a arquitetura alvo.

Capítulo 6

Protótipos e Experimentos

6.1 Introdução

Com o objetivo de validar os modelos propostos, foram implementados dois protótipos de ambiente de análise: um referente ao modelo determinístico e outro ao modelo probabilístico. A análise determinística foi integrada ao ambiente EZPetri, gerando o EZPetri-PCAF (*Power Cost Analysis Framework*). A análise probabilística foi implementada como uma aplicação independente, podendo ser integrada futuramente. Foram realizados experimentos para validar os elementos de modelagem e os ambientes de análise propostos. Nos experimentos, os ambientes de análise foram usados para explorar os padrões comportamentais do código. Para o modelo determinístico, foi realizada a avaliação do conjunto de otimizações de código disponível em um compilador C comercial. O experimento avaliou os perfis de consumo de códigos binários gerados a partir de um mesmo código fonte, compilado sob diferentes opções de otimização. Com base nos resultados da análise, o projetista pode determinar qual otimização apresenta adequação às restrições de consumo de sua aplicação, bem como identificar regiões das memórias de código e de dados cuja implementação em *hardware* pode ser otimizada. Com relação ao modelo probabilístico, foram realizados experimentos que ilustram casos típicos de códigos presentes em sistemas embutidos. Em todos eles, a infra-estrutura de análise foi utilizada para a estimativa da evolução dos valores das métricas (consumo de energia, potência e tempo de execução) em função dos cenários sob análise. Os resultados são apresentados de duas formas básicas: através dos valores médios para cada cenário e por meio das distribuições de probabilidades. As distribuições de probabilidades são representadas como histogramas que descrevem a probabilidade da métrica em questão estar em um dado intervalo de valores (classe), quando o sistema está operando em um dado cenário. Com base nesses resultados, o

projetista pode estabelecer uma visão ampla dos padrões de consumo de energia e dos tempos de execução, sem a utilização de vetores de simulação.

6.2 Análise Determinística: Ambiente EZPetri-PCAF

O EZPetri é um ambiente de redes de Petri desenvolvido pelo Departamento de Sistemas Computacionais da Escola Politécnica da Universidade de Pernambuco (DSC-UPE). O EZPetri é *plugin* Eclipse e suporta recursos de análise, seja usando recursos próprios, seja por operação como *front-end* de outra ferramenta. Portanto, o ambiente EZPetri pode encapsular ferramentas inteiras estabelecendo uma interface apropriada para determinados contextos de análise. A representação interna da rede é feita em PNML (*Petri Net Markup Language*), por meio de conversões do padrão PNML usando compiladores "plugáveis". O EZPetri possibilita a exportação/importação de redes, virtualmente, de quaisquer formatos.

A integração do modelo determinístico com o EZPetri proporciona uma *interface* adequada à execução do fluxo de análise, gerando um ambiente específico para análise de código denominado EZPetri-PCAF. O EZPetri-PCAF incorpora o compilador Binário-CPN, já descrito no Capítulo 4, módulos de análise e formatadores de resultados. Os módulos de análise são responsáveis pela identificação das entidades definidas na taxonomia proposta. Os formatadores de resultados são módulos que geram tabelas e gráficos para apresentação dos resultados ao usuário. O processo de avaliação de um código consiste em três passos básicos: (i) a partir do EZPetri, compila-se o código binário gerando-se seu MCA, (ii) no ambiente CPNTools, simula-se o modelo gerado e (iii) processa-se os resultados brutos da simulação por meio do módulo de análise do EZPetri. Adicionalmente, todos os elementos de análise (tabelas e gráficos) podem ser salvos em um documento no formato pdf (*portable document file*). A próxima seção apresenta um experimento onde o EZPetri-PCAF foi aplicado para explorar opções de otimização de código em compiladores visando otimização de consumo.

6.3 Experimento 1: Explorando Opções de Compilação

6.3.1 Descrição

O objetivo deste experimento consiste na aplicação da metodologia de modelagem proposta para a exploração das opções de otimização de compiladores, gerando subsídios para a otimização do consumo, seja pela simples escolha do melhor código binário; pela modificação do núcleo do processador, para otimizar uma dada implementação de código, ou pela migração *software-hardware*. O compilador escolhido foi o compilador C Keil-C51v7, por amplamente usado na comunidade de *softwares* embutidos. O compilador Keil-C51v7 possui dez níveis de otimização em duas ênfases: extensão de código e tempo de execução, totalizando vinte opções de implementação de código binário. Foram escolhidos três níveis de otimização, aplicando-se as duas ênfases, gerando com isso seis códigos de teste. Como código fonte foi utilizada uma função de ordenamento configurada para vetores de dez posições. A implementação desta função foi realiza aplicando o algoritmo de ordenamento de bolha. Na Tabela 6.1 são relacionados os seis códigos binários gerados e as opções de otimização que lhes deram origem. Descrevendo sucintamente essas técnicas tem-se: *Commom Block Subroutines*, transforma estruturas recorrentes de código em subrotinas; *Constant Folding*, pré-avalia expressões transformando-as em constantes quando possível; e *Register Variables*, aloca as variáveis mais comuns em registradores.

Tabela 6.1: Códigos de teste e otimizações avaliadas

Implementação	Otimização Utilizada
sort_fs_cb	*Commom Block Subroutines (Favour Size)*
sort_fs_cf	*Constant Folding (Favour Size)*
sort_fs_rv	*Register Variables (Favour Size)*
sort_fsp_cb	*Commom Block Subroutines (Favour Speed)*
sort_fsp_cf	*Constant Folding (Favour Speed)*
sort_fsp_rv	*Register Variables (Favour Speed)*

A Figura 6.1 ilustra a seqüência de passos do experimento. Os códigos binários são compilados para seu modelo CPN e simulados no CPNTools. O CPNTools gera como saída arquivos ASCII com todas as sondas habilitadas no modelo. Esses arquivos são os resultados brutos da simulação, resultados esses que são tratados pelo módulo de análise

do EZPetri-PCAF, sendo apresentados como tabelas de *Patches* e perfis de execução e consumo. Adicionalmente, são gerados gráficos representativos da distribuição dos *Patches* na memória de código e perfis de acesso, escrita e leitura, na memória de código e de dados.

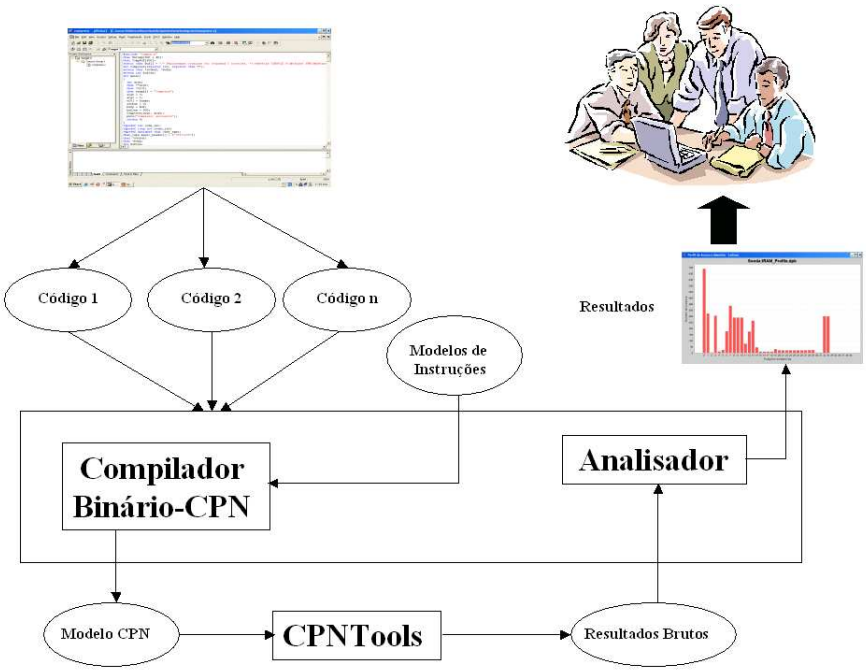

Figura 6.1: Explorando opções de compilação: descrição do Experimento 1

6.3.2 Resultados

As Figuras 6.2, 6.3 e 6.4 apresentam as saídas gráficas geradas para os perfis de consumo de energia das implementações tipo fs (vide Tabela 6.1). Todos os gráficos podem ser gerados objetivando uma visão ampla, como nas figuras citadas, ou analisando setores específicos, como mostra a Figura 6.5. Nota-se que pela avaliação do perfil de todo o código é possível visualizar a distribuição de consumo ao longo do código, identificando a melhor estratégia a ser implementada. O processo de identificação e estimativa de consumo de entidades, como *Loop-Patches* e *Clusters*, dá suporte direto: (i) à aplicação de técnicas de mapeamento de segmentos de código em memória interna [11], (ii) à estratégias de segmentação da memória em bancos otimizados para a aplicação, permitindo a implementação de gerenciamento de consumo, e (iii) à migração *software-hardware*[92].

As Figuras 6.6 e 6.7 apresentam as saídas gráficas geradas para a análise do perfil de acesso de leitura e escrita da memória de dados interna, para a implementação

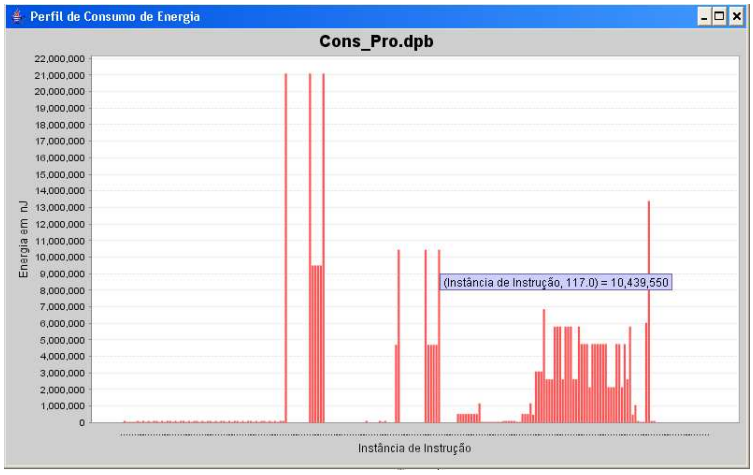

Figura 6.2: Perfil de consumo da implementação sort_fs_rv

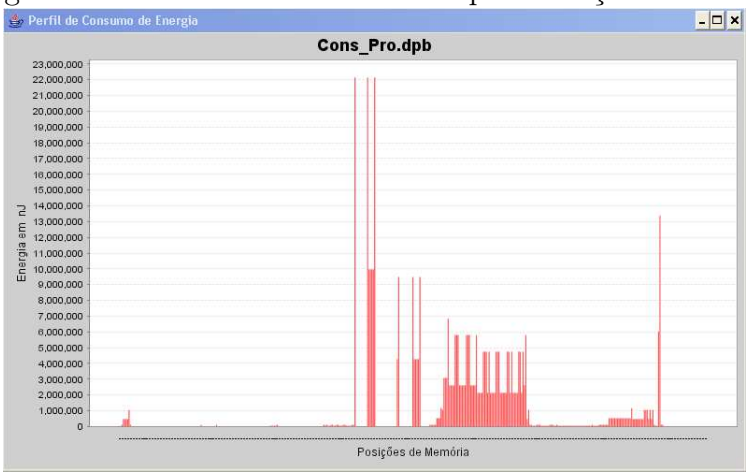

Figura 6.3: Perfil de execução implementação sort_fsp_cb

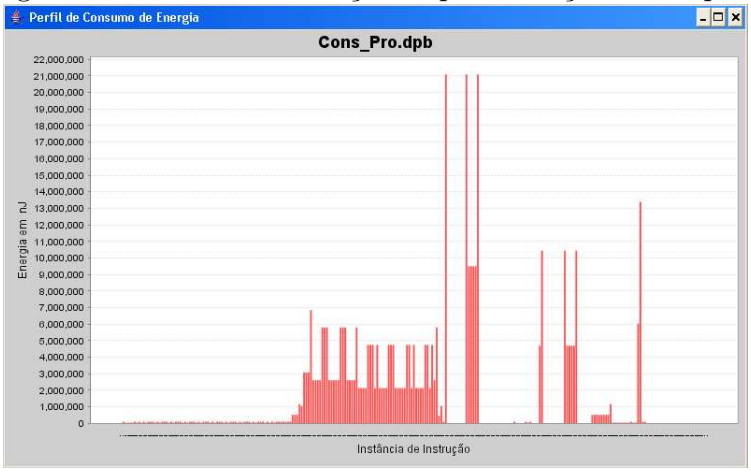

Figura 6.4: Perfil de execução implementação sort_fs_cf

sort_fs_cb. Esses perfis de acesso permitem a otimização do núcleo (*core*) quanto a: (i) segmentação da memória em bancos otimizados para a aplicação, (ii) re-avaliação

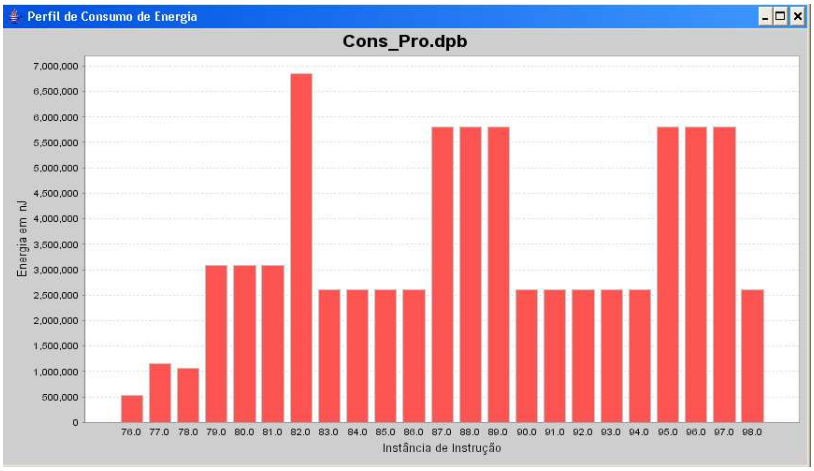

Figura 6.5: Explorando Opções de Compilação: Descrição do experimento

de roteamento de barramentos com base na taxa de acesso a posições específicas, (iii) escolha de políticas de *Cache*. A implementação do código pode então ser escolhida com base em seu perfil de acesso, e na melhor relação de implementação dessas técnicas.

No contexto da exploração das opções de otimização, a melhor implementação pode ser escolhida pela análise comparativa do consumo total de energia. A Tabela 6.2 apresenta, para cada implementação: (i) o consumo de energia, (ii) a potência média, (iii) a economia de energia em relação ao pior consumo (sort_fs_cf) e (iv) o tempo de execução. Dessa maneira, identifica-se a implementação de menor consumo (sort_fsp_cb), que oferece uma economia de 26% em relação à de maior consumo. Observa-se que a pequena variação da potência média está associada ao alto nível de correlação entre consumo de energia e tempo de execução das instruções do micro-controlador AT89S8252, conforme padrão identificado na caracterização do dispositivo (vide Apêndice A). Nota-se ainda que a ênfase de otimização (extensão de código ou tempo de execução) teve impacto mínimo sobre o consumo de energia e o tempo de execução. Neste experimento, o padrão de consumo e tempo de execução mostram-se fortemente relacionados com o nível de otimização.

No contexto da análise para migração *software-hardware*, o segmento de código a ser migrado pode ser escolhido identificando-se o *Patch* de maior consumo em relação ao consumo total (HCP-*Highest Consumption Patch*). Neste experimento, dada a sua dimensão, a avaliação foi centrada na identificação de *Patches*. Em códigos mais extensos idealmente esse processo deve ser dirigido aos *Clusters*. A Figura 6.8 apresenta as contribuições percentuais dos HCPs para cada implementação. Observando-se essa figura, nota-se que a implementação sort_fsp_cb parece oferecer o maior ganho na migração *software-hardware*, uma vez que é a que mais concentra o consumo de energia

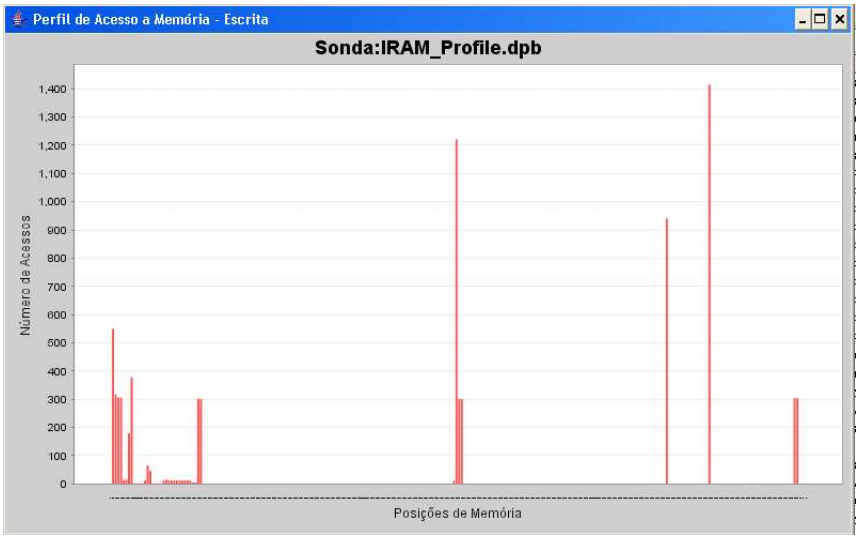

Figura 6.6: Perfil de acesso à memória de dados interna. Acesso de escrita da implementação sort_fs_cb

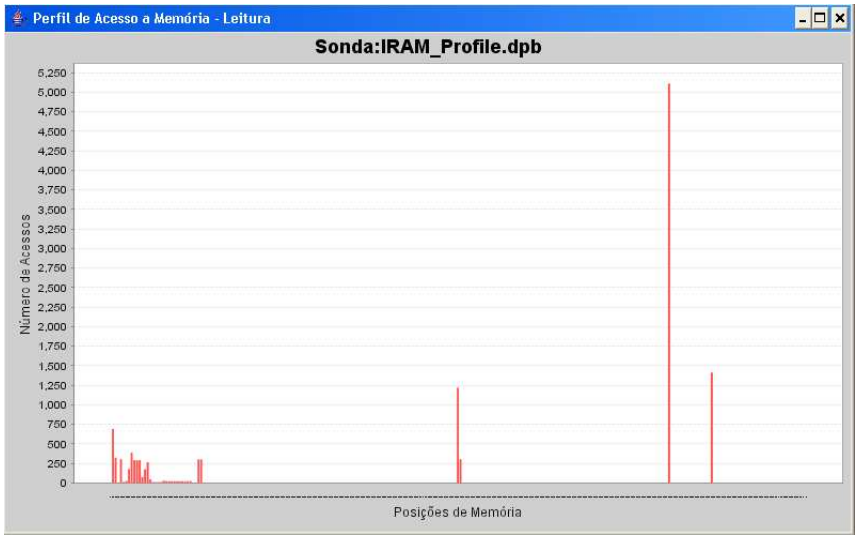

Figura 6.7: Perfil de acesso à memória de dados interna. Acesso de leitura da implementação sort_fs_cb

em seu HCP (79%). Observa-se contudo, que a implementação sort_fs_cb é apenas marginalmente inferior nesse critério (78,82%), mas apresenta o melhor consumo global, o que a qualifica como melhor implementação também considerando migração *software-hardware*.

Tabela 6.2: Padrão de consumo

Implementação	Consumo de Energia (uJ)	Economia de Energia(%)	Potência Média (mW)	Tempo de Execução (ms)
sort_fs_rv	340,48	14,28	50,05	6,669
sort_fs_cf	397,18	0	50,42	7,877
sort_fs_cb	296,42	25,37	50,89	5,825
sort_fsp_rv	337,84	14,94	50,84	6,645
sort_fsp_cf	392,86	1,09	50,18	7,829
sort_fsp_cb	293,35	26,14	50,45	5,815

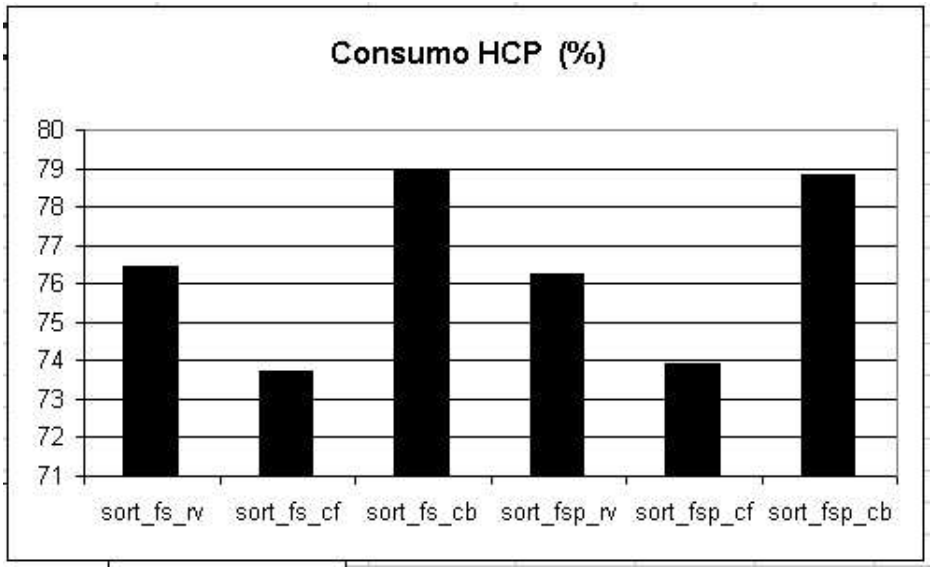

Figura 6.8: Consumo relativo dos *Patches* de consumo (HCP)

6.4 Ambiente de Análise Probabilística

Para a avaliação probabilística, os passos de modelagem e análise apresentados no Capítulo 5 foram integrados em uma ferramenta protótipo, permitindo: (i) a inserção das anotações no código fonte *assembly*, (ii) a geração do código binário, (iii) a geração do arquivo XML-Prob e (iv) a avaliação dos resultados. Para a avaliação dos resultados o protótipo oferece quatro recursos básicos:

- geração, por cenário, de histograma da distribuição de probabilidade associadas as métricas;
- geração de histograma representativo da evolução do valor médio da métrica, devido à cada cenário gerado por uma dada instrução de desvio;

- geração do campo escalar[1] que descreve a distribuição de probabilidades das classes de valores da métrica em função dos cenários, para uma dada instrução ou conjugando todas as instruções; e
- geração de histogramas representativos do perfil da probabilidade de execução de instâncias de instrução e *Patches*.

O protótipo foi desenvolvido em Java utilizando a API JFreeChart para a representação dos gráficos. Com isso, estão disponíveis recursos como aproximação (*zoom*), geração de arquivos de imagem (jpg e pgn) e apresentação de valores pontuais dos gráficos (*tooltips*). A Figuras 6.9 apresenta a tela principal do protótipo durante a seleção de recursos de análise. A aplicação desses recursos ficará clara durante a discussão dos próximos experimentos. Para todos os experimentos seguintes, a anotação de cabeçalho (AC) foi configurada para o nível de confiança de 90%, erro máximo de 5%, efetuando no mínimo 1000 simulações. A aplicação de 1000 simulações foi empregada de forma a prover um espaço amostral significativo à construção de curvas de distribuição de probabilidades. A potência foi usada como a métrica a ser avaliada no critério de parada.

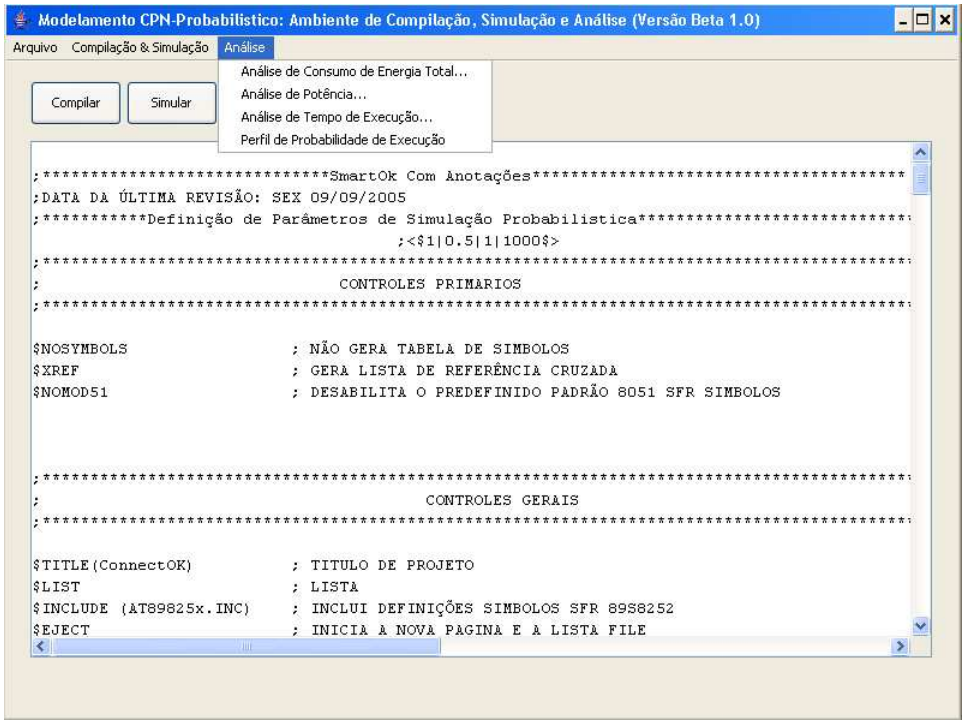

Figura 6.9: Análise probabilística: recursos de análise

[1]Representação em três dimensões de uma função de múltiplas variáveis.

6.5 Experimento 2: Ordenamento em Bolha

6.5.1 Descrição

Este experimento teve como objetivo validar o modelo probabilístico. Um código de ordenamento em bolha foi utilizado devido à facilidade de se prever analiticamente o comportamento do código, particularmente na determinação do melhor e do pior tempo de execução. Na Figura 6.10 é apresentado o código com as anotações de modelagem e parametrização da simulação probabilística (*AI* e *AC*). A modelagem probabilística foi implementada *ad hoc* de acordo com as considerações discutidas em seguida.

```
1 ;**********Ordenamento de Bolha - Avaliação do Modelo Probabilistico************
2 ;****************** Cabeçalho (Header-Annotation) *********************************
3                          ;<$1|0.05|1|1000$>
4 begin:
5      mov r4,#00H
6 loop:
7      mov a,r4
8      add a,#30H
9      mov r4,a
10     lcall swaap
11     mov a,r4
12     subb a,#30H
13     mov r4,a
14     inc r4
15     mov a,r4                ; Anotação de Instrução (Instruction-Annotation)
16     cjne a,#9H,loop         ;<@[0.0,1.0,0.1]|true|1.0|9@>
17     ret                     ; Ponto de saída do código
18 swaap:
19     mov 00H,r4
20     mov r1,00H
21 inner_loop:
22     clr c
23     inc r1
24     mov a,@r0
25     subb a,@r1              ;Anotação de Instrução (Instruction-Annotation)
26     jc exit                 ;<@[0.0,1.0,0.1]|false|1.0|256000@>
27     inc r5
28     mov b,@r0
29     mov a,@r1
30     mov @r0,a
31     mov @r1,b
32 exit:
33     mov a,r1                ;Anotação de Instrução (Instruction-Annotation)
34     cjne a,#39H,inner_loop  ;<@[1.0,1.0,0.1]|true|1.0|5@>
35     ret
```

Figura 6.10: Código *assembly* com anotações, experimento 2

Observa-se que todas as variações do fluxo de programa são regidas por três instruções de desvio, nas linhas 16 (*inst16*), 26 (*inst26*) e 34 (*inst34*). As instruções *inst16* e *inst34* têm comportamento regular, estabelecem laços cujo número de iterações dependem do comprimento do vetor (*comp_vetor*) a ser ordenado, desenvolvendo um comportamento determinístico. Por outro lado, a instrução *inst26* apresenta um comportamento probabilístico, o desvio ocorrerá ou não dependendo do nível de ordenamento do vetor. Analisando o algoritmo de ordenamento em bolha, verifica-se que: (i) a instrução *inst16* sempre desvia até completar $comp\_vetor - 1$ iterações, (ii) para cada desvio de *inst16*, identificado por um índice n, n variando de 1 a $comp\_vetor - 1$, a instrução *inst34* desvia um número b_n de vezes, e (iii) b_n é definido pela seguinte

função de progressão aritmética:

$$b_n = (comp\_vetor - 1) + (n - 1) * (-1).$$

Dessa forma, o total de desvios realizados é dado por:

$$T_{desvios}^{inst34} = \sum_n b_n = \frac{(b_1 + b_{(comp\_vetor-1)}) * (comp-vetor-1)}{2}.$$

Para um vetor de dez posições, $T_{desvios}^{inst34} = 45$. Observa-se ainda, que *inst34* desvia 45 vezes enquanto *inst16* desvia 9 vezes. Uma vez que o modelo opera explorando os fluxos de execução do programa e não reproduzindo fielmente seu comportamento, o limitador de iterações ($Wloop$) deve ser representado pelo valor médio de desvios executados por *inst34* em relação a um desvio de *inst16*, ou seja 5. Sendo assim, *inst34* e *inst16* devem ser modeladas como instruções determinísticas ($Probfix = 1.0$ e $ModProb = true$) limitadas em 5 e 9 desvios consecutivos, respectivamente.

A partir dessa análise, conclui-se que toda a dinâmica comportamental do código pode ser descrita em termos da probabilidade de *inst26* desviar, uma vez que este parâmetro está diretamente ligado ao padrão de ordenamento do vetor. Observado que *inst26* tem seu número de iterações limitado por *inst34*, devido à estrutura do código, o limitador de iterações de *inst26* pode ter valor arbitrário, 256000 nesse experimento. Observa-se ainda que o pior e melhor tempo de execução, e melhor consumo ocorrem, respectivamente, quando a probabilidade de *inst26* desviar for igual a 0,0 ou a 1,0. Neste experimento, foi realizada a avaliação no modo varredura com passo de 0,1 em todas as instruções ($st = 0, 1$), cobrindo assim onze cenários. Adicionalmente, foram realizadas medidas em *hardware* de forma a verificar a consistência das estimativas geradas pelo modelo. A Figura 6.11 ilustra as etapas seguidas no experimento.

Como primeira etapa, 1000 códigos de teste foram gerados a partir de um código original (semente). Cada código de teste compreende o código original ordenando um vetor diferente, gerado com valores e ordens aleatórios. Em seguida, uma versão modificada do sistema de caracterização automática (vide Apêndice A) efetua as medidas para todos os códigos de teste. Identicamente ao processo de caracterização, as medidas são armazenadas em arquivos XML individuais para cada código de teste. Um módulo integrador de resultados processa esses arquivos gerando como saída a distribuição de probabilidades referente a cada métrica medida (tempo de execução, energia consumida e potência). O gerador de código de teste fornece, como saída adicional, um arquivo com todos os vetores gerados. Esse arquivo é processado por um módulo identificador de padrões. O identificador de padrões executa o algoritmo de ordenamento de forma a determinar o valor médio da freqüência relativa de permutação de elementos, bem como

a distribuição de probabilidades dessa freqüência relativa. Essa freqüência relativa representa a probabilidade de *inst26* desviar. Na Figura 6.12 é apresentada a distribuição de probabilidades dessa probabilidade. Exemplificando: observa-se que a probabilidade de *inst26* operar com uma probabilidade de desvio dentro do intervalo [0,4888889 ... 0,4988888] é 0,081. Paralelamente, a simulação probabilística do $MCPA$ do código foi realizada assumindo a probabilidade de permutação mais provável, extraída da distribuição apresentada na Figura 6.12. Dessa forma, o experimento permitiu, além da validação do modelo, a análise de sua exatidão frente a um exemplo real no qual os eventos não possuem probabilidades constantes.

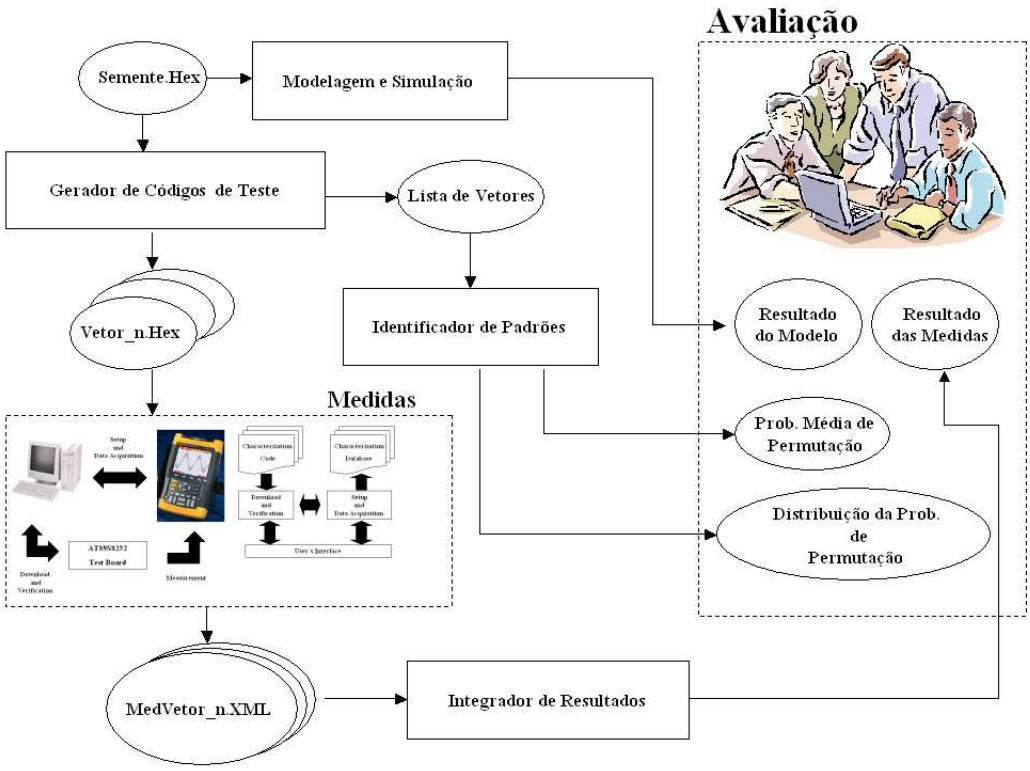

Figura 6.11: Diagrama de etapas do Experimento 2

6.5.2 Resultados

A Figura 6.13 mostra a composição de cinco telas geradas pelo protótipo durante a análise do código. Na tela (a) é apresentada a distribuição de probabilidades do tempo de execução para P(*inst26*)=1,0. Conforme exposto, isso corresponde ao melhor caso do tempo de execução. Nas telas (b), (c) e (d) são apresentadas as distribuições para P(*inst26*)=0,9 , P(*inst26*)=0,5 e P(*inst26*)=0,1 , respectivamente. Na tela (e) é apresentada a distribuição para P(*inst26*)=0,0, correspondente ao pior caso. Em todas as

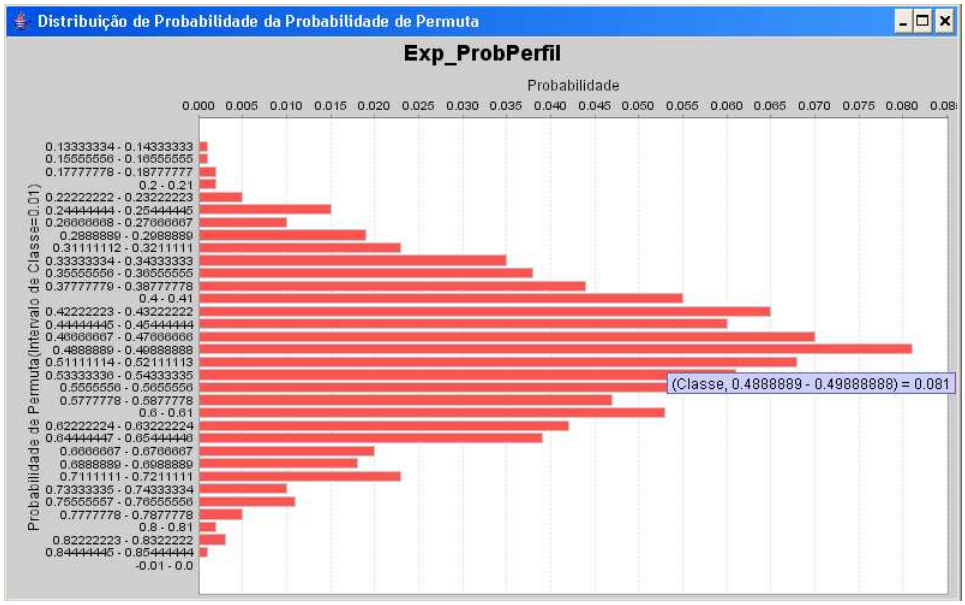

Figura 6.12: Distribuição de probabilidade da probabilidade de permutação de elementos do vetor durante o ordenamento

telas o intervalo de classe é de 1μs, sendo esse o *quantum* de tempo de execução de instrução[2]. Como esperado, P(*inst26*)=1,0 e P(*inst26*)=0,0 definiram perfeitamente o comportamento determinístico que o código assume quando o vetor está ordenado, tanto em ordem direta como em ordem inversa. Na Figura 6.14 é apresentada a tela de saída para a avaliação da energia consumida no cenário P(*inst26*)=0,5. As Figuras 6.15, 6.16 e 6.17 apresentam o padrão de evolução das métricas em função dos cenários avaliados, em termos de valor médio e desvio padrão. A avaliação de tais padrões permite a identificação dos cenários críticos da aplicação. Adicionalmente, o protótipo fornece uma visualização do campo escalar da distribuição de probabilidade versus cenários, possibilitando uma visão panorâmica do comportamento estatístico das métricas em função do cenário (vide Figura 6.18). Os experimentos feitos em *hardware* reproduziram com exatidão o comportamento previsto pelo modelo. A Tabela 6.3 relaciona os valores estimados e medidos bem como os erros de estimativa.

Tabela 6.3: Estimativas de melhor e pior caso de consumo e tempo de execução

	Estimado	Estimado	Medido	Medido	Erro(%)	Erro(%)
Métrica	Pior-Caso	Melhor-Caso	Pior-Caso	Melhor-Case	Pior-Case	Melhor-Case
Potência(mW)	51,421	49,765	52,878	52,419	2,8	5,0
Energia Total(uJ)	42,731	27,968	46,046	29,980	7,2	6,7
Tempo de Execução(us)	831	562	831	562	0	0

[2]Micro-controlador AT89S8252 operando com fereqüência de relógio igual a 12MHz.

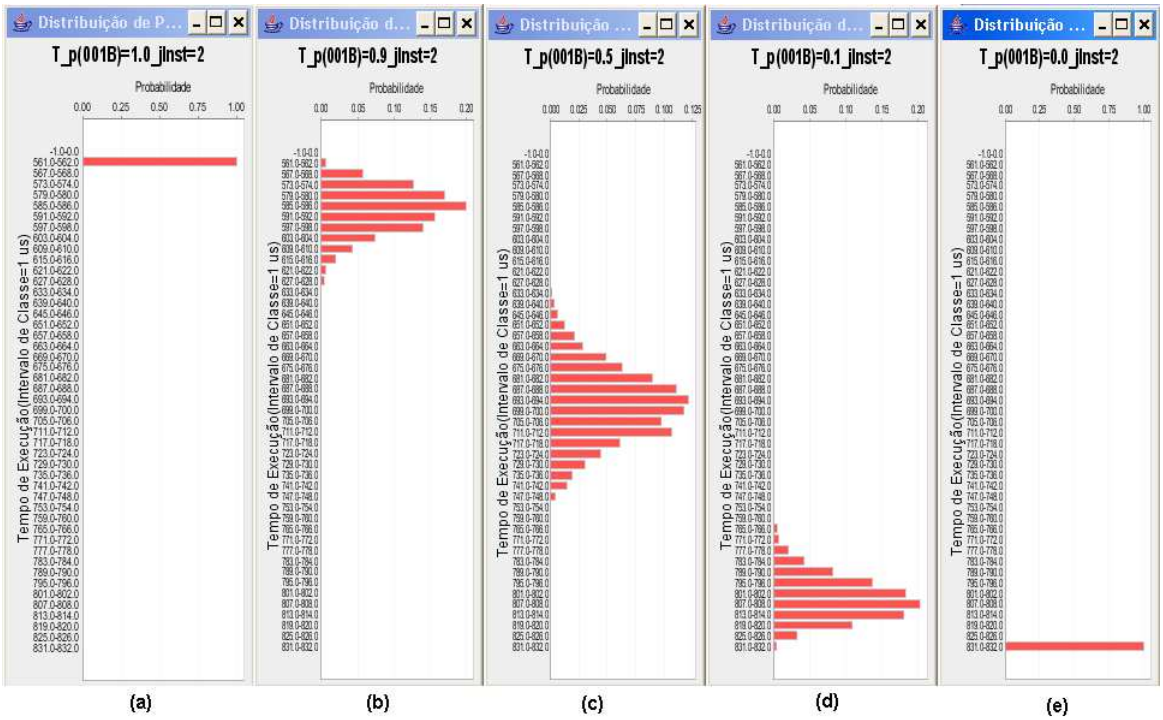

Figura 6.13: Distribuição de probabilidade de tempo de execução para cinco cenários consecutivos.

A verificação de consistência com a execução em *hardware* foi feita comparando-se a distribuição de probabilidades do tempo de execução em *hardware* com a estimativa de distribuição gerada pelo modelo para P(*inst26*)=0,49511, sendo esse o valor médio da freqüência relativa de permutação de elementos fornecido pelo identificador de padrões (vide Figura 6.11). Na Figura 6.19 observa-se que a curva do experimento de *hardware* apresenta maior dispersão do que a gerada pelo modelo. Isso é explicado pelo fato do experimento de *hardware* reproduzir condições nas quais a probabilidade do evento de permutação não é constante (vide Figura 6.12), enquanto o modelo opera com probabilidade constante.

A diferença de dispersão das curvas informa apenas que o valor numérico da probabilidade não pode ser estimado quando se modela um conjunto de cenários (probabilidades de eventos) por um cenário médio (probabilidade média). A classe de maior probabilidade, ou seja o intervalo de valores mais prováveis da métrica, pode ser identificada com grande exatidão. O tempo de execução mais provável apresentado pelo experimento foi de 701μs, e o modelo determinou 694μs, perfazendo um erro de 0,99%. Cabe ressaltar ainda que o experimento reproduziu uma situação extrema, na qual os elementos dos vetores são aleatórios. Em muitas aplicações de sistemas embutidos, algoritmos de ordenamento são normalmente aplicáveis no refinamento de processos

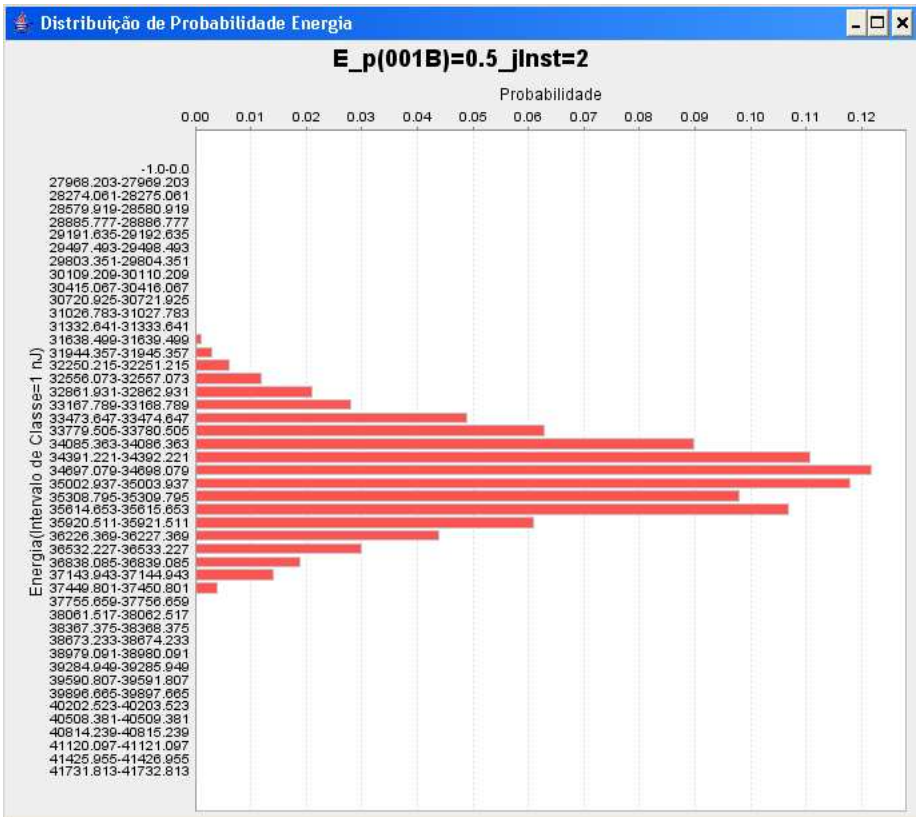

Figura 6.14: Distribuição de probabilidade de consumo de energia para o cenário P(*inst26*)=0.5

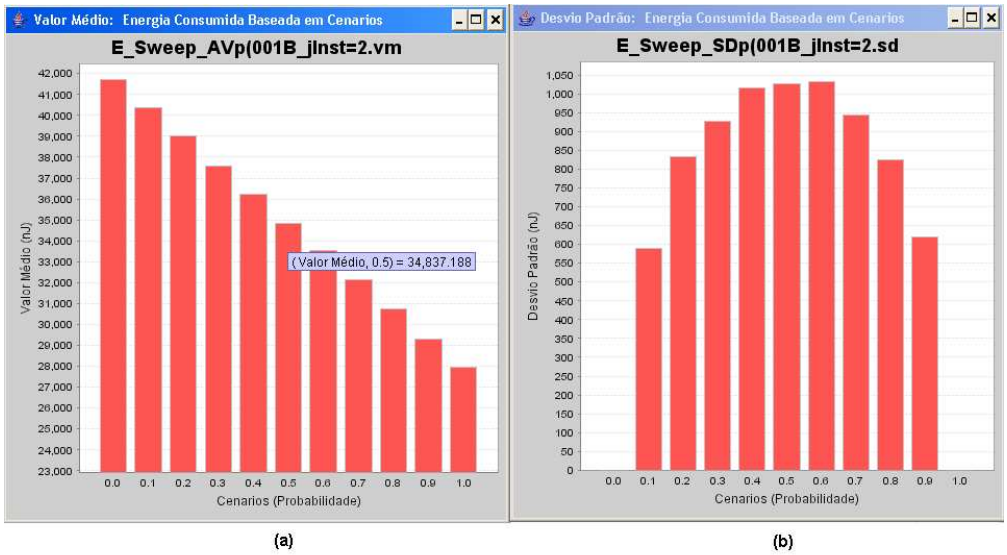

Figura 6.15: Consumo de Energia em função do cenário. (a) Valor médio (b) Desvio Padrão

de medidas [45], apresentando assim baixa variação na probabilidade do evento de permutação.

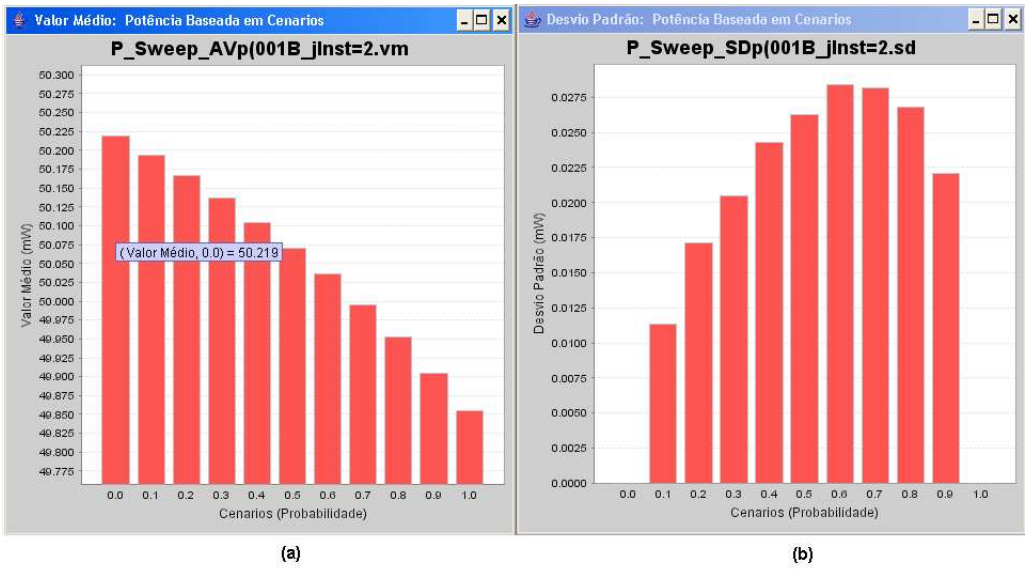

Figura 6.16: Potência em função do cenário. (a) Valor médio (b) Desvio Padrão

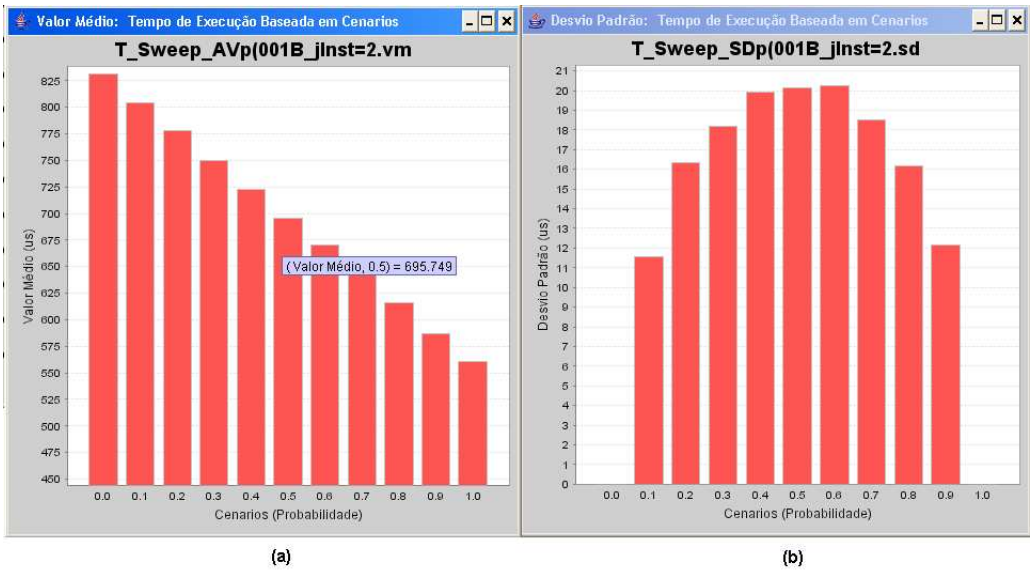

Figura 6.17: Tempo de Execução em função do cenário. (a) Valor médio (b) Desvio Padrão

6.6 Experimento 3: Filtro de Convolução

6.6.1 Descrição

Este experimento teve como objetivo aplicar a simulação probabilística para identificação do padrão de consumo de um filtro digital de ponto fixo com 20 células multi-

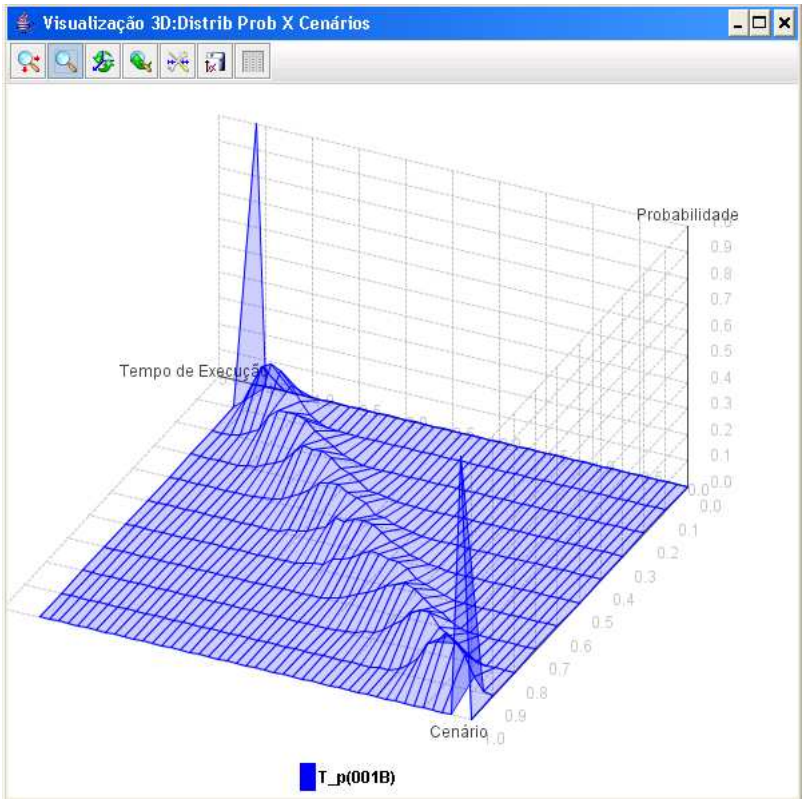

Figura 6.18: Visualização panorâmica: distribuição de probabilidades do tempo de execução versus cenários

plicação-adição implementado em linguagem de montagem (*assembly*). A simulação do modelo foi realizada considerando, como tempo de execução, o tempo necessário para se processar um dado de entrada (três *bytes*). Em sua maioria, os desvios condicionais desse código estão voltados para o teste dos valores presentes nas células de atraso. Caso o valor seja zero a rotina de multiplicação não é executada, o que causa forte impacto no tempo de execução e no consumo de energia do código. As anotações de instrução desses desvios condicionais foram configuradas de forma a permitir uma varredura de 0,0 a 1,0 com passo de 0,1, e com probabilidade no rCS igual a 0,5. O número limite de iterações foi arbitrariamente configurado para vinte iterações. O mesmo critério foi aplicado para os demais desvios observando apenas pequenas variações no valor da probabilidade no rCS. O código possui 51 instruções probabilísticas, perfazendo 561 cenários, resultando na realização 562.122 simulações. Adicionalmente, o mecanismo de validação apresentado na seção anterior foi empregado também nesse experimento, mostrando a eficiência da análise comparada às medidas efetuadas no *hardware*. Neste experimento, os vetores de teste alimentaram os valores armazenados nas células de atraso do filtro. Identicamente ao modelo, a execução foi definida como o processamento de uma entrada de dados.

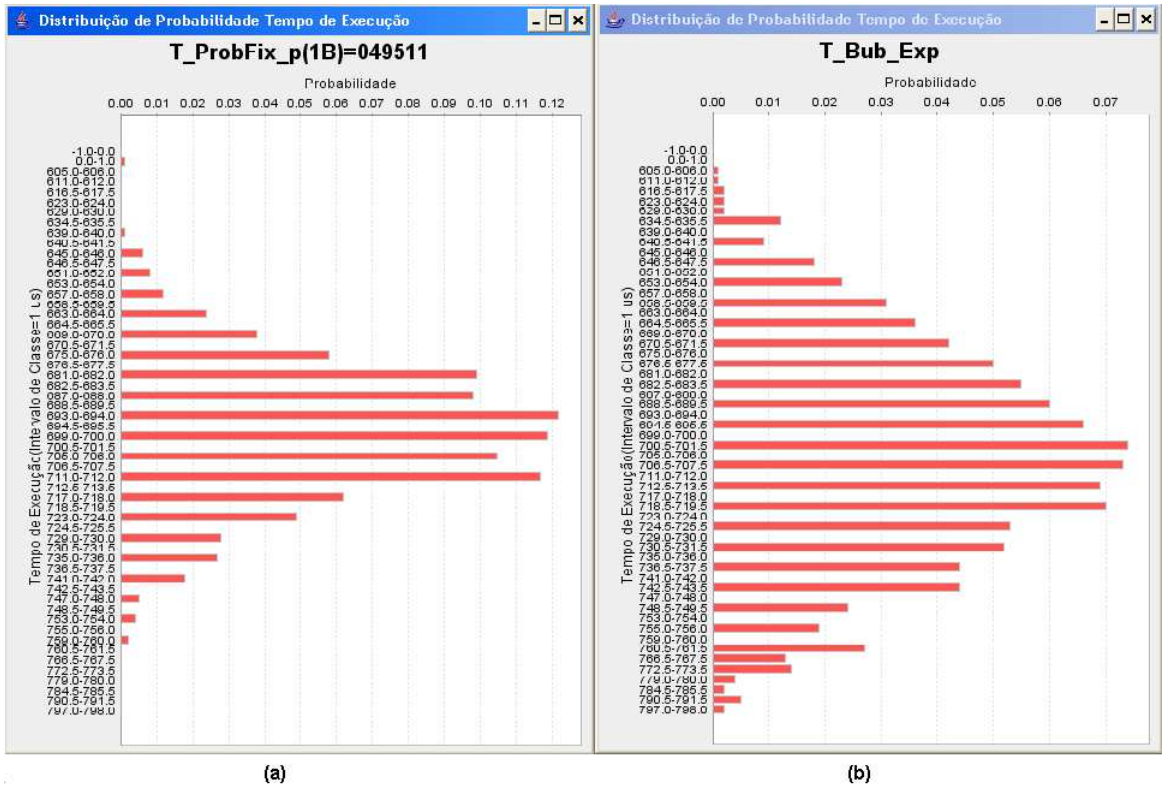

Figura 6.19: Verificação de consistência com execução em *hardware*:(a) Distribuição de probabilidade para P(*inst26*)=0.49511 (b)Distribuição gerada pelo experimento em *hardware*

6.6.2 Resultados

A análise dos cenários mostrou que este código possui baixa variação da distribuição de probabilidades do tempo de execução e do consumo de energia em função dos cenários. Isso pode ser explicado pela homogeneidade de comportamento dos desvios condicionais. Em sua grande maioria, esses desvios replicam a mesma estrutura de código mudando apenas as variáveis alvo. Isso faz com que cada um apresente uma contribuição similar ao tempo de execução e ao consumo de energia. Dado o grande número de instruções nessa condição, os possíveis cenários, regidos por uma só instrução, não apresentam impacto sobre o comportamento estatístico do código. A análise definiu perfeitamente esse comportamento. A Figura 6.20 apresenta o panorama de consumo de energia gerado pela exploração dos cenários criados pela instrução situada na posição 021A(*inst021A*) da memória, analisando com intervalo de classe de 3μJ. A distribuição de probabilidades de consumo possui um perfil de natureza gaussiana, apresentando o pico de probabilidade na classe definida pelo intervalo $[160.326, 163.326]$ μJ.

A Figura 6.21 mostra a variação do valor médio e o nível de dispersão para cada

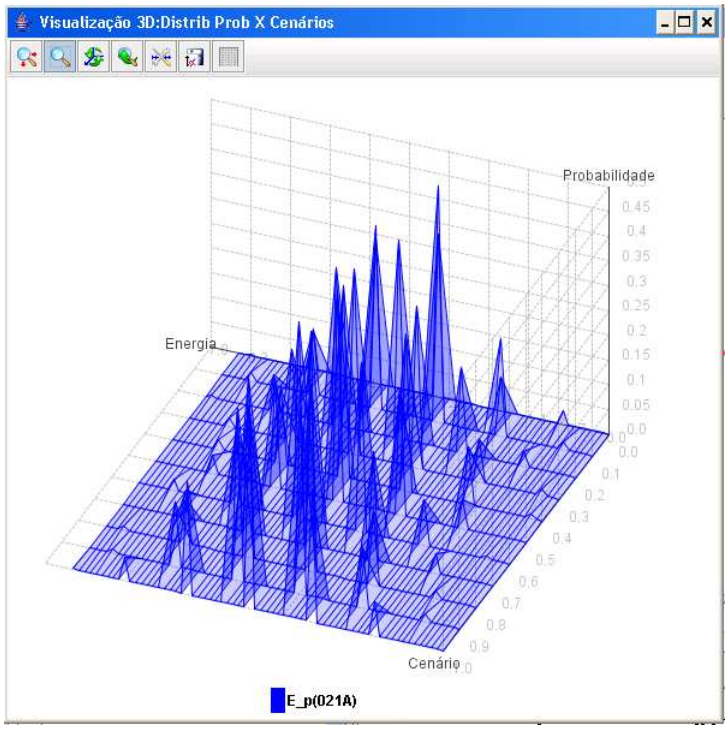

Figura 6.20: Visualização panorâmica: distribuição de probabilidades do consumo de energia versus cenários

cenário da *inst021A*. Observa-se que a maior variação do valor médio ocorre entre o cenário 0,4 e 0,6, sendo contudo extremamente baixa, 0,22%. O padrão de dispersão (desvio padrão) apresenta uma variação máxima de 6,7%, o que indica o baixo impacto de uma instrução de desvio isoladamente no padrão de consumo desse código.

A Figura 6.22 permite a análise comparativa entre a distribuição de probabilidades de tempos de execução, medida no experimento de *hardware*, e a estimada pelo modelo. Mais uma vez observa-se que a dispersão da curva é maior no experimento em *hardware*, devido ao mesmo fato apresentado na seção anterior. Contudo, o erro na determinação do tempo de execução mais provável é de apenas 0.18%, 3.306ms[3] no modelo contra 3.312ms no experimento em *hardware*. Esse erro é particularmente baixo devido à natureza quase-estática do comportamento estatístico do código. Os resultados mostram a consistência do modelo frente à execução real em *hardware*.

A Figura 6.23 mostra o perfil de probabilidade de execução do código. Com base nele, pode-se identificar os *Patches* e *Clusters* de maior probabilidade de execução. Essa informação pode ser usada para alimentar as análises de otimização do *hardware* de memória e migração *software-hardware*.

[3]Considerando o maior valor dentro do intervalo da classe.

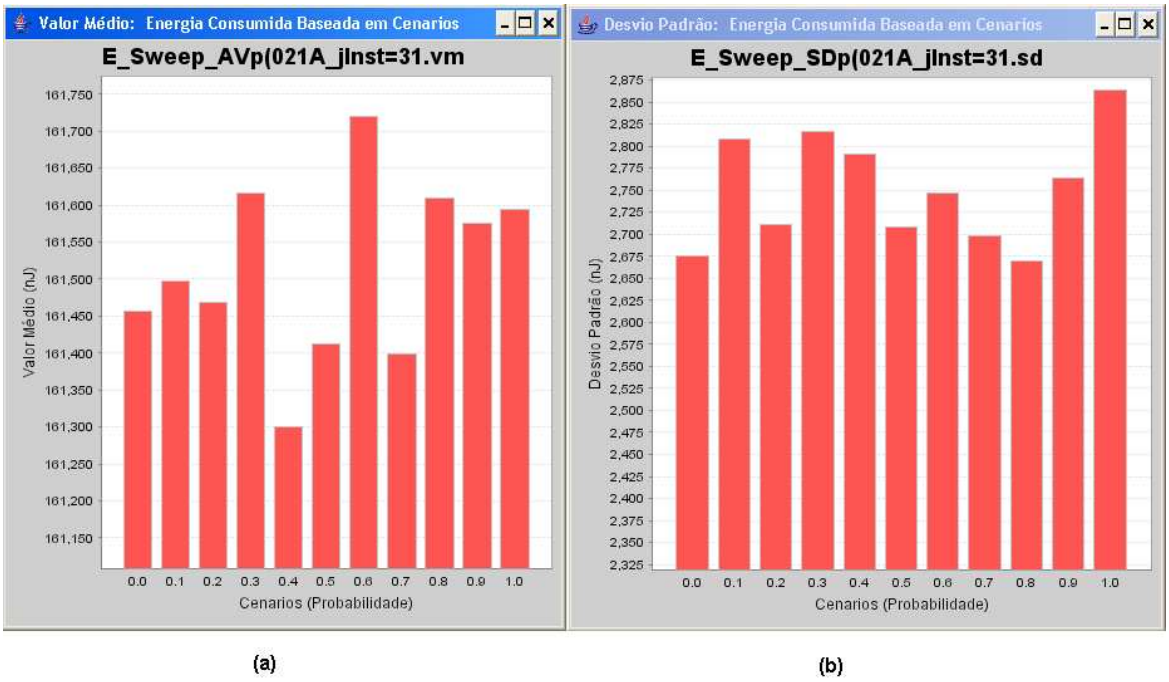

Figura 6.21: Consumo de energia em função do cenário, Experimento 3. (a) Valor Médio (b) Desvio Padrão

6.7 Experimento 4: Sistema *SmartOk*

6.7.1 Descrição

O SmartOk foi avaliado objetivando aplicar o arcabouço de análise em uma aplicação real da indústria. O SmartOk é um módulo de monitoração e controle de vazão de água em instalações condominiais. Ele faz parte de um sistema de tele-supervisão desenvolvido pela TechnOk, empresa do grupo Procenge. As Figuras 6.25 e 6.24 ilustram o sistema no qual o SmartOk está inserido. Ele é constituído por um micro-controlador AT89S8252, monitorando pulsos enviados por um hidrômetro de saída pulsada[4] e atuando sobre uma válvula de oclusão. O micro-controlador envia as leituras de vazão e recebe comandos por meio de uma rede de controle local implementada sobre a interface elétrica RS232C. Existem quatro comandos possíveis: fechar válvula, abrir válvula, realizar leitura e inicialização. O comando de inicialização cumpre o papel de carregar o valor inicial do hidrômetro na memória do SmartOk. Dessa forma, o SmartOk possui uma imagem dos valores acumulados no hidrômetro. A medida de vazão é realizada por meio da contagem de transições de descida dos pulsos enviados pelo hidrômetro.

[4]Terminologia usada para designar hidrômetro que fornece pulsos como registro de vazão.

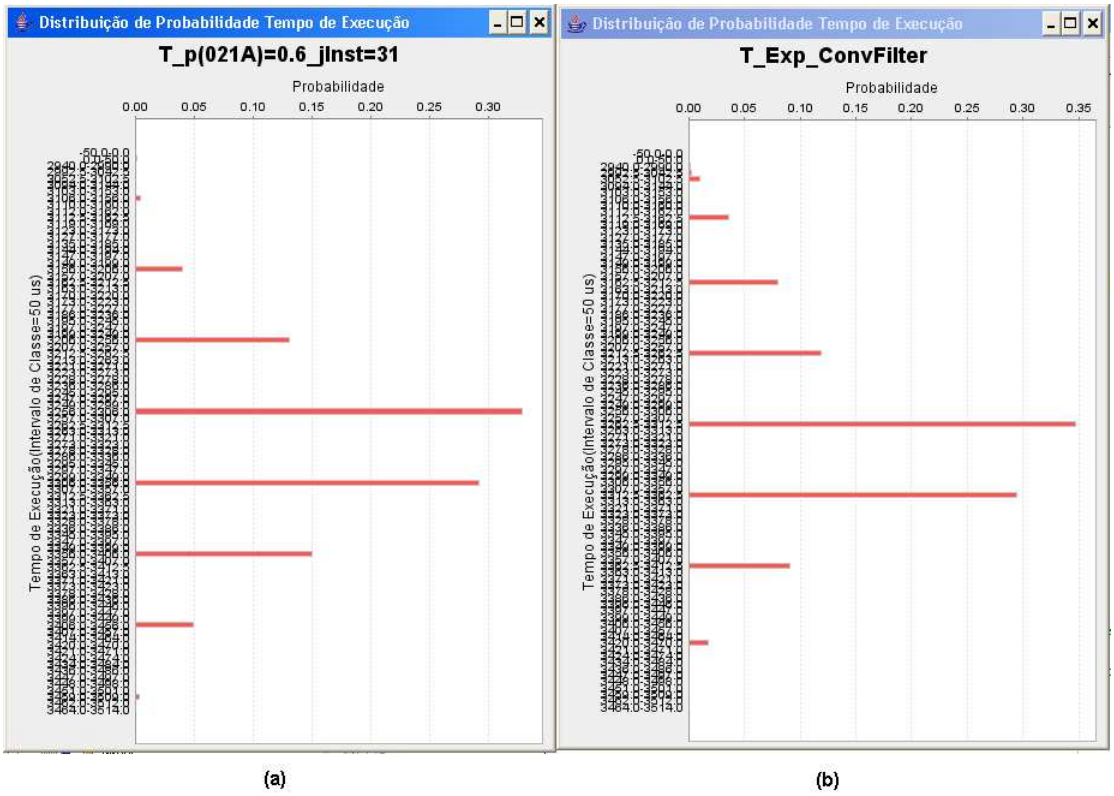

Figura 6.22: Verificação de consistência com execução em *hardware*:(a) Distribuição de probabilidade para P(021A)=0.6 (b)Distribuição gerada pelo experimento em *hardware*

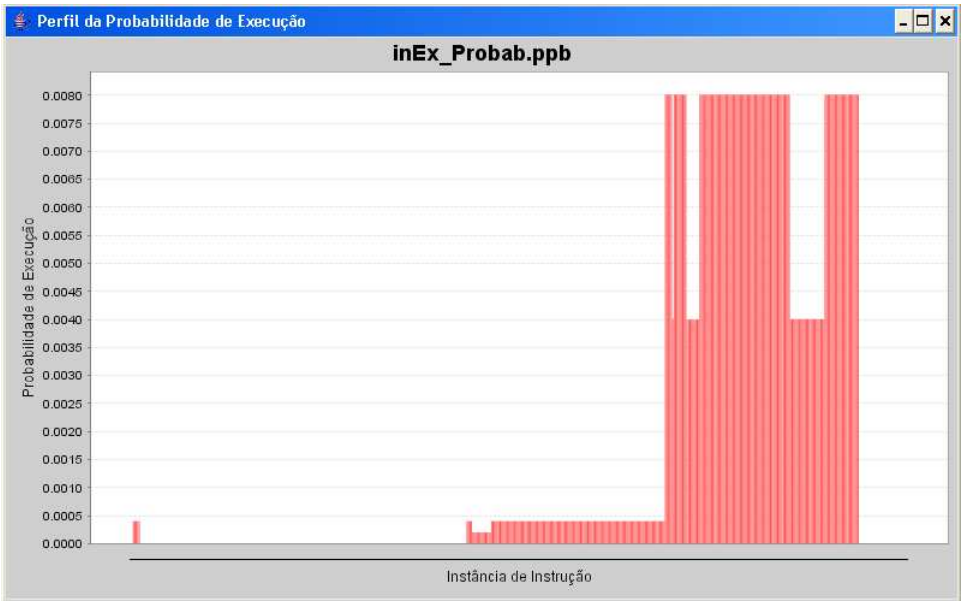

Figura 6.23: Perfil de probabilidade de execução por instância de instrução

O código opera varrendo continuamente a entrada de pulsos e o registrador de interrupção serial. O processo de supervisão e controle é realizada por um outro módulo

denominado ConnectOk. O ConnectOk permite que um operador realize medidas de vazão, bloqueios e desbloqueios, bem como programação de políticas de racionalização do consumo de água.

A avaliação do código presente no SmartOk visou a identificação de seus padrões de comportamento e seus respectivos consumo de energia e tempo de execução. O tempo de execução a ser analisado foi definido como o tempo de uma varredura de busca nas entradas de pulso e interrupção da serial. A verificação de consistência em *hardware* foi realizada comparando os tempos de execução para três condições básicas medidas em *hardware*, com o estimado pelo modelo. As condições básicas foram: sinal do hidrômetro em nível alto, em nível baixo e ocorrência de evento de comunicação. As probabilidades do rCS foram calculadas tomando como base o período de um mês. Por exemplo, a freqüência esperada para um evento de comunicação é de uma vez por mês, baseando-se na contagem de instruções e estimativa manual de tempo de execução, foi assumida uma probabilidade 1.10^{-9} de haver um evento de comunicação durante uma varredura.

Entre os comandos, existe uma divisão bem desigual quanto à probabilidade de ocorrência, o comando de leitura é o de maior probabilidade $(0,968)$, seguido pelos de abertura e fechamento da válvula $(0,015)$, sendo o comando de inicialização o de menor probabilidade $(0,002)$. Observa-se que a probabilidade de ocorrência de um comando específico é uma probabilidade condicional, ou seja, é gerada naturalmente no modelo como o produto da probabilidade de haver uma comunicação (1.10^{-9}), que se traduz pelo evento de interrupção da serial, pela probabilidade da comunicação trazer aquele comando específico. Dessa forma, a percepção *ad hoc* que se tem do comportamento do código é que seu tempo de execução e consumo de energia é determinado pelo processo de medida, que sofre pertubações devido aos eventos de comunicação. O arcabouço de análise proposto averigua e quantifica tais percepções *ad hoc* do projetista - por meio da modelagem probabilística -, possibilitando traçar um panorama amplo de execução.

Figura 6.24: Sistema SmartOk

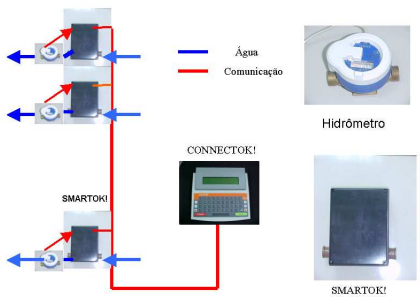

Figura 6.25: Módulos do Sistema SmartOK

6.7.2 Resultados

Os resultados comprovaram a suposição prévia de que existem dois contextos principais de execução: um referente ao processo contínuo de medida e outro referente à comunicação, sendo o processo de medida dominante nos padrões de consumo e tempos de execução. A Figura 6.26 apresenta a distribuição de probabilidades de consumo de energia para um comando de inicialização, (*inst00A0*) operando no cenário P(*inst00A0*)=0,5 e o panorama de consumo a partir do conjunto de cenários gerados por ela. A análise comprova a dominância dos eventos de consumo devido aos eventos de comunicação, quantificando-a em termos de consumo de energia. Na Figura 6.26(a), observa-se que o consumo mais provável do código se encontra entre 2.383μJ e 2.595μJ, havendo eventos de baixa probabilidade de consumo na faixa de 377.759μJ a 377.776μJ. Esses eventos, quando ocorrem, provocam grandes aumentos do consumo de energia. Esse padrão foi observado para todos os cenários gerados. Como era de se esperar, o mesmo padrão foi encontrado nas distribuições associadas ao tempo de execução. A Figura 6.27 mostra a distribuição de probabilidades do tempo de execução do cenário apresentado na Figura 6.26(a). Os testes comparativos em *hardware* mostraram que os intervalos de tempo apresentados correspondem respectivamente: ao sinal do hidrômetro em nível alto (47μs), em nível baixo (48μs), e à ocorrência de evento de comunicação (7,170ms). Observa-se ainda que há um evento de menor probabilidade associado a um tempo de execução de 51μs, este valor é consistente com o processamento de uma transição de descida do sinal do hidrômetro.

Como teste adicional, além do comparativo em *hardware*, foi verificado se o modelo realmente estava identificando os eventos de comunicação. As anotações de instrução foram modificadas de forma a forçarem uma alta incidência dos eventos de comunicação, e o código foi avaliado mais uma vez. A Figura 6.28 mostra o resultado obtido. A simulação capturou perfeitamente os eventos de comunicação, comprovando que os pontos observados nas Figuras 6.26 e 6.27 representam as classes de valores associados aos eventos de comunicação.

Este experimento em especial, demonstrou a importância da avaliação do consumo em termos das curvas de distribuição de probabilidades, não somente em termos de valores médios. A Figura 6.29 apresenta a evolução do tempo de execução em função dos cenários gerados por *inst00A0*. Uma vez que os cenários de *inst00A0* oferecem impacto mínimo no padrão de comportamento do código (vide Figura 6.26 (b)), a Figura 6.29 é representativa do tempo de execução médio do código. Observa-se que estes valores não são coerentes com os valores mais prováveis de tempo de execução apresentados na Figura 6.27, isso é explicado pelo fato da distribuição de probabilidades

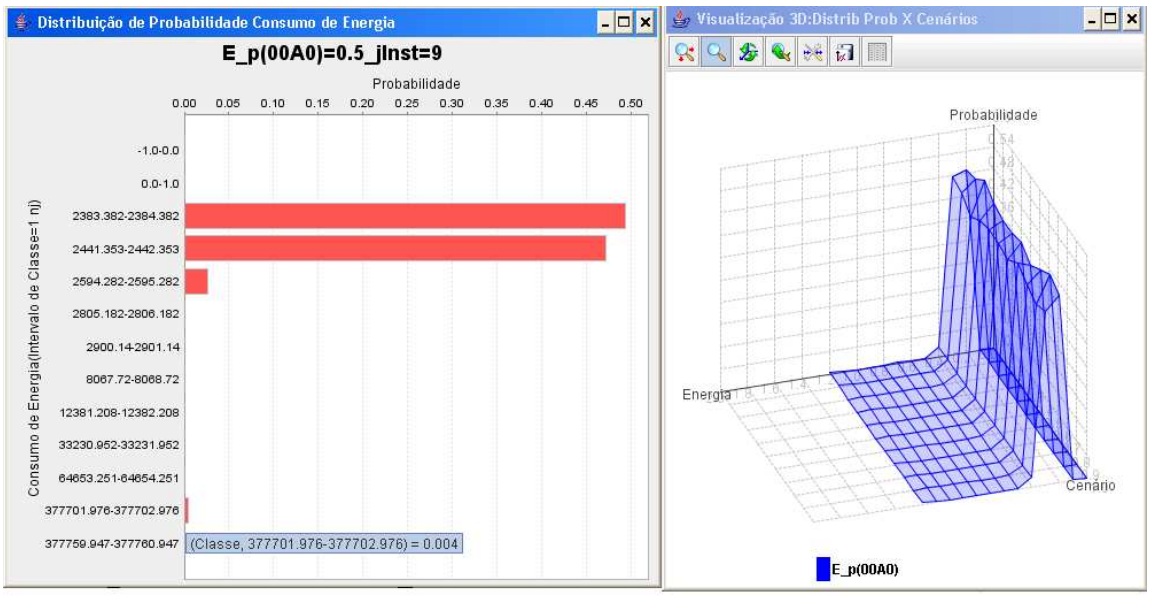

Figura 6.26: Avaliação do SmartOk: (a)Distribuição de probabilidade de consumo para p(*inst00A0*)=0,5. (b)Distribuição de probabilidade de consumo versus cenários gerados pela *inst00A0*

Figura 6.27: Distribuição de probabilidade de tempo de execução para p(00A0)=0,5.

não ser uma curva gaussiana. Isso reforça a importância da análise das curvas de distribuição de probabilidades frente à análise de valores médios.

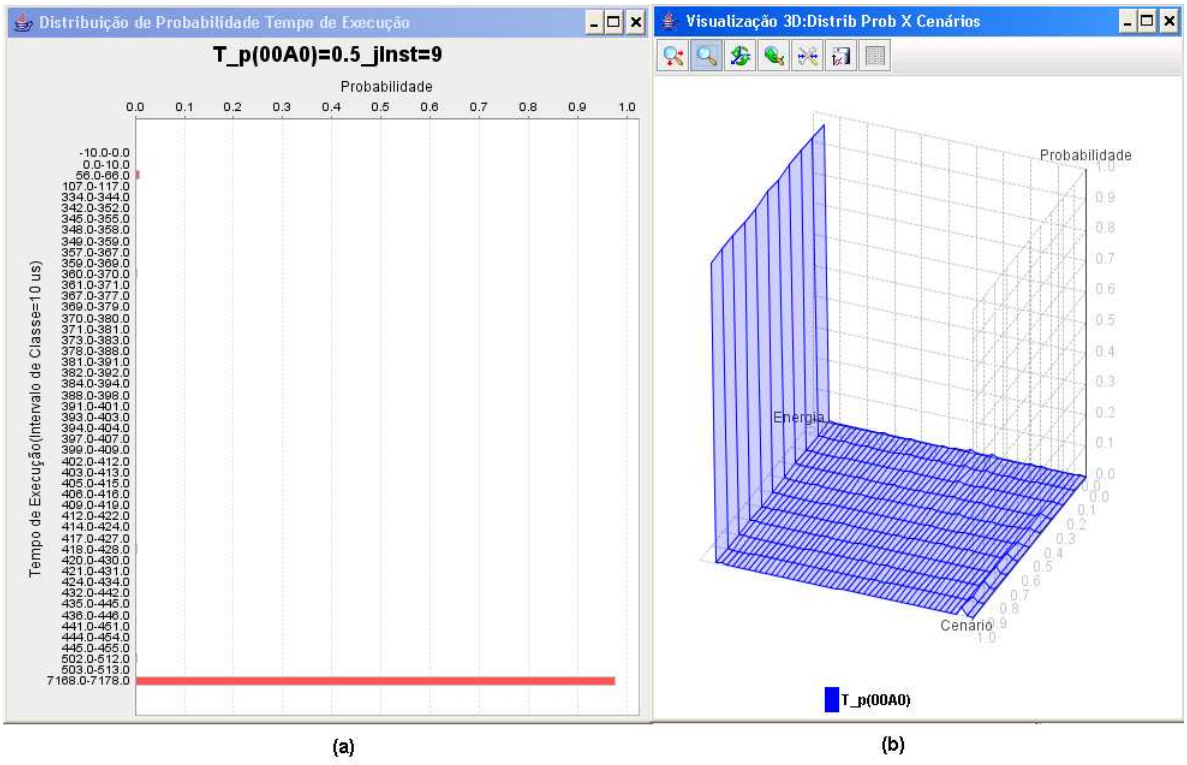

Figura 6.28: Avaliação do SmartOk com ênfase nos eventos de comunicação: (a)Distribuição de probabilidade de tempo de execução para p(*inst00A0*)=0,5. (b)Distribuição de probabilidade de tempo de execução versus cenários gerados pela *inst00A0*

6.8 Experimento 5: Inter-comunicador

6.8.1 Descrição

Para servir como objeto do último experimento, um pequeno inter-comunicador foi projetado. Desta forma o mecanismo de análise pôde ser testado sobre uma aplicação propriamente dita, não uma função ou módulo funcional. A Figura 6.30 ilustra a estrutura do inter-comunicador. O sistema é formado por um único circuito de enlace interligando quatro aparelhos telefônicos. Um micro-controlador AT89S8252 opera como controlador, gerenciando as comutações e emitindo os tons de sinalização. Circuitos de linha (CLs na Figura 6.30) operam como sensores para detecção de fones fora do gancho. A alocação do circuito de enlace é realizada por meio de chaves analógicas comandadas diretamente pelo micro-controlador. O tom de discar consiste na emissão de um tom de 450Hz. O sinal de chamada é implementado com o mesmo tom operando por um segundo seguido de quatro segundos de silêncio. O mesmo sinal é enviado para

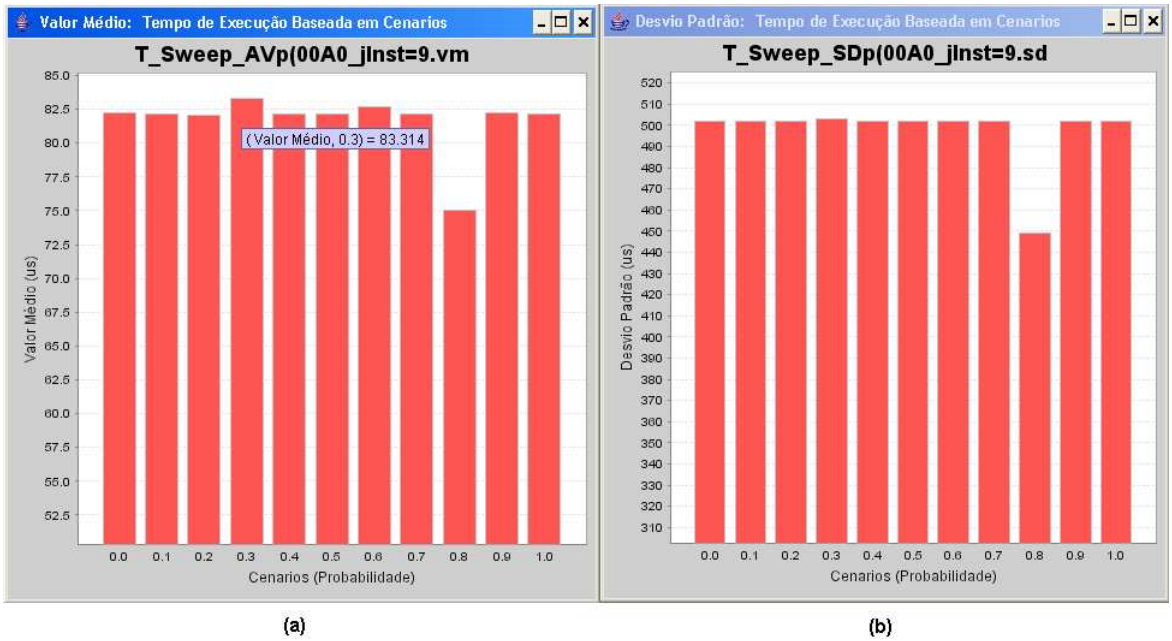

Figura 6.29: Consumo de energia em função do cenário, Experimento 4. (a) Valor Médio (b) Desvio Padrão

o telefone recebedor, constituindo o sinal de toque. A discagem é interpretada como pulsos que codificam os números de ramal, sendo estes de 1 a 4. Se o telefone recebedor estiver fora do gancho ou não atender em 20 segundos o originador recebe um tom de ocupado. Todos estes recursos foram implementados no código sem o uso dos temporizadores do AT89s8252.

Para reduzir o impacto no tempo de simulação, causado pela avaliação de todo o sistema, foram feitas duas simplificações para a análise. Na primeira, as anotações são inseridas de forma a se considerar tempo zero entre o ato de tirar o fone do gancho e de se realizar a discagem. A segunda assume que o recebedor sempre atende. O tempo de execução foi definido como o intervalo entre o originador tirar o fone do gancho e o recebedor atender. Desta forma o modelo do inter-comunicador explora as possibilidades de execução referentes à combinação de número originador e número recebedor.

6.8.2 Resultados

As Figuras 6.31 e 6.32 apresentam um panorama do tempo de execução e do consumo de energia em função de todos os cenários gerados por todas as instruções de desvio condicional do código. Adicionalmente, essas figuras mostram a distribuição de probabilidades gerada por um cenário específico dentro daquele panorama, provendo uma

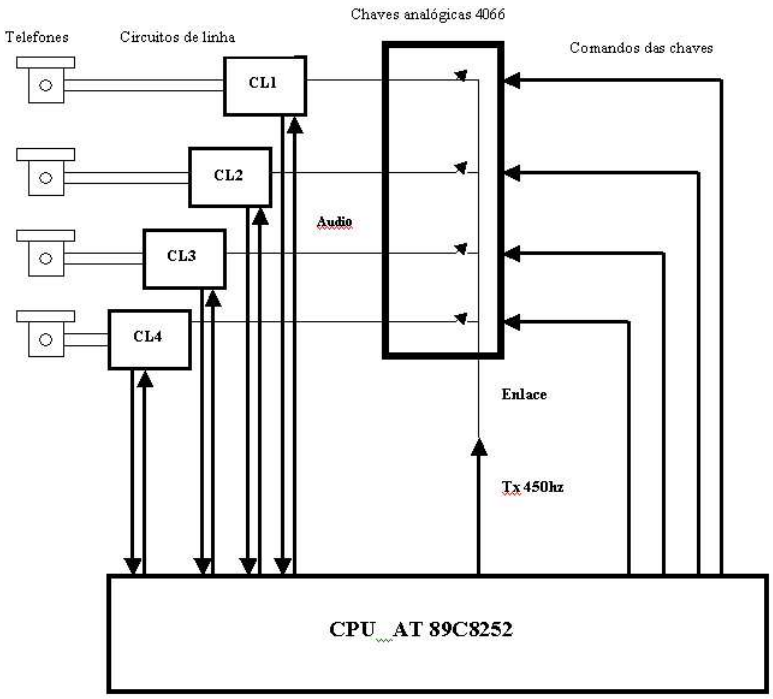

Figura 6.30: Experimento 5: Inter-Comunicador

visão de corte. Para uma melhor visualização, esses gráficos apresentam intervalo de classe de 5 segundos, para tempo de execução, e 0,3J para consumo de energia. Vale lembrar que tal intervalo de classe é ajustável, permitindo que o projetista avalie a distribuição de probabilidades para diferentes intervalos de classe. Analisando-se tais gráficos, por meio dos recursos de aproximação (*zoom*) e medida (*tooltips*), observa-se que as classes de consumo abaixo de 0,3J possuem probabilidades da ordem de 0,68. Classes de consumo superiores aparecem com baixas probabilidades associadas. O mesmo padrão é observado com relação ao tempo de execução, classes inferiores a 5s possuem probabilidades da ordem de 0,61. Assumindo a relação direta entre probabilidade e freqüência de eventos, a análise informa que o consumo de energia será inferior a 0,3J para 68% das comutações executadas neste sistema. Identicamente, pode-se inferir que em 61% das chamadas o processo de comutação levará menos de 5s.

6.9 Conclusões

Os experimentos mostraram a aplicação do modelo e do arcabouço de análise na captura de informações cruciais às estratégias de otimização de sistemas embutidos. O modelo determinístico mostrou-se eficaz à identificação e análise de trechos críticos do código, auxiliando análises voltadas à otimização e a avaliações de particionamento

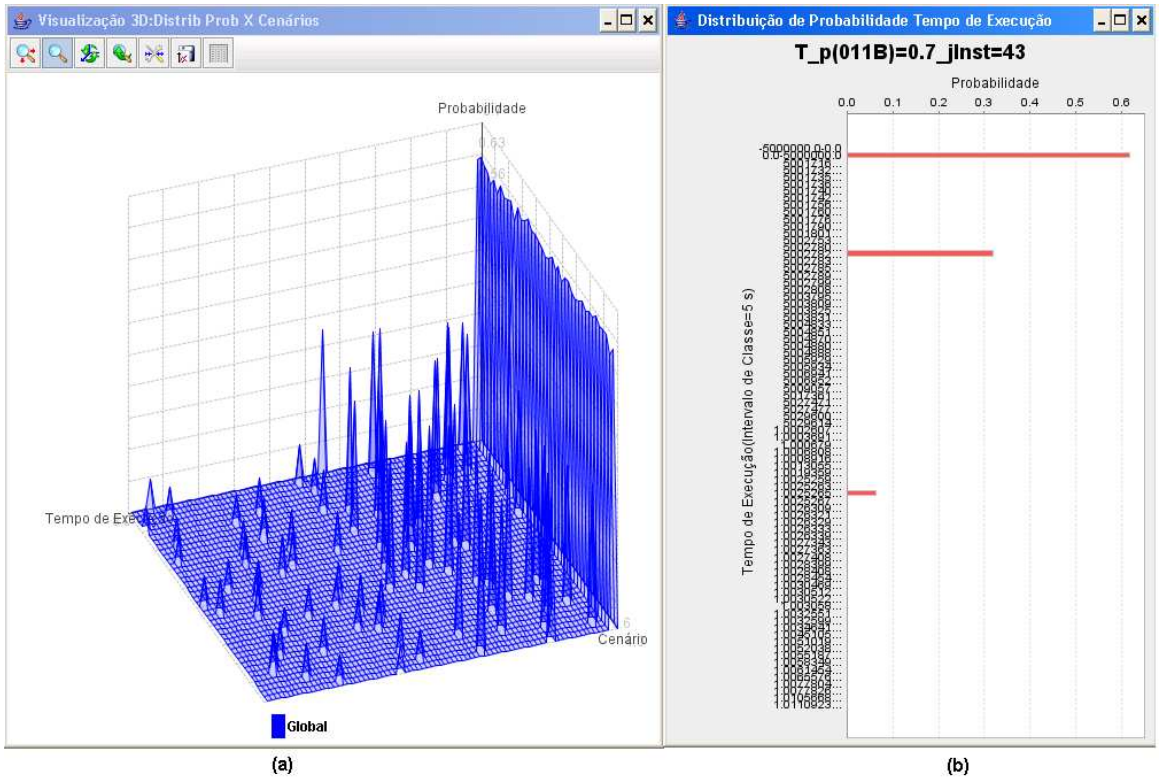

Figura 6.31: Experimento 5: (a) Panorama global do tempo de execução (b) Amostra de um cenário

software-hardware. O modelo mostrou completa fidelidade ao comportamento real da aplicação, simulando perfeitamente todos os estados internos[5] do micro-controlador alvo. As variáveis sondas mostraram-se de fácil implantação e eficientes em capturar esses estados internos, sem acarretar restrições funcionais no modelo de simulação. O modelo probabilístico mostrou-se eficaz em seu propósito de traçar perfis probabilísticos do comportamento do código. Os Experimentos 2 e 3 mostraram que o modelo é capaz de explorar com grande fidelidade o comportamento do código. O Experimento 2 demonstrou que uma grande gama do espectro comportamental do código pode ser estimado mediante a inferência do comportamento probabilístico das instruções de desvio, e que tais inferências podem ser geradas com base nos algoritmos de implementação do código. O Experimento 3 demonstrou a capacidade do modelo de traçar perfis em códigos com elevado número de instruções de desvio, elucidando o impacto das instruções individualmente no comportamento geral do código. No caso específico do Experimento 3 (filtro de convolução), a distribuição de probabilidades, praticamente homogênea para qualquer cenário (vide Figura 6.20), elucida o baixo impacto de uma instrução de desvio isoladamente. Ambos os experimentos mostraram que os perfis ge-

[5]Referente ao nível de abstração proposto.

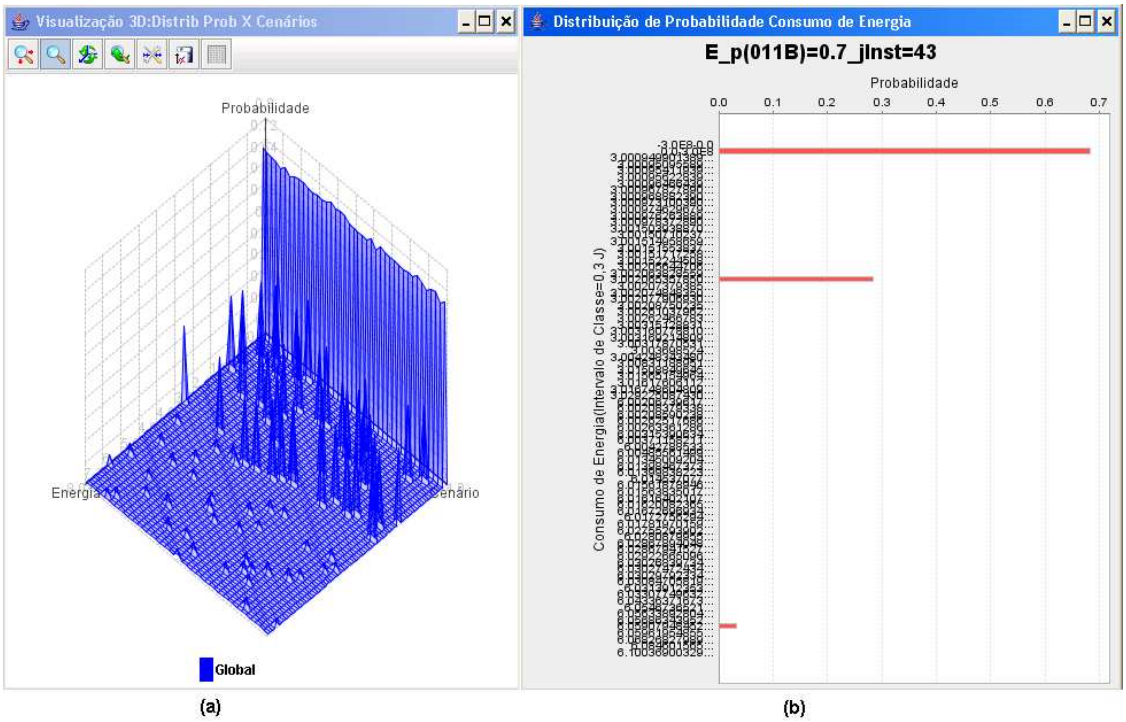

Figura 6.32: Experimento 5: (a) Panorama global do consumo de energia (b) Amostra de um cenário

rados guardam forte consistência com o comportamento real do código executado em *hardware* (vide Figuras 6.19 e 6.22). Os Experimentos 4 e 5 demonstraram que o arcabouço de análise é capaz de identificar padrões comportamentais específicos do código, mapeando-os em classes de tempo de execução e consumo de energia. O Experimento 4 comprovou que classes de elevada probabilidade estão diretamente associadas a padrões de comportamento específicos do código, elucidando o comportamento do código para o sinal de hidrômetro em nível alto, nível baixo e transição de nível. Adicionalmente, o experimento demonstrou que comportamentos latentes podem ser detectados pela análise da distribuição de probabilidades de classe em cada cenário. No Experimento 4, foi detectada a latência do comportamento associado à comunicação de dados do sistema. Foi demonstrado que tais comportamentos latentes podem ser investigados por meio de modificações nas anotações de instrução. No Experimento 4, as classes de consumo e tempo de execução associadas à comunicação foram identificadas aumentando-se artificialmente a probabilidade dos eventos de comunicação. O Experimento 5 exemplificou a interpretação dos resultados da análise como a freqüência de consumo de energia associada a um evento do sistema. No caso, foi estimado que 68% dos eventos de comutação do inter-comunicador consumiriam menos de 0,3J, e que 61% levariam menos de 5s para serem executadas.

No concernente a velocidade de simulação, os experimentos demonstraram a necessidade da implementação de um simulador específico, centrado em desempenho. O CPNTools mostrou-se adequado para a construção, validação e experimentação dos modelos, não sendo adequado, contudo, como mecanismo final de simulação. Executado em um Athlon 3000+ com 512 MBytes de memória RAM, o CPNTools mostrou-se capaz de executar em média 412 instruções por segundo no modelo determinístico e 34720 instruções por segundo no modelo probabilístico. Durante os experimentos observou-se que essas taxas diminuem à medida que o tamanho do código (rede) aumenta. Observa-se que a grande diferença de desempenho entre os dois modelos deve-se à inexistência de processamento comportamental da instrução no modelo probabilístico. Adicionalmente, o CPNTools acrescenta um tempo de verificação sintática da rede, que é proporcional ao tamanho da rede. Tal latência pode ser completamente retirada em um simulador específico, uma vez que o compilador Binário-CPN garante a correção da rede gerada. A Tabela 6.4 apresenta os tempos de verificação sintática e simulação para os experimentos, bem como os tempos de execução em *hardware* associados ao processo de verificação de consistência. É importante ressaltar que a análise probabilística compreendeu dezenas (Experimento 2) ou mesmo centenas (Experimento 5) de milhares de simulações, enquanto as verificações em *hardware* foram realizadas com base em 1000 ciclos de execução e medida. Feita essa ponderação, conclui-se que a simulação do modelo probabilístico, mesmo usando o CPNTools, é mais rápida do que a avaliação em *hardware*. Além do mais, experimentos em *hardware* exigem a criação de vetores de teste, o que é eliminado pela modelagem probabilística do código.

Tabela 6.4: Tempos de simulação do modelo CPN

Experimento	Checagem Sintática	Simulação	Verif. *Hardware*
1-Exp. Compilação	40 minutos e 6 segundos	6 minutos	–
2-Orden. em Bolha	16 minutos e 21 segundos	40 minutos e 10 segundos	4 horas e 23 minutos
3-Filtro de Convolução	43 minutos e 34 segundos	24 horas e 6 minutos	7 horas e 11 minutos
4-Sistema SmartOk	11 minutos e 40 segundos	44 horas e 6 minutos	–
5-Inter-Comunicador	12 minutos e 59 segundos	71 horas e 4 minutos	–

Os experimentos mostraram a aplicação direta do arcabouço de simulação e análise propostos nessa tese para a avaliação do consumo de energia, bem como sua distribuição ao longo do código. Dado o mecanismo de modelamento, diferentes micro-controladores podem ser avaliados pela concepção de seus modelos CPN de instruções e das classes de instruções do compilador Binário-CPN. Esse aspecto dá subsídios para a construção de uma linguagem de descrição de arquiteturas formalmente centrada em redes de Petri.

Capítulo 7

Conclusão e Trabalhos Futuros

7.1 Conclusão

Esta tese propõe dois modelos para a análise de consumo de energia de código no contexto dos sistemas embutidos. Tais modelos são baseados na descrição do conjunto de instruções da arquitetura alvo em redes de Petri coloridas. O primeiro modelo aplica as redes de Petri coloridas para a avaliação do comportamento do código frente a um modelo determinístico das instruções. O segundo modelo estende o primeiro visando a explorar o espaço de possibilidades de execução do código por meio da modelagem probabilístico dos possíveis fluxos de execução do código. Tal estratégia permite abstrair o conceito de vetor de teste, mapeando-o em uma descrição probabilística da aplicação, constituindo uma metodologia inédita na avaliação de códigos. Em ambos os casos, o modelo é formalmente construído seguindo a semântica CPN. Devido ao formalismo CPN assumido, mecanismos de simulação e análises são automaticamente construídos de forma independente da arquitetura alvo. Ferramentas e algoritmos, validados para simulação de redes de Petri coloridas, podem ser aplicados diretamente ao modelo. Tal premissa foi constatada pela construção de arcabouços de modelagem e análise, cuja etapa de simulação é implementada por uma ferramenta de uso genérico, o CPNTools. Adicionalmemente, a base formal empregada permite o uso das técnicas de análise comuns ao universo das redes de Petri.

A metodologia proposta oferece três contribuições básicas:(i) criação de modelos de descrição de arquiteturas sobre uma linguagem de modelagem formal, de grande penetração e suportada por mecanismos de simulação e análise, as redes de Petri coloridas; (ii) aplicação de descrição explicitamente estrutural do *software*, capaz de elucidar as razões estruturais de determinado padrão de consumo e desempenho, de suma importância para o estabelecimento de figuras de mérito para a migração (parti-

cionamento) *software-hardware*; (iii) proposição de modelo probabilístico inédito para descrição, simulação e avaliação de consumo de energia devido ao *software*, atacando o problema de dependência de padrão pela eliminação do vetor de teste, seja ele determinístico ou estocástico, em favor de um modelo probabilístico de comportamento do *software*. Dessa forma, estabelecendo uma metodologia nova para análise de consumo e promovendo um formalismo baseado em redes de Petri coloridas para a criação de ferramentas.

Conforme apresentado no Capítulo 4, o processo de modelagem não é centrado na construção de um simulador de processador, mas sim em um mecanismo formal de simulação do fluxo de execução de uma aplicação. Isso constrói um mecanismo funcionalmente idêntico ao simulador, uma vez que reproduz o comportamento exato do processador, mas cuja regra de construção é baseada em redes de Petri. Diferentemente de outras abordagens voltadas à descrição de arquiteturas, a descrição e simulação da arquitetura alvo não é baseada em um mecanismo proprietário, concebido de forma *ad hoc*, ou guardando um formalismo especificamente proposto para o qual não existe ferramentas de análise. A aplicação do formalismo de redes de Petri estabelece o pressuposto do emprego dos recursos de análise descritos no Capítulo 3, recursos aceitos e amadurecidos pela comunidade científica. A aplicação de tais recursos no modelo apresentado, estabelece perspectivas positivas com relação à capacidade analítica inerente ao modelo proposto.

No contexto da simulação probabilística, o modelo proposto implementa os preceitos lançados por Burch *et al* em [16], estabelecendo mecanismos de inferência probabilística para a avaliação do desempenho e consumo de energia associados ao *software*. A metodologia proposta estabelece as seguintes contribuições: (i) aplicação específica para processadores, (ii) avaliação centrada no modelo de consumo do *software*, (iii) exploração dos cenários de execução por fluxos de execução aleatórios parametrizáveis a partir de inferências de probabilidade dos eventos do sistema, (iv) capacidade de investigação de cenários específicos de execução, (v) formalismo decorrente de ser um MCD e (vi) reconfigurabilidade quanto à arquitetura alvo.

Adicionalmente, esta tese demonstrou a aplicação da semântica das redes de Petri coloridas como uma ADL comportamental, objetivando a geração automática de mecanismos de análise de consumo de energia devido ao *software*. Dessa forma, estabelecendo a semântica interna básica para pesquisas na direção de ADLs formais, nas quais os mecanismos de verificação das redes de Petri poderão ser usados para verificar propriedades de projetos de arquiteturas e sistemas.

7.2 Trabalhos Futuros

Futuros trabalhos devem ser centrados na ampliação e aplicação dos conceitos aqui propostos na pesquisa de sistemas embutidos. A seguir são sugeridos alguns tópicos para trabalhos futuros.

7.2.1 Implementação do MCD-simplescalar

Um modelo CPN de descrição para arquiteturas *simplescalar* pode ser implementado como uma extensão natural da metodologia de modelagem do MCD. Nesta seção serão apresentadas as idéias para o modelo CPN-*Simplescalar* a ser desenvolvido em trabalhos futuros. A ponto de partida para o modelo CPN-*Simplescalar* consiste em expandir o modelo CPN-SemPipeline de forma a processar os custos decorrentes da parada de *pipeline* (*pipeline stall*) com um mecanismo similar ao do custo inter-instrução. A idéia básica é a introdução de um *token* "testemunho", andando na frente do *token* de "execução". O *token* de testemunho alimenta variáveis internas do modelo de forma que, a cada passo do *token* de execução, a presença de um contexto interno de conflito é avaliada. O *token* testemunho contém uma *sonda* no formato de lista, com número de elementos igual ao número de estágios do *pipeline*, transportando um registro das instruções presentes no *pipeline*. Este mecanismo constrói o que definimos como *pipeline virtual*, que pode ser visualizado como uma entidade, composta de cabeça e cauda, que se move pela rede sendo a cabeça representada pelo *token* de testemunho e a cauda pelo *token* de execução. A *sonda* de *pipeline* é compartilhada com o *token* de execução por meio de compartilhamento de uma variável global - uma *sonda*. A transição disparada pelo *token* de execução representa a instrução no último estágio do *pipeline*. Ao ser disparada, a transição executa a funcionalidade da instrução e calcula o custo inter-instrução daquele contexto do *pipeline*. Transições que geram desvios destroem o *pipeline* virtual pela modificação do *token* de testemunho. Essa modificação (um *flag* de sua estrutura de dados) instrui a próxima transição, habilitada pelo *token* de testemunho, a destruí-lo. Simultaneamente, a transição de desvio gera um novo *token* de testemunho, que segue o novo fluxo de execução. Uma sub-rede, interna à transição, gera o atraso necessário para a liberação do *token* de execução. Esse mecanismo de destruição e reconstrução do *pipeline* virtual modela o *pipeline flush*. Com esse mecanismo, caracteriza-se o custo de esvaziamento do *pipeline* como um evento na rede, passível de captura e análise. A Figura 7.1 ilustra a estrutura básica do modelo CPN-*Simplescalar*.

Os preceitos básicos expostos definem as linhas mestras de um modelo CPN *Sim-*

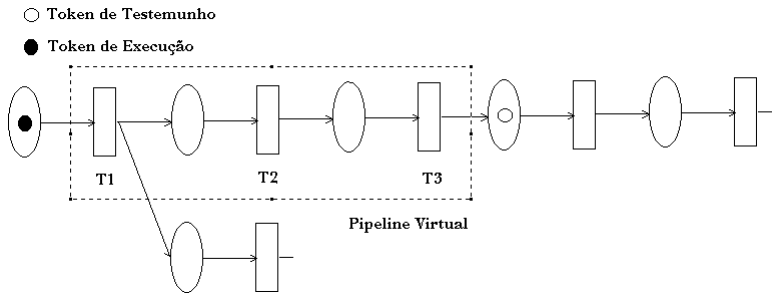

Figura 7.1: Modelo CPN-*Simplescalar*

plescalar, que não foi implementado no escopo desse trabalho, mas que caracteriza um interessante trabalho futuro.

7.2.2 Proposição de uma ADL-CPN

A atual característica multi-alvo do arcabouço de modelagem e análise ganhará maior flexibilidade e facilidades se a etapa de modelagem for implementada a partir de uma ADL *front end*. Uma vez que, no atual estágio da pesquisa, a descrição da arquitetura exige do projetista: (i) familiaridade com CPN e CPN-ML, para a criação do MCD e $MCPD$ da arquitetura alvo, (ii) familiaridade com a estrutura de classes do compilador Binário-CPN para reconfigura-lo para a arquitetura alvo. A Figura 7.2 ilustra o mecanismo de geração de modelos por meio de uma ADL *front end*, que será denominada aqui ADL-CPN. A aplicação de um pré-processamento na descrição ADL-CPN, deve gerar automaticamente tanto o conjunto de classes do compilador Binário-CPN como o MCD ou $MCPD$ da arquitetura. Adicionalmente, a estrutura de linguagem da ADL-CPN deve comportar a implementação de sondas de análise assim como é feito na descrição CPN.

7.2.3 Extensão CPN-C

A definição de uma extensão CPN-C dos modelos propostos compreende uma modelagem centrado nas estruturas de controle do C ANSI, no lugar das instruções do processador. Isso implica em um nível de abstração mais alto do modelo de potência e desempenho, eliminando a necessidade de um modelo funcional da arquitetura. O modelo funcional será substituído por modelos de estruturas C alimentados por macromodelos de consumo, criados a partir da caracterização das estruturas de controle do C ANSI. Sendo assim, um modelo CPN-C descreve a aplicação (MCA e $MCPA$) em termos das estruturas C, não mais em termos da cadeia de instruções do código binário. Os ganhos imediatos esperados são: compactação do modelo de aplicação, aumento da

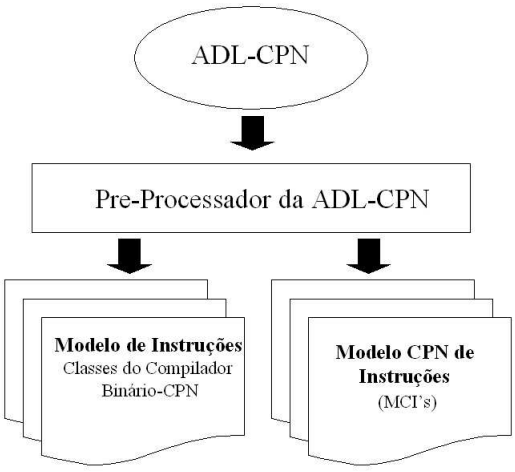

Figura 7.2: ADL-CPN: idéia principal.

velocidade de simulação e melhor elucidação dos centros de custos no código. A melhor elucidação dos centros de custos advém do fato da taxonomia proposta (*Patches*, *Clusters*, *Loop-Patches* etc) está agora relacionada ao código C, que é facilmente visualizado pelo projetista. Um desafio da pesquisa proposta é a criação de um modelo de consumo das estruturas C onde seja contemplado os efeitos decorrentes da estrutura do código, bem como das otimizações presentes nos compiladores C.

7.2.4 Síntese de Clusters em *Hardware*

Tendo em vista o paradigma *Hardware-Software Co-Design*, a síntese de *Clusters* em *hardware* aparece como uma conseqüência natural para a aplicação do arcabouço de análise proposto nesta tese. Com base nos resultados obtidos por Stitt *et al* em [93] e [92], conclui-se que a migração *software-hardware* é uma opção interessante para a redução do consumo global em sistemas embutidos. Uma seção de particionamento assistido pode ser integrada no arcabouço, por meio de um mecanismo de conversão de *Clusters* em uma descrição de *hardware* e instanciamento de processos de comunicação *software-hardware*. Transformações de estruturas seqüenciais em concorrentes podem ser naturalmente implementadas e validadas devido à semântica CPN e aos seus mecanismos de simulação. Adicionalmente, uma vez que há maior eficiência no processo de tradução C para HDL [93], particularmente para a linguagem Verilog, a extensão CPN-C estabelece perspectivas interessantes. O principal desafio de uma pesquisa como esta é a definição do modelo de mapeamento CPN-HDL.

7.2.5 Modelo de Instrução-CPN para Estimativa de Potência em *Hardware*

Conforme discutido na Seção 2.3.4, Givargis *et al* propuseram em [102] e [103] um modelo para a estimativa de consumo de energia em *hardware* baseado no conceito de *instrução*. Nesse modelo, elementos da descrição RTL do *hardware* são aglutinados funcionalmente e classificados como *instruções*, sendo feitas estimativas de consumo com base nos traços de execução das seqüências de instruções. A descrição das instruções de *hardware* em um modelo CPN de instrução, permitirá a criação de um modelo CPN de descrição de *hardware*, similarmente ao modelo CPN de descrição de *ISA* apresentado nesta tese. Dada a aplicação da semântica das redes de Petri coloridas apresentada nesta tese, tal modelo permitirá a análise de concorrência de instruções de *hardware*, bem como a implementação de extensões probabilísticas. O principal desafio de uma pesquisa como essa, será a definição de um mapeamento formal HDL-CPN que contemple o conceito de instrução[1]. Os mecanismos de simulação e avaliação de resultados devem herdar, em grande monta, as estruturas e conceitos desenvolvidos nesta tese.

7.2.6 Linguagem de Modelagem de EPS

Conforme discutido na Seção 5.2.2, o conjunto de especificações probabilísticas do sistema ($cEPS$), compreende uma percepção dos eventos do sistema, suas probabilidades e sua relação com os desvios de fluxo de execução da aplicação. Podendo ser construído mediante uma percepção *ad hoc*, conforme foi feito nesta tese. Tal abordagem é, contudo, muito difícil de se implementar em sistemas mais complexos. Surge, portanto, a necessidade de se estabelecer um mecanismo de descrição das especificações probabilísticas do sistema (EPS). Tal pesquisa compreende, antes de mais nada, a proposição de uma linguagem de descrição. O principal desafio consiste em estender métodos de descrição difundidos, como o UML (*Universal Modeling Language*) e o LSC (*Living State Chart*) de forma a suportarem o conceito de EPS.

[1]Nos trabalhos de Givargis *et al* o mapeamento é feito manualmente, da descrição HDL para um modelo *ad hoc* escrito em C++ ou Java.

Referências Bibliográficas

[1] *Xtensa LX Microprocessor: Orverview Hanbook*.

[2] At89s8252 datasheet. disponível em http://www.atmel.com/dyn/resources/prod_documents/doc0401.pdf, 2006.

[3] synopsys products & solutions. disponível em http://www.synopsys.com/products/solutions/galaxy/power/power.html, 2006.

[4] Nevine AbouGhazaleh, Bruce Childers, Daniel Mosse, Rami Melhem, and Matthew Craven. Energy management for real-time embedded applications with compiler support. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 284–293, New York, NY, USA, 2003. ACM Press.

[5] H. Akaboshi and H. Yasuura. Behavior extraction of mpu from hdl description. In *Asia Pacific Conference on Chip Design Languages (APChDL'94)*, 1994.

[6] T. Arnould and S. Tano. Interval-value fuzzy backward reasing. *IEEE Transaction on Fuzzy Systems*, 3:425–437, November 1995.

[7] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[8] Alexandro Baldassin, Paulo Centoducatte, and Sandro Rigo. Extending the archc language for automatic generation of assemblers. *17th International Symposium on Computer Architecture and High Performance Computing*, pages 60–67, October 2004.

[9] R. Barreto, S. Cavalcante, and P. Maciel. A time petri net approach for finding pre-runtime schedules in embedded real-time systems. In *1st Int. Workshop on Embedded Computing Systems (ECS'04) in conjunction with the 24th Int.Conf. on Distributed Computing Systems (ICDCS'04)*. IEEE CS Press, march 2004.

[10] R. Barreto, P. Maciel, and S. Cavalcante. A modeling methodology and pre-runtime scheduling for embedded real-time software. In *Proc. 15th Brazilian Symposium on Computer Architecture and High Performance Computing*, pages 72–79. IEEE CS Press, November 2003.

[11] Rodrigo Possamai Bastos, Fernanda Lima Kastensmidt, and Ricardo Reis. Designing low-power embedded software for mass-produced microprocessor by using a loop table in on-chip memory. In *PATMOS*, pages 59–68, 2005.

[12] A. Bona, M. Sami, D. Sciuto, V. Zaccaria, C. Silvano, and R. Zafalon. Energy estimation and optimization of embedded vliw processors based on instruction clustering. In *Proceedings of the 39th conference on Design automation*, pages 886–891. ACM Press, 2002.

[13] A. Bona, M. Sami, D. Sciuto, V. Zaccaria, C. Silvano, and R. Zafalon. An instruction-level methodology for power estimation and optimization of embedded vliw cores. In *Proceedings of the conference on Design, automation and test in Europe*, page 1128. IEEE Computer Society, 2002.

[14] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94. ACM Press, 2000.

[15] A. J. Bugarin and S. Barro. Fuzzy reasoning supported by petri nets. *IEEE Transaction on Fuzzy Systems*, 2:135–4150, May 1994.

[16] Richard Burch, Farid Najm, Ping Yang, and Timothy Trick. Mcpower: a monte carlo approach to power estimation. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 90–97, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[17] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 288. IEEE Computer Society, 1995.

[18] Frank Burns, Albert Koelmans, and Alexandre Yakovlev. Wcet analysis of superscalar processor using simulation with coloured petri nets. *International Journal of Time-Critical Computing Systems,*.

[19] Frank Burns, Albert Koelmans, and Alexandre Yakovlev. Modelling of superscala processor architectures with design/CPN. In Jensen, K., editor, *Daimi PB-532: Workshop on Practical Use of Coloured Petri Nets and Design/CPN, Aarhus, Denmark, 10-12 June 1998*, pages 15–30. Aarhus University, 1998.

[20] Christos G. Cassandras and Stéphane Lafortube. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, November 1999.

[21] Alexander Chatzigeorgiou and George Sthepanides. Energy metric for software systems. *Software Quality Journal,*.

[22] Shyi-Ming Chen. A methodology for knowledge base verification using fuzzy petri nets. *Int. Journal Inf. Manage*, 4(2):119–135, December 1993.

[23] Shyi-Ming Chen. Fuzzy backward reasoning using fuzzy petri nets. *IEEE Transaction on Systems , Man and Cybernetics*, 30, Part B:846–856, Octuber 2000.

[24] Shyi-Ming Chen, J. S. Ke, and J. F. Chang. Knowledge representation using fuzzy petri nets. *IEEE Trans. Knowl. Data Eng.*, 2:311–319, september 1990.

[25] T. Cignetti, K. Komarov, and C. Ellis. Energy estimation tools for the palm, 2000.

[26] F. Commoner. *Deadlock in Petri Nets*. Applied Data Reasearch Inc., 1972.

[27] Meta Software Corporation. *Design/CPN-Manual*. , address =.

[28] Srinivas Devadas and Sharad Malik. A survey of optimization techniques targeting low power vlsi circuits. In *Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 242–247. ACM Press, 1995.

[29] Jakob Engblom. *Processor Pipeline and Static Worst-Case Execution Time Analysis*. PhD thesis, Acta Universtatis Upsalienssis, 2002.

[30] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *European Design and Test Conference (ED&TC '95)*, pages 503–507, 1995.

[31] F. C. Filho, P. Maciel, and E. Barros. Using petri nets for data dependency analysis. In *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC'2000), 8-11 October 2000, Nashville, TN*, volume 4, pages 2998–3003, 2000.

[32] William Fornaciari, Paolo Gubian, Donatella Sciuto, and Cristina Silvano. Vhdl-based approach for power estimation of embedded systems. *Journal of Systems Architecture*, 44(1):37–61, October 1997.

[33] Meimei Gao, Zhiming Wu, and Megchu Zhou. A petri net-based fromal reasoning algorihtm for fuzzy production rule. *IEEE Transaction on Systems , Man and Cybernetics*, pages 3093–3097, Octuber 2000.

[34] M. Hack. *Analysis of Production Schemata by Petri Nets.* MSc Thesis, Department of Electrical Engineering of Massachusett Institute of Technology, Februaray 1972.

[35] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302, 1997.

[36] A. Halambi and P. Grun. Expression: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the European Conference on Design, Automation and Test (DATE)*, 1999.

[37] A. Halambi, P. Grun, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic software toolkit generation for embedded systems-on-chip, 1999.

[38] Peter Huber, Kurt Jensen, and Robert M. Shapiro. Hierarchies in coloured petri nets. In *APN 90: Proceedings on Advances in Petri nets 1990*, pages 313–341, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[39] K. Jensen. An introduction to the theoretical aspects of coloured petri nets. *Lecture Notes in Computer Science: A Decade of Concurrency*, 803:230–272, 1994.

[40] Kurt Jensen. An introduction to the practical use of coloured petri nets. In *Petri Nets (2)*, pages 237–292, 1996.

[41] Kurt Jensen. A brief introduction to coloured petri nets. *Tools and Algorithms for the Construction and Analysis of Systems. Proceeding of the TACAS'97 Workshop*, 1217(2):203–208, 1997.

[42] A. Arcoverde Jr, G. Alves Jr, R. Lima, P. Maciel, M. Oliveira Jr, and R. Barreto. Ezpetri: A petri net interchange framework for eclipse based on pnml. In *In First Int. Symp. on Leveraging Applications of Formal Method (ISoLA'04)*, 2004.

[43] Nathalie Julien, Johann Laurent, Eric Senn, and Eric Martin. Power consumption estimation of a c program for data-intensive applications. In *PATMOS 2002*, volume 2451, pages 332–341. LNCS Kluwer Academic Pubishers, September 2002.

[44] Nathalie Julien, Johann Laurent, Eric Senn, and Eric Martin. Power estimation of a c algorithm based on the functional-level power analysis of a digital signal processor. In *High Performance Computing, 4th International Symposium, ISHPC*, volume 2327, pages 354–360. LNCS Kluwer Academic Pubishers, May 2002.

[45] M. Nogueira Oliveira Júnior. *Desenvolvimento de Um Protótipo para a Medida Não Invasiva da Saturação Arterial de Oxigênio em Humanos - Oxímetro de Pulso (in portuguese).* MSc Thesis, Departamento de Biofísica e Radiobiologia, Universidade Federal de Pernambuco, August 1998.

[46] Meuse N.O. Junior, Paulo Maciel, Raimundo Barreto, and Fernando Carvalho. A software power cost analysis based on colored petri net. In *ToBaCo in ATPN 2004*, pages 33–42, June 2004.

[47] B. Klass, D.Thomas, H. Schmit, and D. Nagle. Modeling inter-instruction energy effects in a digital signal processor. In *Power-Driven Microarchitecture Workshop In conjunstion with ISCA98*, 1998.

[48] Samir M. Koreim. A fuzzy petri net tool for modeling and verification of knowledge-based sytems. *The Computer Journal*, 43(3):206–223, November 2000.

[49] Paul Landman. High-level power estimation. In *Proceedings of the 1996 international symposium on Low power electronics and design*, pages 29–35. IEEE Press, 1996.

[50] Th. Laopoulos, P. Neofotistos, C. Kosmatopoulos, and S. Nikolaidis. Current variations measurements for the estimation of software-related power consumption. *IEEE Instrumentation and Measurement Technology Conference*, May 2002.

[51] Chingren Lee, Jenq Kuen Lee, TingTing Hwang, and Shi-Chun Tsai. Compiler optimization on instruction scheduling for low power. In *Proceedings of the 13th international symposium on System synthesis*, pages 55–60. IEEE Computer Society, 2000.

[52] Mike Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. Power analysis and minimization techniques for embedded dsp software. *IEEE Transactions on Very Large Scale Integration Systems*, pages 123–135, March 1997.

[53] Jim Lipman. Soc embedded software needs a low-power perspective. *TechOnline http://www.techonline.com*, 2002.

[54] R. J. Lipton. The reachability problem requires exponential space. Research report 62, Department of Computer Science. Yale University, January 1976.

[55] Lee Lloyd, Alex V. Yakovlev, Enric Pastor, and Albert M. Koelmans. Estimation of power consumption in asynchronous logic as derived from graph based circuit representations. In Anne-Marie Trullemans-Anckaert and Jens Sparsø, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 367–376, 1998.

[56] Carl G. Looney. Fuzzy petri nets for rules-based decisionmaking. *IEEE Transaction on Systems , Man and Cybernetics*, 18(1):178–183, January 1988.

[57] P. Maciel. *Petri Net Based Estimators for Hardware/Sofware Codesign.* PhD Thesis, Centro de Informática. Universidade Federal de Pernambuco, Dec 1999.

[58] P. R. M. Maciel, R. D. Lins, and P. R. F. Cunha. *Introdução às Rede de Petri e Aplicação.* X Escola de Computação- Campinas SP, Centro de Informática. Universidade Federal de Pernambuco, Julho 1996.

[59] Paulo Maciel, Edna Barros, and Wolfgang Rosenstiel. A petri net model for hardware/software codesign. In *Design Automation for Embedded Systems*, volume 4, pages 243–310, 1999.

[60] Sujit Dey Marcello Lajolo, Anand Raghunathan and Luciano Lavagno. Cosimulation-based power estimation for syste-on-chip design. *IEEE Transactions on Very Large Scale Integration (VLSI) System*, 10(3):253–266, June 2002.

[61] A. Marsan and G. Chiola. On petri nets with deterministic and exponentially distributed firing times. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 266 of *Lecture Notes in Computer Science*, pages 132–145. Springer-Verlag, 1987.

[62] M. Marsan, A. Bobbio, and D. Donatelli. Petri nets in performance analysis: An introduction. *LNCS: Lectures on Petri Nets I: Basic Models*, 1491:211–256, June 1998.

[63] M. Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with Generalised Stochastic Petri Nets*. John Wiley & Sons Ltd, November 1995.

[64] Marco Ajmone Marsan, Gianfranco Balbo, Andrea Bobbio, Giovanni Chiola, Gianni Conte, and Aldo Cumani. On petri nets with stochastic timing. In *International Workshop on Timed Petri Nets*, pages 80–87, Washington, DC, USA, 1985. IEEE Computer Society.

[65] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122, 1984.

[66] R. Mehra and J. Rabaey. Behavioral level power estimation and exploration. *Proc. Int. Workshop on Low Power Des.*, pages 197–202, 1994.

[67] Paul L. Meyer. *Probabilidades , Aplicaçãoes à Estatística*. Livros Técnicos e Científicos Editora SA, 1978.

[68] Prabhat Mishra, Arun Kejariwal, and Nikil Dutt. Synthesis-driven exploration of pipelined embedded processors. In *International Conference on VLSI Design*, pages 921–926, 2004.

[69] Wai Sum Mong and Jianwen Zhu. A retargetable micro-architecture simulator. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 752–757, New York, NY, USA, 2003. ACM Press.

[70] T. Morimoto, K. Saito, H. Nakamura, T. Boku, and K. Nakazawa. Advanced processor design using hardware description language AIDL, 1997.

[71] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[72] Ashok K. Murugavel and N. Ranganathan. Petri net modeling of gate and interconnect delays for power estimation. In *Proceedings of the 39th conference on Design automation*, pages 455–460. ACM Press, 2002.

[73] Ashok K. Murugavel and N. Ranganathan. Power estimation of sequential circuits using hierarchical colored hardware petri net modeling. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*, pages 267–270, New York, NY, USA, 2002. ACM Press.

[74] Kshirasagar Naik and David S. L. Wei. Software implementation strategies for power-conscious systems. *MONET*, 6(3):291–305, 2001.

[75] Farid N. Najm. A survey of power estimation techniques in vlsi circuits. *IEEE Trans. Very Large Scale Integr. Syst.*, 2(4):446–455, 1994.

[76] Farid N. Najm. Towards a high-level power estimation capability. In *Proceedings of the 1995 international symposium on Low power design*, pages 87–92. ACM Press, 1995.

[77] Paulo Nascimento, Manoel Lima, Paulo Maciel, Abel Filho, Edna Barros, and Sérgio Cavalcante. Cdfg - petri net temporal partitioning for switching context applications. In *15th Symposium on Integrated Circuits and System Design - SBCCI 2002*. IEEE Press, 2002.

[78] S. Nikolaidis, N. Kavvadias, and P. Neofotistos. Instruction-level power measurement methodology. Technical report, Electronics Lab., Physics Dept., Aristotle University of Thessaloniki, Greece, March 2002.

[79] S. Nikolaidis, N. Kavvadias, and P. Neofotistos. Instruction-level power models for embedded processors. Technical report, Electronics Lab., Physics Dept., Aristotle University of Thessaloniki, Greece, December 2002.

[80] A. Parikh, M.Kandemir, and N. Vijaykrishnan. Instruction scheduling based on energy and performance constraints. In *Proceedings of IEEE Computer Siciety Workshop on VLSI*, pages 37–42. IEEE Computer Society Press, 2000.

[81] Ferdinand Peper, Jia Lee, Susumu Adachi, and Teijiro Isokawa. Token-based computing on nanometers scales. In *ToBaCo in ATPN 2004*, pages 01–20, June 2004.

[82] Peter Petrov and Alex Orailoglu. Compiler-based register name adjustment for low-power embedded processors. In *ICCAD*, pages 523–528, 2003.

[83] Wei Qin. *Modeling and Description of Embedded Processors for the Development of Software Tools*. PhD Thesis, Department of Electrical Engineering University of Princeton, November 2004.

[84] Sandro Rigo, Guido Araujo, Marcus Bartholomeu, and Rodolfo Azevedo. Archc: A systemc-based architecture description language. In *SBAC-PAD*, pages 66–73, 2004.

[85] Chris Rowen. Reducing soc simulation and development time. *Computer*, 35(12):29–34, 2002.

[86] Rozenberg. Advances in petri nets. In *APN 90: Proceedings on Advances in Petri nets 1990*, pages 313–341, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[87] Jeffrey Russell and Margarida Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. *International Conference on Computer Design (ICCD)*, October 1998.

[88] Alberto Sangiovanni-Vincentelli and Grant Martin. A vision for embedded software. In *CASES'01*. IEEE Press, November.

[89] Davide Sarta, Dario Trifone, and Giuseppe Ascia. A data dependent approach to instruction level power estimation. In *Proceedings of the IEEE Alessandro Volta Memorial Workshop on Low-Power Design*, page 182. IEEE Computer Society, 1999.

[90] Eric Senn, Nathalie Julien, Johann Laurent, and Eric Martin. Softexplorer: Estimation, characterization, and optimization of the power and energy consumption at the algorithmic level. In *PATMOS 2004*, volume 3254, pages 342–351. LNCS Kluwer Academic Pubishers, September 2004.

[91] Manuel Silva, Enrique Teruel, Robert Valette, and Hervé Pingaud. Petri nets and production systems. In *Lectures on Concurrency and Petri Nets*, volume 1492 of *Lecture Notes in Computer Science*, pages 85–125. Springer, 1998.

[92] Greg Stitt and Frank Vahid. Energy advantages of microprocessor platforms with on-chip configurable logic. *IEEE Design and Test of Computers*, 19(6):36–43, Nov/Dec 2002.

[93] Greg Stitt and Frank Vahid. Hardware/software partitioning of software binaries. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 164–170. ACM Press, 2002.

[94] Greg Stitt, Frank Vahid, Tony Givargis, and Roman Lysecky. A first-step towards an architecture tuning methodology for low power. In *Proceedings of the international conference on Compilers, architectures, and synthesis for embedded systems*, pages 187–192. ACM Press, 2000.

[95] C. L. Su, C. Y. Tsui, and A. M. Despain. Low power architecture design and compilation technique for high-performance processors. In *Proc. IEEE COMPCON*, pages 209–214, 1994.

[96] Luca Benini Tajana Simunic and Giovanni De Micheli. Energy-efficient design of battery-powered embedded system. *International Symposium on Low Power Electronics and Design*, pages 49–54, July 2000.

[97] Eduardo Tavares, Paulo Maciel, Meuse N. O. junior, and Raimundo Barreto. An approach for finding pre-runtime scheduling in embedded real-time systems with power contraint. *16th Symposium on Computer Architecture and High Performance Computing*, October 2004.

[98] Vivek Tiwari and Mike Lee. Power analysis of a 32-bit embedded microcontroller. *VLSI Design Journal*, 7(3), 1998.

[99] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration Systems*, 2(4):437–445, December 1994.

[100] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. *J. VLSI Signal Process. Syst.*, 13(2-3):223–238, 1996.

[101] H. Tomiyama, A. Halambi, P. Grün, N.D. Dutt, and A. Nicolau. Architectural description languages for systems-on-chip design. In *Asia Pacific Conference on Chip Design Languages (APChDL'99)*, 1999.

[102] Frank Vahid Tony Givargis and Jorg Henkel. Trace-driven system-level power evaluation of system-on-a-chip peripheral cores. *Asia South-Pacific Design Automation Conference*, pages 306–311, January 2001.

[103] Frank Vahid Tony Givargis and Jorg Henkel. Instruction-based system-level power evaluation of system-on-chip peripheral cores. *IEEE Transaction on Very Large Scale Integration Systems*, 10(6):856–863, December 2002.

[104] Rüdiger Valk. Object petri nets: Using the nets-whtin-nets paradigm. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 819–848. Springer, 2004.

[105] A. Valmari. Compositional state space generation. *LNCS: Advances in Petri Nets*, 674:427–457, 1993.

[106] Antti Valmari. Petri nets and production systems. In *Lectures on Concurrency and Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–529. Springer, 1998.

[107] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 95–106. ACM Press, 2000.

[108] Alex Yakovlev. Is the die cast for the token game? In *Lecture Notes in Computer Science: 23rd International Conference on Applications and Theory of Petri Nets, Adelaide, Australia, June 24-30, 2002 / J. Esparza, C. Lakos (Eds.)*, volume 2360, pages 1–70pp. Springer Verlag, June 2002. InternalNote: Submitted by: hr.

[109] Hongbo Yang, Guang R. Gao, and Clement Leung. On achieving balanced power consumption in software pipelined loops. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 210–217, New York, NY, USA, 2002. ACM Press.

[110] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of the 37th conference on Design automation*, pages 340–345. ACM Press, 2000.

[111] Gary Yeap. *Practical Low Power Digital VLSI Design.* Kluwer Academic Publishers, 1998.

[112] H.C. Yen. Unified approach for deciding the existence of certain petri net path. *Information and Computaion*, 1992.

[113] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa - machine description language and generic machine model for hw/sw co-design. In *VLSI Signal Processing IX*, 1996.

Apêndice A

Modelo de Potência: Caracterizando o AT89S8252

A.1 Introdução

Esta tese usou como objeto para a criação dos seus modelos a arquitetura i8051, sendo o micro-controlador AT89S8252 o dispositivo alvo no processo de validação dos modelos propostos. Esse micro-controlador foi escolhido devido à sua grande penetração no mercado de sistemas embutidos e por permitir atualização do seu *firmware* por meio de comunicação SPI (*Serial Peripheral Interfacing*). Dessa forma, um protocolo automático de medidas pôde ser estabelecido pelo uso de computador PC e um sistema de aquisição, neste trabalho um osciloscópio digital cumpriu o papel de sistema de aquisição. Este capítulo descreve o modelo de potência de instruções criado a partir da caracterização desse dispositivo. As seções seguintes descrevem: a metodologia de caracterização empregada, o mecanismo de caracterização automática e o modelo de potência de instrução resultante.

A.2 Metodologia de Caracterização

A caracterização foi realizada por meio da medida de tensão sobre um resistor colocado em série com o dispositivo. As medidas foram realizadas usando um osciloscópio digital Fluke da série 960 - banda de 100MHz e máxima freqüência de amostragem em 1GSa/s - como sistema de aquisição de dados. Tal processo reproduziu basicamente as técnicas apresentadas em [99], [87] e [25]. O dispositivo foi caracterizado em uma placa especialmente construída para este fim, de forma a garantir completo isolamento com relação a periféricos. A caracterização buscou estabelecer um modelo de consumo

por instrução do sistema micro-controlador, integrando o consumo de energia devido ao acesso à memória ao modelo de consumo de instrução. Dessa forma, esse modelo foca a utilização do micro-controlador em sua forma mais essencial, sem memórias externas ou dispositivos que demandem corrente de suas portas. O processo de medida objetivou a geração de subsídios para o modelamento do consumo por instrução e do consumo inter-instrução. De forma a prevenir a criação de conjuntos muito grandes de códigos de teste para a avaliação do custo inter-instrução[1], foram criados primeiramente códigos para a avaliação do custo por instrução. Com base nessa avaliação, as instruções foram agrupadas por consumo e o custo inter-instrução foi modelado como o custo inter-grupos representado pelo custo inter-instrução de um par de instruções representativo dos grupos de interesse. As instruções foram caracterizadas por meio de códigos de caracterização implementados para executar a instrução alvo em um laço infinito.

O código foi escrito de forma que a condição de execução da instrução fosse exatamente a mesma para todas as execuções. As instruções foram divididas em cinco categorias: aritméticas, booleanas, de desvio, de transferência e lógicas. Para cada instrução, foi gerado um subconjunto de códigos de caracterização explorando diferentes condições dos operandos. Os códigos de teste, dentro desses subconjuntos, exploraram as condições de máximo e mínimo número de transições dos bits dos registradores/memórias afetados pela instrução. A Figura A.1 mostra o código de caracterização para a instrução $ADDCA, @R_0$, na qual não há transições no registrador A[2]. O código de teste gera um marcador da posição da instrução no tempo de execução, por meio de um sinal marcador enviado no bit 0 da porta 1 (P1.0). O osciloscópio é conectado à placa de teste de forma que o seu canal A captura o sinal do resistor e o canal B captura o sinal marcador na porta P1.0.

Objetivando a normalização do efeito inter-instrução nessas medidas, os códigos executam uma instrução NOP (*No OPeration*) antes da instrução alvo. Dessa forma, todos os custos inter-instrução presentes nas medidas estão referenciados a instrução NOP. Devido à essa abordagem foram gerados 589 códigos de caracterização, a Tabela A.1 apresenta o número de códigos de caracterização por categoria de instrução.

Com base nos resultados obtidos foram identificadas aglutinações de instruções com valores de consumo similares. Foi escolhido, arbitrariamente, uma instrução por aglutinação, como representante da aglutinação. Foram realizadas medidas manuais

[1] A problemática da avaliação e modelos referentes aos custos inter-instrução foram apresentados no Capítulo 2.

[2] Vide comentários no cabeçalho do código.

```
;*************************************************************
;*    Códigos para Caracterização de Consumo de Energia      *
;*         por instrução na arquitetura 8051                 *
;*  Instrução: ADDC A,@r0 (A= 00H,(r0)=00H) minima dist H    *
;*************************************************************
org 0H
     mov IE,#00H    ;* Desabilita todas as interrupções
     mov psw,#00H   ;* Zera os flags
     mov a,#00H     ;* Garante valor inicial A=00H
     mov 30H,#00H   ;* Garante valor inicial (30H)=00H
     mov r0,#30H    ;* Faz r0n apontar para 30H
inicio:             ;* Marca loop de analise

     clr p1.0       ;* Reseta  marcador de sincronização do
                    ;* osciloscópio

;*************************************************************
;* Instrução de 12 ciclos de clock (1us).                    *
;* Observar apos 1us a partir da borda de descida            *
;* do marcador                                               *
;*************************************************************
;* INSTRUÇÃO:
     nop            ;* NOP padrão
     addc a,@r0     ;* Distância de  H = 0 bits

;*************************************************************

     setb p1.0      ;* Seta  marcador de sincronização do
                    ;* osciloscópio
     mov psw,#00H   ;* Zera os flags
     mov a,#00H     ;* Garante valor inicial A=00H
     mov 30H,#00H   ;* Garante valor inicial (30H)=00H
     mov r0,#30H    ;* Faz r0 apontar para 30H


     jmp inicio     ;* Executa loop "dentro do gate marcador"
```

Figura A.1: Exemplo de código de caracterização.

do custo inter-instrução dos pares formados pelas instruções representantes de cada aglutinação. Essa abordagem foi assumida com base nos resultados obtidos por Bona *et al* [12] avaliando arquiteturas VLIW. Eles concluíram que as instruções poderiam ser aglutinadas em grupos cujo consumo é muito próximo, criando um modelo baseado em classes de consumo, a Figura A.2 mostra os resultados de Bona *et al.* Dada a maior regularidade da arquitetura i8051, assumiu-se que tais classes também poderiam ser identificadas no i8051. Os resultados comprovaram essa hipótese.

Tabela A.1: Categorias de códigos de caracterização

Categoria	Número de Códigos de Caracterização
Aritmética	126
Booleana	56
De Desvio	158
De Transferência	153
Lógica	96

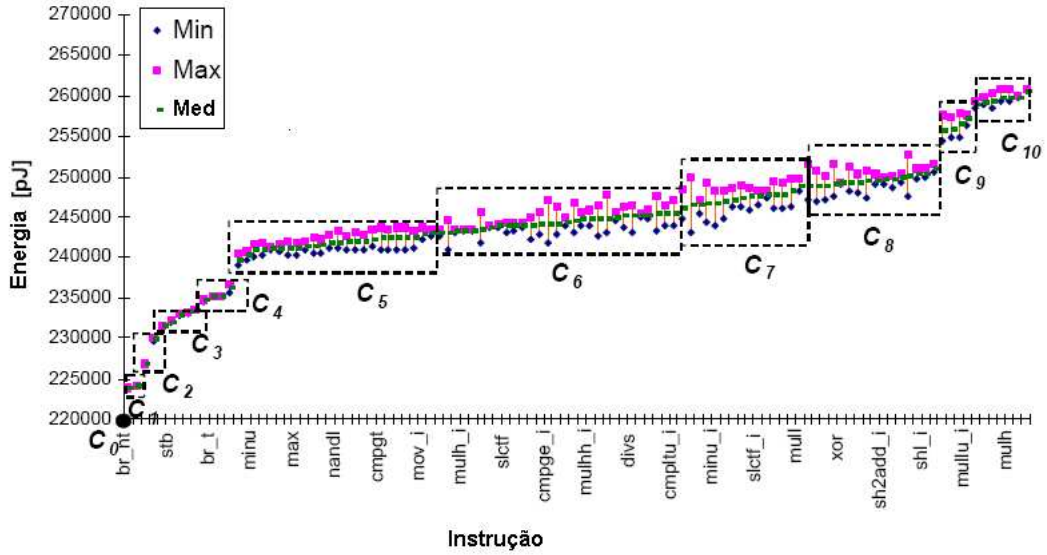

Figura A.2: Associação do consumo de instruções em *Clusters*(Adaptado de [12])

A.3 Sistema de Caracterização Automática

Objetivando a realização das 589 medidas de forma rápida e segura, um sistema de caracterização automática foi desenvolvido, a Figura A.3 ilustra tal sistema. O sistema é constituído por um computador PC-compatível, uma placa de teste com o dispositivo alvo e um osciloscópio digital. Uma aplicação Java comanda o sistema operando sobre três módulos básicos: programação e verificação, configuração e aquisição, e interface com o usuário. O módulo de programação e verificação implementa a gravação dos códigos no micro-controlador. Para tal, foi aplicado o protocolo de gravação SPI [2] usando como interface física a porta paralela do PC. A porta paralela foi utilizada por permitir uma interface elétrica compatível com o micro-controlador (interface TTL). Como o mecanismo de comunicação SPI é serial, o aplicativo realiza operações de mascaramento para a transmissão dos dados. O módulo de programação opera sobre o código de caracterização no formato HEX[3]. A gravação é feita linha a linha, após a gravação de cada linha é realizada uma leitura dos dados gravados verificando-se a consistência da gravação. Caso haja erros, o aplicativo realiza mais um ciclo gravação-verificação para a dada linha. Se ocorrerem três erros consecutivos, o aplicativo levanta uma exceção que resulta na paralisação do processo, e em uma mensagem na interface

[3]Arquivo ASCII com a representação hexadecimal do código de máquina. Esse tipo de arquivo é organizado por linhas e trás campos específicos indicando posições de memória de alocação de código e mecanismo de *checksum*

do usuário. Após a gravação do código, o aplicativo libera o micro-controlador para a execução e inicia a operação do módulo de configuração e aquisição. Esse módulo é responsável por comandar o osciloscópio, fazendo isso através da porta serial do PC, aplicando o protocolo de comunicação do osciloscópio. Antes do começo do processo de caracterização ele configura a velocidade de comunicação e trava o teclado. Os canais do osciloscópio são lidos como vetores de dados e apresentados na interface do usuário. Tais vetores são processados de forma a se extrairem parâmetros como: tempo de execução da instrução, consumo de energia e potência. Para cada medida, é gerado um arquivo no formato XML com os parâmetros extraídos e com os vetores de dados capturados do osciloscópio.

Figura A.3: Estrutura do processo de caracterização automática

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<instrucoes>
<instrucao intervalo="9.980000000000002E-7"
Tensao="0.48626912499999986" Potencia="0.043037018883759486"
Energia="4.295094484599198E-8"
VetorA="[0.0,0.3609375,0.35775,0.34268750000000003,0.3298125,0.31975000
000000003,0.3078125,0.3190625,0.3440625,0.3623125,0.40650000000000003,0
.4533125,0.4888125,0.5406875,0.57775,0.602125,0.6391875,0.6541875,0.638
5000000000001,0.6315625,0.6144375,0.5835625,0.5658125,0.55,0.542625,0.5
4025,0.5369375,0.532125,0.5136875,0.4919375,0.47725,0.4541875,0.4254375
,0.4089375,0.39575,0.3773125,0.3666875,0.3661875,0.36,0.3641875,0.37587
5,0.3735625,0.3765,0.382750000000000003,0.3710625,0.35325,0.3443125,0.33
69375,0.337,0.3520625,0.3721875,0.39975,0.440875,0.489,0.5300625,0.5704
375,0.61925,0.6521250000000001,0.666,0.6724375,0.6575,0.6235,0.5926875,
```

Figura A.4: Amostra de arquivo gerado pelo caracterizador automático.

A Figura A.5 mostra a interface do sistema de caracterização automática. O módulo de interface possui cinco comandos básicos: *iniciar, configuração, FFT, perfil de consumo e perfil de consumo por grupo.* O botão *configuração* permite que o usuário configure o diretório de origem dos códigos de caracterização. Os botões *perfil de consumo e perfil de consumo por grupo* permitem a análise dos resultado da caracterização.

O botão *FFT* permite que seja feita uma transformada rápida de Fourier nos vetores capturados, a idéia é permitir a realização de análises no domínio da freqüência dos vetores de medidas. Tal recurso, embora esteja implementado, não é utilizado neste trabalho, estando disponibilizado para trabalhos futuros (vide Capítulo 8).

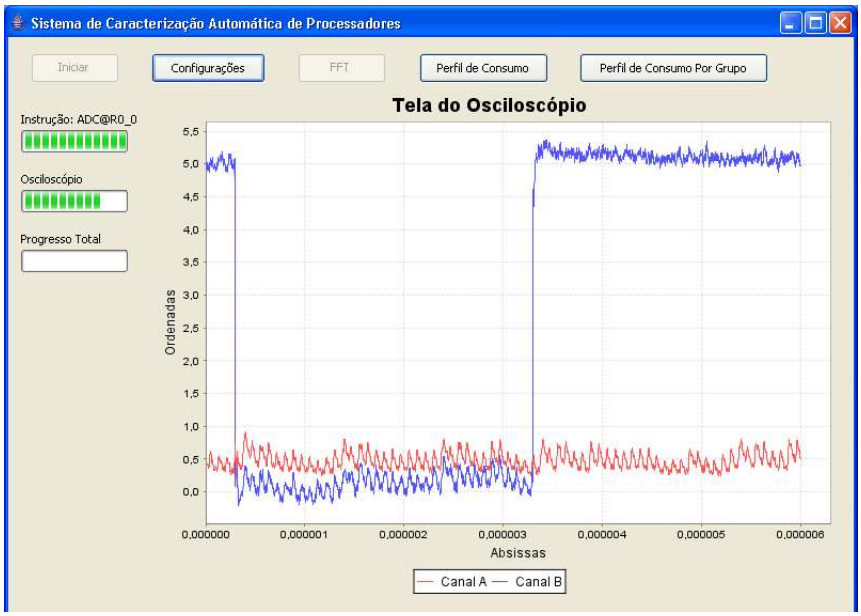

Figura A.5: Interface do sistema de caracterização automática.

O botão *perfil de consumo* aciona um menu suspenso que permite que o usuário escolha a visualização dos resultados por parâmetro medido (energia ou potência), categoria de instrução e por tempo de execução. A Figura A.6 mostra as telas do aplicativo quando analisando os resultados de consumo de energia das instruções booleanas que executam em 1μs. Os resultados são apresentados como um histograma, de forma que o usuário pode rolar o gráfico para a direita ou para esquerda por meio dos botões *Anterior* e *Próximo*. Recursos de medidas (*tooltips*) e aproximação (*zoom*) estão disponíveis de forma que, por comandos no *mouse* sobre o gráfico, o usuário ler a medida da instrução e amplia as escalas. Identicamente, o botão *perfil de consumo por grupo* permite a visualização dos resultados em termos dos valores médios das medidas nas categorias de instrução. A visualização é organizada por tempos de execução, assim pode-se visualizar os valores médios das instruções de 1μs, 2μs ou 4μs; por categoria de instrução. A Figura A.7 mostra a tela gerada pelo aplicativo para a visualização das instruções de 1us.

Embora tenha sido concebido especificamente para a caracterização do AT89S8252, o sistema mostra-se potencialmente aplicável à caracterização de quaisquer processador, micro-controlador ou *core* em FPGA (*Field Programmable Gate Array*). O aplicativo

Figura A.6: Análise de consumo por instrução.

Figura A.7: Análise de consumo médio por grupo de instrução.

permite a inclusão imediata de novos protocolos de gravação de dispositivos e aquisição de medidas. Adicionalmente, o caracterizador foi modificado de forma a ser utilizado nos testes de consistência em *hardware* apresentados no Capítulo 7. Os resultados foram satisfatórios, particularmente com relação à medida dos tempos de execução. As

medidas de energia e potência apresentaram erros maiores devido ao efeito *aliasing*[4] decorrente da redução da taxa de amostragem em função da base de tempo utilizada no osciloscópio. O impacto das questões metrológicas no processo de caracterização é discutido na seção seguinte.

A.4 Questões Metrológicas

Com relação à técnica de medida, a implementação do sistema caracterizador teve de lidar com duas questões básicas: a definição do resistor sensor e a adequação da taxa de amostragem do osciloscópio nas medidas específicas. Com relação ao resistor, sua definição foi realizada empiricamente por meio da avaliação de diversas opções. A questão básica por trás de escolha do resistor está no fato de se determinar seu valor ótimo. O resistor deve afetar o mínimo possível a tensão de alimentação do dispositivo, ao mesmo tempo não pode ser baixo ao ponto do ruído ambiente mascarar o sinal. Adicionalmente, o uso de resitores espiralados em corpo cerâmico implica na inserção de uma reatância indutiva no circuito, fazendo com que o fator de qualidade do circuito RL resultante impacte na definição do sinal adquirido. Em resistores espiralados, o fator qualidade tende a piorar em baixas resistências, dada a redução da relação $R/\omega L$. Após a análise de diversos valores foi utilizado um resistor de filme metálico com 51Ω. Tal valor posicionou as medidas na faixa de 500mV, o que proporcionou excelente relação sinal/ruído para a medida. A queda de 500mV na alimentação do dispositivo está perfeitamente dentro da faixa de operações práticas do dispositivo que permite tensões na faixa de 4,5V a 6,0V, sendo 5,0V a tensão nominal recomendada.

Com relação à configuração do osciloscópio, todas as medidas foram realizadas em modo tempo real, sem que os modos de cálculo de valor médio fossem ativados. As medidas foram feitas dessa forma visando observar flutuações estatísticas no consumo enquanto caracterizando uma instrução. Como resultado, observou-se que o consumo é perfeitamente estável, não apresentando alterações significativas entre diversas execuções de uma mesma instrução. Embora o osciloscópio em questão opere em tempo real com taxa de amostragem nominal de 1GSa/s, sua baixa profundidade de memória, 2400 pontos, força a taxa de amostragem efetiva a ser gradativamente reduzida quando se utiliza bases de tempo superiores da 200ns/div. A Figura A.8 ilustra esse efeito comparando a curva de decaimento da taxa de amostragem de um

[4]Efeito de sobreposição da informação de mais alta freqüência presente nos componentes harmônicos da freqüência de amostragem. Esse efeito faz com que a informação de alta freqüência seja perdida e um "sósia" (*alias*) de baixa freqüência do sinal amostrado seja gerado.

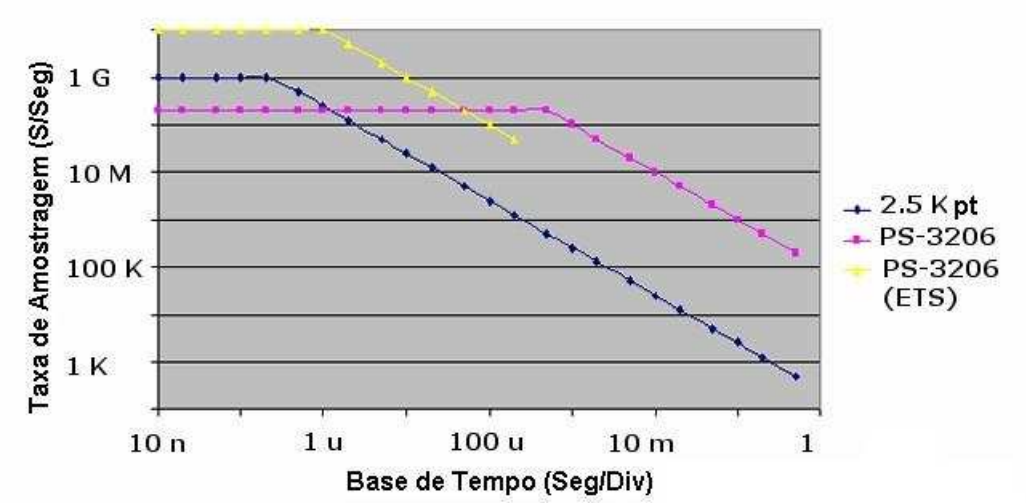

Figura A.8: Efeito da limitação de memória na redução da taxa de amostragem.

osciloscópio de 2500 pontos de memória com um sistema de aquisição com profundidade de memória superior (PS-3206), e com sistema de operação em tempo equivalente (PS-3206-ETS). Todas as medidas foram realizadas com o osciloscópio operando com base de tempo ajustada para 500ns/div, objetivando uniformizar a exatidão da medida para todo o experimento de caracterização. Isso implicou em medidas a uma taxa de amostragem de 500MSa/s. O efeito de redução da taxa de amostragem foi impactante nos experimentos de consistência em *hardware* apresentados no Capítulo 7. O Experimento 2 foi realizado com base de tempo de 100μ/div, devido à seu tempo de execução máximo de 831μs. Resultando em uma taxa de amostragem efetiva de 2MSa/s, sendo essa taxa exatamente o limite teórico para prevenção de *aliasing*. [5] O Experimento 3 apresentou tempo de execução da ordem de 3ms, estabelecendo o uso da base de tempo de 500μs/div. Como resultado a taxa de amostragem efetiva foi de 16KSa/s, de forma que o efeito *aliasing* eliminou a resolução da medida em termos de tempos de instrução. Cabe, contudo, enfatizar que a medida de tempo de execução sofreu apenas um pequeno erro de quantização devido à taxa de amostragem, sendo de 0,060% para o Experimento 2 e 0,078% para o Experimento 3.

[5]Uma vez que o menor tempo de execução de instrução é de 1μs, significando freqüência máxima de execução de instrução de 1MHz ($f_{max} = 1MHz$). O teorema de Nyquist estabelece uma taxa de amostragem mínima de $2 * f_{max}$, ou seja 2MSa/s

A.5 Modelo do AT89S8252

Os resultados mostraram que o consumo de energia das instruções no AT89S8252 está fortemente relacionado à categoria de instrução e ao seu tempo de execução. A variação do consumo dentro de um grupo de instruções, pertencentes a uma mesma categoria e com mesmo tempo de execução, é inferior a 0,02%. Essa variação, dado o seu baixo valor, é mais facilmente associada às flutuações da medida do que ao real comportamento do dispositivo. Como pode ser observado na Figura A.6, as variações dos operandos também não afetam visivelmente o padrão de consumo das instruções. Frente a esses fatos, o modelo de consumo por instrução pode ser resumido aos valores de consumo associados aos grupos de instruções pertencentes a uma mesma categoria e com mesmo tempo de execução. A Figura A.7 apresenta o consumo médio para as instruções de 1μs em função de suas categorias. Adicionalmente, a avaliação do custo inter-grupo mostrou que o efeito inter-instrução no AT89S8252 pode ser modelado como nulo. A Tabela A.2 resume o modelo de consumo de energia por grupo de instruções.

Tabela A.2: Modelo de consumo de energia por instrução do AT89S8252

Grupos		Energia	Potência
Tempo de Execução	Categoria	(nJ)	(mW)
1μs	Aritmética	40,466	40,466
	Booleana	42,951	42,951
	De Transferência	31,184	31,184
	Lógica	40,022	40,022
2μs	Aritmética	86,047	43,023
	Booleana	86,046	43,023
	De Desvio	86,046	43,023
	De Transferência	76,989	38,494
4μs	Aritmética	172,266	43,066

Dados esses resultados, o modelo de consumo é perfeitamente descrito pelos valores presentes na Tabela A.2. A simplicidade do modelo pode ser atribuída a regularidade da arquitetura. O fato do i8051 ter uma arquitetura sem *pipeline* estabelece a execução da instrução como uma operação uniforme, na qual os recursos básicos de *hardware* são usados de forma idêntica. Essa regularidade pode ser percebida pela relação direta entre tempo de execução e consumo de energia, havendo contudo uma clara distinção em relação ao uso da Unidade Lógica Aritmética (ULA). O impacto do uso da ULA pode ser percebido pela diferença de consumo entre as instruções de transferência que não

usam a ULA e as demais, que usam. A diferença do consumo médio chega a 27,4% para instruções de 1μs, nota-se ainda que as instruções booleanas apresentam um consumo 5,8% maior que as aritméticas. Embora tal resultado possa parecer estranho, uma vez que não há motivo aparente para tamanha diferença, dado que ambas utilizam a ULA de forma similar, essa diferença pode ser explicada pela arquitetura de instruções do i8051. O i8051 permite a definição de variáveis booleanas em seu *assembly* através do mapeamento de bits das palavras de dados presentes entre os endereços 20H e 2FH da memória RAM interna. Esses endereços são conhecidos como posições bit-endereçáveis. Sendo assim, uma instrução booleana, diferentemente de uma aritmética, implementa um processo de acesso e mascaramento de bits, que impacta diretamente no consumo de energia. Cabe observar que este efeito é intrínseco à implementação da instrução na descrição do *hardware*, podendo variar de dispositivo a dispositivo.

A.6 Conclusões

O sistema de caracterização automática mostrou-se eficiente, guardando grande versatilidade para aplicação na caracterização de outros dispositivos. Embora seja plenamente funcional para o levantamento do modelo de consumo de instruções, o sistema apresenta deficiências metrológicas para medidas de tempos acima de alguns micro-segundos, devido à baixa profundidade de memória do osciloscópio utilizado. Essa limitação é contornável mediando a incorporação de um sistema de aquisição específico, baseado em placas de aquisição operando sobre um microcomputador PC-compatível. Tais sistemas de aquisição específicos podem ajustar a profundidade de memória e o modo de aquisição - tempo real ou tempo equivalente - em função do contexto de medida. Essas e outras melhorias figuram como implementos *a posteriori*.

Apêndice B

Simulação de Monte Carlo

B.1 Introdução

A expressão "Método de Monte Carlo" possui um sentido bem geral, identificando processos associados a técnicas de simulação estocásticas. Recebe a denominação de estocásticas, as simulações baseadas em avaliação de números aleatórios e probabilidades. Pode-se encontrar o método Monte Carlo em uma larga gama de aplicações, da avaliação de modelos em engenharia nuclear à engenharia de trafego. Evidentemente que as nuancias de implementação variam de domínio a domínio, mantém-se contudo a natureza estocástica do modelo de simulação. De uma forma geral, pode-se definir como simulação de Monte Carlo aquela na qual um modelo estocástico do sistema é simulado iterativamente até que um padrão de qualidade de estimativa possa ser atingido, ou seja, até que a estimativa da métrica de interesse apresente um erro menor que um patamar máximo estabelecido **a priori**.

O objetivo do do método de Monte de Carlo é identificar o ponto de parada da simulação, garantindo a exatidão pretendida sem desperdícios de ciclos de simulação. Dada uma hipótese de distribuição de valores da métrica alvo, é aplicado um critério de parada que avalia o número de simulações, o valor médio e o desvio padrão dos resultados obtidos na simulação. Por meio de tabelas de normalização, avalia-se a adequação dos resultados à meta de qualidade. Esse grau de adequação é estabelecido pela probabilidade da métrica ser determinada como um erro $E < \epsilon$, onde ϵ é um erro máximo admitido. A probabilidade expressa em termos percentuais caracteriza o *nível de confiança* da estimativa. Por exemplo, implementar uma simulação Monte Carlo com a parametrização $\epsilon = 0,03$ e $\alpha = 0,05$, estabece que a simulação terá tantos ciclos quanto forem necessários até que a estimativa da métrica alvo seja determinada com uma probabilidade de 95% (nível de confiança) de possui error menor que 3%. As

próximas seções apresentam com maior detalhamento esee processo.

B.2 Estimativas de Parâmetros

Na abordagem clássica de estimativa por simulação determinística um conjunto vetores de teste deve ser construído e o padrão determinístico de comportamento, associado a cada vetor, deve ser analisado. Dessa premissa, eclode o problema da dependência da qualidade da estimativa com o padrão de dados do vetor de teste. O problema da dependência de padrão resume-se na pergunta: quão descritivo é o conjunto de vetores de teste em relação aos reais cenários de execução do sistema? Burch *el al* [16] atenuaram a dependência de padrão do vetor de simulação aplicando padrões gerados aleatoriamente e avaliando a adequação do valor médio da potência frente a uma hipótese de distribuição de probabilidade de ocorrência de valores de potência no sistema alvo. O modelo do sistema[1] é simulado iterativamente, com vetores aleatórios a cada iteração. Ao fim de cada iteração, é avaliado se o valor médio da população de amostras (valores estimados da potência) enquadra-se na expectativa de erro máximo do processo.

Burch formulou o problema avaliando o consumo de energia do sistema a intervalos regulares, ou seja após um intervalo T_i uma potência p_i é estimada. Capturando dessa forma uma série de amostra de potência $p_0, p_1, p_2, p_3 ... p_n$, cujo valor médio é determinado por:

$$P = \frac{1}{N} \cdot \sum_{n=0}^{n=N} p_n. \tag{B.1}$$

Onde N é o número de iterações ou amostras. O problema clássico da estimativa do valor médio situa-se no fato da exatidão da estimativa depender de quão grande é o número de amostra N, em relação a população analisada. Aparece então uma questão: quão grande é a população de valores de potências possíveis em um sistema? A resposta mais sóbria a se dá é : *inumeravelmente grande*[2]. Dessa forma, se N tiver um baixo valor a exatidão da estimativa será comprometida, se for arbitrariamente alto, iterações desnecessárias serão executadas impactando na performance do sistema de estimativa. Uma vez que o tamanho da população é *inumeravelmente grande*, o valor adequado de N não pode ser determinado *a priori*, pode-se contudo realizar-se um trabalho com base em uma hipótese de distribuição de probabilidade dos valores de potência, assumindo p_i como uma variável aleatória, onde μ e σ^2 são o valor médio e a variância de p_i, respectivamente. O valor μ representa a métrica que se quer avaliar

[1] O Burch limitou-se a modelos de *hardware*.

[2] Uma vez que o sistema é digital o universo amostral será discreto, embora *inumeravelmente grande*

após N iterações de simulação do modelo. A questão que eclode é: quão exata é a estimativa de μ dada as N iterações? O teorema do limite central estabelece que variáveis aleatórias apresentam valores médios que assumem uma distribuição normal. Dessa forma, tem-se que para uma estimativa de potência P:

$$P \equiv \mu \text{ e } \sigma_P^2 = \frac{\sigma^2}{N}.$$

A medida que N aumenta a variância σ_P^2 diminui e a precisão da estimativa de $P \equiv \mu$ aumenta. Dado que P é uma estimativa de μ, questiona-se: para um erro máximo de estimativa ϵ, qual a probabilidade de P ser determinado com um erro $E \leq \epsilon$? Ou seja $0 \leq \frac{|P-\mu|}{\mu} \leq \epsilon$. Quanto maior for esta probabilidade maior será a **confiança** atribuída a estimativa. Definindo a variável aleatória $z_{\alpha/2}$ como a medida da dispersão de P em torno de μ, para uma avaliação com significância α, tem-se:

$$z_{\alpha/2} = \frac{P - \mu}{\sigma_P}, \tag{B.2}$$

de onde pode-se formular que o deslocamento aleatório de P em torno de μ será

$$P - \mu = z_{\alpha/2} \cdot \sigma_P. \tag{B.3}$$

A Figura B.1 ilustra a função de distribuição de probabilidade de P e o intervalo aleatório definido pela dispersão de P em torno de μ, em que a probabilidade de P ocorrer no intervalo $\left[\mu - z_{\alpha/2} \cdot \sigma_P, \mu + z_{\alpha/2} \cdot \sigma_P\right]$ é igual a área $(1 - \alpha)$. Observa-se portanto, que:

$$E \leq \epsilon \Longleftrightarrow \frac{|P - \mu|}{\mu} \leq \frac{z_{\alpha/2} \cdot \sigma_P}{\mu} \leq \epsilon.$$

Aplicando a igualdade $\sigma_P^2 = \frac{\sigma^2}{N}$ constrói-se a inequação:

$$\frac{z_{\alpha/2} \cdot \sigma}{\mu \cdot \sqrt{N}} \leq \epsilon. \tag{B.4}$$

Dado que α e ϵ são constantes definidoras da qualidade da estimativa, ou seja, erro máximo e significância estatística da estimativa, a inequação B.4 pode ser reformulada como:

$$N \geq \left(\frac{z_{\alpha/2} \cdot \sigma}{\epsilon \cdot \mu}\right)^2 \tag{B.5}$$

A inequação B.5 informa o número mínimo de amostras N que garantem uma probabilidade de $(1-\alpha)$ de P ser estimado com $E \leq \epsilon$. Dado que P possui distribuição normal $(N(\mu, \frac{\sigma^2}{N}))$, z possui distribuição $N(0, 1)$ [67]. Dessa forma, $z_{\alpha/2}$ pode ser determinado em função de α consultando a tabela de distribuição de z. Infelizmente, essa análise assume que há um conhecimento **a priori** de μ e σ, o que é completamente falso em um processo experimental.

B.3 Método Monte Carlo

O método de Monte Carlo compreende a aplicação da análise apresentada na seção anterior substituindo μ por P e σ por uma estimativa dada por:

$$S_N = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(p_i - P)^2}. \tag{B.6}$$

Uma vez que a $S_N \to \sigma$ apenas se $N \to \infty$, a dispersão de P em torno de μ para $N \neq \infty$ precisa ser substituída por uma outra variável aleatória, definida em função do número de amostras (N) e nível de significância (α), dada por:

$$t_{N-1,\alpha/2} = \frac{P - \mu}{S_N/\sqrt{N}}. \tag{B.7}$$

Se as amostras $p_0, p_1, p_2, p_3 ... p_n$ apresentam-se normalmente distribuídas a variável $t_{N-1,\alpha/2}$ assume a forma de uma distribuição conhecida como **distribuição t de Student** [20]. Identicamente ao caso da variável $z_{\alpha/2}$, a variável $t_{N-1,\alpha/2}$ pode ser capturada de uma tabela, em função do $N-1$ (grau de liberdade) e α (nível de significância).

Sendo assim, o número de amostras necessárias para se estimar $P \equiv \mu$, com uma probabilidade $(1-\alpha)$ de $E \leq \epsilon$, é descrito pela inequação:

$$N \geq \left(\frac{t_{N-1,\alpha/2} \cdot S_N}{\epsilon \cdot P}\right)^2. \tag{B.8}$$

Em outras palavras, em um processo contínuo de amostragem de p_n e estimativa de $P \equiv \mu$, a condição $P(E \leq \epsilon) = 1-\alpha$ é alcançado quando a inequação B.8 for verdadeira.

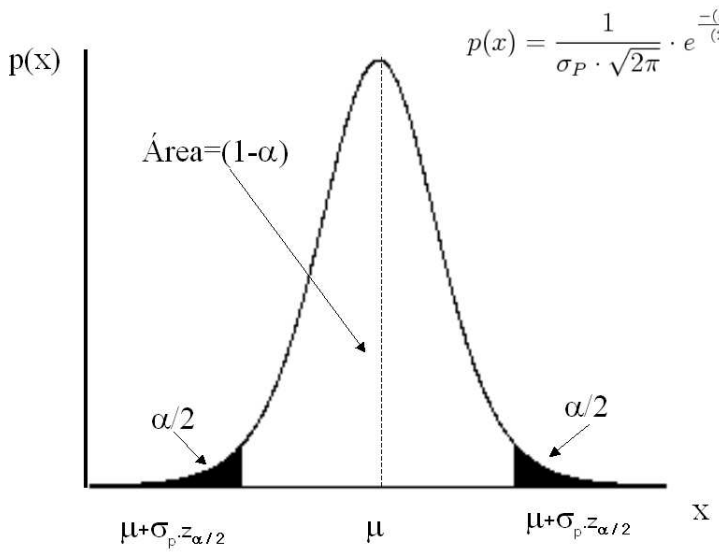

Figura B.1: Curva de distribuição normal da média de amostras de P.

B.3.1 Simulação Monte Carlo

A simulação de Monte Carlo compreende simular um modelo estocástico continuamente até que a inequação B.8 seja verdadeira. O processo de simulação pode ser resumido em três passos básicos:

1. Capturar n amostras incluíndo ao **bag** de amostras BA_i tal que $BA_i = BA_{i-1} \cup \{p_0, p_1, p_2, p_3 ... p_n\}$
2. Calcular P e S_N segundo as equações B.1 e B.6
3. Verificar a inequação B.8, caso a inequação seja falsa, voltar para passo 1, caso contrário, parar a simulação.

A simulação de Monte Carlo portanto garante a estimativa do parâmetro μ dentro de requisitos pré-definidos (nível de confiança $1 - \alpha$ e erro máximo ϵ) com o menor número de amostras possível.

B.4 Conclusão

A aplicação da simulação de Monte Carlo permite o emprego de técnicas estocásticas de simulação a partir de um padrão finito de estímulos, onde seria conceitualmente necessários infinitos estímulos. Dessa forma, sistemas complexos podem ser avaliados por meio de simulação de modelos determinísticos operando com vetores de testes aleatórios, como no trabalho de Burch *et al* [16], ou por modelos probabilísticos como o proposto nesta tese.

Apêndice C

Publicações

C.1 Referentes à pesquisa

C.1.1 *Analyzing Embedded Systems Software Performance and Energy Consumption by Probabilistic Modeling: An Approach Based on Coloured Petri Net.*

Meuse N. Oliveira Júnior, S. Neto, P. Maciel, R. Lima, A. Ribeiro, R. Barreto, E. Tavares, and F. Braga. ***Analyzing Embedded Systems Software Performance and Energy Consumption by Probabilistic Modeling: An Approach Based on Coloured Petri Net.*** **In 27th International Conference on Application and Theory of Petri Nets and Other Models of Councurrency (Petri Nets 2006)**. Springer LNCS 4024. Turku, Finland, June 26-30, 2006, pp. 261-281 (ISBN: 3-540-34699-6).

C.1.2 *A Retargetable Environment for Power-Aware Code Evaluation: An Approach Based on Coloured Petri Net*

Meuse N. Oliveira Júnior, P. Maciel, A. Ribeiro, C. , A. Arcoverde Jr, R. Lima, L. Amorim, R. Barreto, and E. Tavares. ***A Retargetable Environment for Power-Aware Code Exploration: An Approach Based on Coloured Petri Nets.*** **In International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'05)**, Leuven, Belgium. September 20-23, 2005.Lecture Notes in Computer Science 3728 Springer 2005, ISBN 3-540-29013-3.

C.1.3 *Towards A Software Power Cost Analysis Framework Using Colored Petri Net*

Meuse N. Oliveira Júnior, P. Maciel, R. Barreto, and F. Carvalho. ***Towards A Software Power Cost Analysis Framework Using Colored Petri Net.*** **In 14th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'2004)**. Enrico Macii, Vassilis Paliouras, Odysseas Koufopavlou (Editors). Lecture Notes in Computer Science (LNCS 3254). Springer-Verlag. Isle of Santorini,Greece. pp. 362 - 371, September 15-17, 2004. ISBN 3-540-23095-5.

C.1.4 *A Software Power Cost Analysis Based on Colored Petri Net*

Meuse N. Oliveira Júnior, P. Maciel, R. Barreto, and F. Carvalho. ***A Software Power Cost Analysis based on Colored Petri Net.*** **In Workshop on Token-Based Computing (TOBACO), in conjuction with 25th Int. Conf. on Applications and Theory of Petri Net (ICATPN'04)**. pp. 33-42, Bologna, Italy, June 22, 2004.

C.2 Contribuições em outras pesquisas

C.2.1 *An Approach for Analysis and Verification of Embedded Systems' Properties Based on CPN and LSC.*

L. Amorim, P. Maciel, **Meuse N. Oliveira Júnior**, R. Barreto, and E. Tavares. ***An Approach for Analysis and Verification of Embedded Systems' Properties Based on CPN and LSC.*** **In ACM SIGSOFT Software Enginnering Notes**, Volume 31, Issue 3, Maio de 2006.

C.2.2 *An Integrated Environment for Embedded Hard Real-Time Systems Scheduling with Timing and Energy Constraints.*

E. Tavares, R. Barreto, P. Maciel, **Meuse N. Oliveira Júnior**, R. Lima, A. Arcoverde Jr, G. Alves Jr, A. Bessa. ***An Integrated Environment for Embedded Hard Real-Time Systems Scheduling with Timing and Energy Constraints.*** **In International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'05)**, Leuven, Belgium. September 20-23, 2005.Lecture Notes in Computer Science 3728 Springer 2005, ISBN 3-540-29013-3.

C.2.3 *Embedded Hard Real-Time Software Synthesis Considering Dispatcher Overheads.*

R. Barreto, E. Tavares, P. Maciel, M. Neves, **Meuse N. Oliveira Júnior**, L. Amorim, A. Bessa, and R. Lima. ***Embedded Hard Real-Time Software Synthesis Considering Dispatcher Overheads.*** **In International Embedded System Symposium (IESS'05)**. 15-17 August 2005. Manaus, Brazil. Published as book. "From Specification to Embedded Systems Application". Springer Publisher. ISBN 0-387-27557-6. A Rettberg, M. Zanella, and F. Rammig, eds

C.2.4 *A Methodology for Software Synthesis of Embedded Real-Time Systems Based on TPN and LSC*

L. Amorim, R. Barreto, P. Maciel, E. Tavares, **Meuse N. Oliveira Júnior**, A. Bessa, and R. Lima. ***A Methodology for Software Synthesis of Embedded Real-Time Systems Based on TPN and LSC.*** **In 2nd International Conference on**

Embedded Software and Systems (ICESS'05), pp 50-62, Xi'an, China. December 16-18, 2005. Lecture Notes in Computer Science 3820 Springer 2005, ISBN 3-540-30881-4.

C.2.5 *Pre-Runtime Scheduling considering Timing and Energy Constraints in Embedded Systems with Multiple Processors.*

E. Tavares, P. Maciel, **Meuse N. Oliveira Junior**, B. Souza, R. Barreto, R. Freitas, and M. Custódio. ***Pre-Runtime Scheduling considering Timing and Energy Constraints in Embedded Systems with Multiple Processors***. **In 5th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'2006)**. Braga, Portugal. October 11-13, 2006.

C.2.6 *EZPetri: A Petri net interchange framework for Eclipse based on PNML*

A. Arcoverde Jr, G. Alves Jr, R. Lima, P. Maciel, **Meuse N. Oliveira Júnior**, and R. Barreto. ***EZPetri: A Petri net interchange framework for Eclipse based on PNML.*** **In First International Symposium on Leveraging Applications of Formal Method (ISoLA'04)**. 30th October - 2st November 2004, Paphos, Cyprus

Tese de Doutorado apresentada por **Meuse Nogueira de Oliveira Junior** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Estimativa do Consumo de Energia Devido ao Software: Uma Abordagem Baseada em Redes de Petri Coloridas"**, orientada pelo **Prof. Paulo Romero Martins Maciel** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Paulo Roberto Freire Cunha
Centro de Informática / UFPE

Prof. Nelson Souto Rosa
Centro de Informática / UFPE

Prof. Ricardo Massa Ferreira Lima
Escola Politécnica de Pernambuco / UPE

Prof. Norian Marranghello
Departamento de Ciência da Computação e Estatística / UNESP

Prof. Tomaz de Barros Carvalho
Departamento de Eletrônica e Sistemas / UFPE

Visto e permitida a impressão.
Recife, 27 de outubro de 2006

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.