Pós-Graduação em Ciência da Computação

Bruno Costa e Silva Nogueira

EXPLORAÇÃO MULTIOBJETIVO DO ESPAÇO DE PROJETO DE SISTEMAS EMBARCADOS DE TEMPO-REAL NÃO CRÍTICOS

Tese de Doutorado

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

RECIFE
2015

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciência da Computação

Bruno Costa e Silva Nogueira

EXPLORAÇÃO MULTIOBJETIVO DO ESPAÇO DE PROJETO DE SISTEMAS EMBARCADOS DE TEMPO-REAL NÃO CRÍTICOS

Trabalho apresentado ao Programa de 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: *Paulo Romero Martins Maciel*
Co-Orientador: *Ricardo Martins Abreu Silva*

RECIFE
2015

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

778e Nogueira, Bruno Costa e Silva
Exploração multiobjetivo do espaço de projeto de sistemas embarcados de tempo-real não críticos / Bruno Costa e Silva Nogueira – Recife: O Autor, 2015.
148 f.: il., fig., tab.

Orientador: Paulo Romero Martins Maciel.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2015.
Inclui referências e apêndice.

1. Engenharia da computação. 2. Arquitetura de computador. 3. Otimização. 4. Avaliação de desempenho. I. Maciel, Paulo Romero Martins. (orientador). II. Título.

621.39 CDD (23. ed.) UFPE- MEI 2015-34

Tese de Doutorado apresentada por **Bruno Costa e Silva Nogueira** à Pós Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Exploração Multiobjetivo do Espaço de Projeto de Sistemas Embarcados de Tempo-Real Não Críticos"** 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. Ricardo Massa Ferreira Lima
Centro de Informática / UFPE

_____

Prof. Djamel Fawzi Hadj Sadok
Centro de Informática / UFPE

_____

Prof. Marius Strum
Escola Politécnica/USP

_____

Prof. Henrique Pacca Loureiro Luna
Instituto de Computação/UFAL

Visto e permitida a impressão.
Recife, 12 de fevereiro de 2015.

_____

Profa. Edna Natividade da Silva Barros
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Agradecimentos

Este trabalho não seria possível sem a contribuição de muitas pessoas, as quais, agora, eu tenho a oportunidade de agradecer.

Primeiramente, agradeço ao prof. Paulo Maciel, pela paciência, orientação e apoio oferecidos desde a época de iniciação científica até o fim desta jornada. Considero-me uma pessoa de sorte por tido Paulo como orientador e sou extremamente grato por seus conselhos sobre pesquisa, engenharia e escrita. Agradeço também ao prof. Ricardo Martins pela co-orientação do trabalho e importantes contribuições.

Agradeço aos amigos do grupo de pesquisa MoDCS, com quem convivi nos últimos nove anos (desde o período de graduação), pelo companheirismo, períodos de descontração e contribuições ao trabalho. Meu agradecimento em especial a: Eduardo Tavares, Erica Souza, Ermeson Carneiro, Gustavo Callou, Jean Araujo e Julian Menezes.

Agradeço aos amigos professores da UFRPE/UAG, em especial, Diogo Lages, Marcius Petrucio e Rian Gabriel.

Agradeço a meu pai, Fernando Nogueira, e minha mãe, Maria da Anunciação, por todo apoio e incentivo. Sempre fizeram o melhor que podiam para contribuir com a minha formação. Agradeço também aos amigos e restante da família Costa e Silva.

Agradeço aos membros da banca, profs.: Paulo Cunha, Ricardo Massa, Djamel Sadok, Marius Strum e Henrique Pacca pelos conselhos valiosos e que ajudaram bastante a melhorar este trabalho.

Ao CIn-UFPE e à CAPES, por todo o suporte recebido.

Resumo

Nos últimos anos, a indústria tem adotado sistemas embarcados com múltiplos e heterogêneos processadores como uma resposta viável à demanda por mais desempenho e baixa potência consumida. No entanto, programar, depurar, simular e otimizar arquiteturas heterogêneas são atividades complexas, e isso tem forçado as empresas a lidar com diversos novos desafios para aumentar a produtividade de seus projetistas. Um dos desafios proeminentes é disponibilizar métodos para que os projetistas possam eficientemente explorar o espaço de projeto. A exploração do espaço de projeto refere-se ao processo de explorar e avaliar diferentes decisões (opções) de projeto durante o desenvolvimento do sistema. Diversas abordagens têm sido propostas para resolver o problema de exploração, que é composto por duas questões complementares: (i) como representar e avaliar uma alternativa de projeto (modelos), e (ii) como percorrer o espaço de projeto (algoritmos), dado que a exploração exaustiva é usualmente inviável. Apesar da disponibilidade de métodos de exploração, as abordagens atuais possuem diversas restrições, principalmente, em relação ao tempo de avaliação e a exatidão dos modelos adotados para representar sistemas de tempo-real não críticos.

Este trabalho apresenta um novo método de exploração do espaço de projeto para sistemas embarcados de tempo-real não críticos. O principal objetivo deste trabalho é prover meios para que o projetista possa escolher uma arquitetura composta por processadores heterogêneos e programáveis para uma dada aplicação de tempo-real não crítica, considerando diversas restrições conflitantes de projeto, como: probabilidades de violação de *deadlines* e potência consumida. O método adota uma abordagem centrada em simulação estocástica para evitar os problemas relacionados ao tempo de avaliação e exatidão dos métodos existentes. Dentre as contribuições do método proposto, destacam-se: (i) novos modelos de especificação para definir as restrições e os atributos da aplicação/plataforma de *hardware*, (ii) método automático de mapeamento dos modelos de especificação em modelos formais DEVS (*Discrete Event System Specification*) para simulação estocástica, (iii) novos algoritmos de exploração multiobjetivo, baseados em algoritmos genéticos, e (iv) uma biblioteca para dar suporte ao desenvolvimento de aplicações que executam em arquiteturas compostas por processadores heterogêneos e programáveis. Diversos experimentos foram conduzidos para demonstrar a viabilidade do método proposto. Os resultados mostram a boa exatidão dos modelos de desempenho desenvolvidos (erro máximo de 5%, em comparação a medições em um sistema real), e a eficiência do método proposto em encontrar soluções de boa qualidade para especificações que os métodos existentes têm dificuldade em explorar.

Palavras-chave: Sistemas Embarcados. Exploração do Espaço de Projeto. Sistemas de Tempo-Real. Simulação Estocástica. Algoritmos Genéticos.

Abstract

In the last years, industry has adopted embedded systems with multiple and heterogeneous processors as a viable solution for the ever-increasing demand for higher performance and lower power consumption. However, programming, debugging, simulating, and optimizing heterogeneous architectures are complex tasks, which has forced companies to deal with several new challenges in order to increase their designers' productivity. One prominent challenge is to provide efficient methods for design space exploration. Design space exploration refers to the activity of exploring and evaluating different design decisions (options) during system development. Several approaches have been proposed to tackle the exploration problem, which is composed of two complimentary issues: (i) how to represent and evaluate a design alternative (models), and (ii) how to traverse the design space (algorithms), given that exhaustive exploration is usually infeasible. Although several methods have been proposed for design space exploration, they have many drawbacks, mainly related to the evaluation time and accuracy of the models adopted to represent soft real-time embedded systems.

This work presents a new method for design space exploration of soft real-time embedded systems. The main objective of this work is to provide to the designer means for choosing an optimized architecture for a given application, considering several conflicting design objectives, such as: deadline miss violation probabilities and power consumption. The proposed method adopts an approach centered on stochastic simulation to prevent the problems related to evaluation time and accuracy of current methods. Among the contributions of this work are: (i) new models for specifying application/architecture restrictions and attributes. (ii) automatic method for mapping the specification models into formal DEVS (*Discrete Event System Specification*) models for stochastic simulation, (iii) new algorithms, based on genetic algorithms theory, for multiobjective exploration, and (iv) a new library for designing applications that execute on multiprocessor heterogeneous architectures. Several experiments have been conducted to demonstrate the viability of the proposed method. Results show the accuracy of the proposed performance models (maximum error of 5%, in comparison with measurements on a real system), and the efficiency of the proposed method in finding good quality solutions for specifications that current methods cannot satisfactorily explore.

Keywords: Embedded Systems. Design Space Exploration. Real-Time Systems. Genetic Algorithms. Stochastic Simulation.

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

AHB - *Advanced High-performance Bus*.

AMP - *Asymmetrical Multi-Processing*.

API - *Application Program Interface*.

ATG - *Architecture Template Graph*.

DAG - *Directed Acyclic Graph*.

DEVS - *Discrete Event System Specification*.

DMA - *Direct memory access*.

EC - Elemento de Comunicação.

EP - Elemento de Processamento.

FIFO - *First In, First Out*.

HDL - *Hardware Description Language*

HSDG - *Homogenous Synchronous Dataflow Graph*.

IP - *Intelectual Property.*

NoC - *Network on Chip*.

RTL - *Register Transfer Level*.

SO - Sistema operacional.

SMP - *Symmetrical Multi-Processing*.

WCET - *Worst-Case Execution Time*.

Sumário

1 Introdução

Um sistema embarcado pode ser definido como qualquer dispositivo que inclui um ou mais processadores programáveis, mas que, no entanto, não é considerado pelo usuário como um computador (WOLF, 2012; LAVAGNO; PASSERONE, 2006). Dessa forma, um *desktop*, *laptop* ou servidor não são sistemas embarcados. Em muitos casos, os sistemas embarcados são auto contidos (ex.: celular). Em outros casos, eles são integrados a produtos maiores com o intuito de prover mais funcionalidades como, por exemplo, o controle de estabilidade de um carro. Da agricultura de precisão aos aviões, das casas inteligentes à medicina, o mundo tem testemunhado nas últimas décadas um grande crescimento nas áreas de aplicações destes sistemas. Atualmente, não é nenhum exagero afirmar que quase todo dispositivo eletrônico possui ou em breve possuirá um sistema computacional embarcado em si (VAHID; GIVARGIS, 2001).

Embora os sistemas embarcados sejam usados em aplicações bastante distintas, frequentemente eles possuem várias características em comum. Muitos sistemas embarcados, por exemplo, podem ser classificados como sistemas de tempo-real (VAHID; GIVARGIS, 2001; MARWEDEL, 2011; WOLF, 2012). Em um sistema de tempo-real, o correto funcionamento das operações do sistema depende não apenas dos resultados lógicos das computações, mas também do *tempo* em que esses resultados são produzidos (KOPETZ, 2011). O tempo máximo para que o sistema produza um resultado é chamado de *deadline* (LIU, 2000; KOPETZ, 2011). Caso a violação do *deadline* acarrete em um defeito grave, então o *deadline* é dito crítico, caso contrário, ele é chamado de não crítico (LIU, 2000). Sistemas de tempo-real em que pelo menos um *deadline* é crítico são chamados de sistemas de tempo-real críticos (ex.: sistemas embarcados usados em automóveis, aviões ou em aplicações médicas) (KOPETZ, 2011). Se nenhum *deadline* crítico existe, então eles são chamados de sistemas de tempo-real não críticos (KOPETZ, 2011). Assim, para sistemas não críticos, violações de *deadlines* são toleráveis, embora não sejam desejáveis, pois podem resultar na degradação do serviço oferecido. Sistemas embarcados usados em aplicações multimídia são exemplos proeminentes de sistemas de tempo-real não críticos. Por exemplo, ao exibir um *stream* de vídeo a 25 quadros por segundo, a perda ocasional de um quadro geralmente não é percebida pelo usuário (ABENI; MANICA; PALOPOLI, 2012; MANOLACHE; ELES; PENG, 2008). Processamento de sinais e controle em tempo-real são outras áreas em que esses sistemas podem ser encontrados (ABENI; MANICA; PALOPOLI, 2012; GARDNER; LIU, 1999).

Além das restrições temporais, o projeto de um sistema embarcado comumente deve levar em consideração diversas outras restrições não funcionais, tais como, custo, potência/consumo de energia, confiabilidade, tamanho e peso (VAHID; GIVARGIS, 2001; LAVAGNO; PASSERONE, 2006; MARWEDEL, 2011; WOLF, 2012). Por exemplo, consumo de energia afeta diretamente a duração das baterias, que são importantes nas situações em que o sistema embarcado está presente em um dispositivo portátil. Não obstante, frequentemente as restrições de projeto estão em conflito. Para tentar diminuir a probabilidade do sistema violar um *deadline*, uma possibilidade seria utilizar um *hardware* mais rápido, no entanto, isso poderia afetar outras

restrições de projeto já que frequentemente um *hardware* mais rápido acarreta em um sistema mais caro ou com maior consumo de energia.

Existem inúmeras metodologias de desenvolvimento para sistemas embarcados. Elas diferem de empresa para empresa e até mesmo em equipes diferentes de uma mesma empresa (GAJSKI et al., 2009). Com o aumento da complexidade dos sistemas digitais, essas metodologias de projeto, assim como as ferramentas de automação (também conhecidas como CAD - *Computer-Aided Design tools*) evoluíram para considerar níveis de abstração cada vez mais altos (CHU, 2006; GAJSKI et al., 2009). Segundo (GAJSKI et al., 2009), normalmente quatro níveis de abstração são considerados: (i) circuito, (ii) lógico, (iii) microarquitetura (ou RTL - *Register Transfer Level*) e (iv) sistema. A diferenciação entre esses níveis de abstração é dada principalmente pelos blocos básicos utilizados em cada nível (CHU, 2006; GAJSKI et al., 2009). O nível de abstração mais baixo é o nível de circuito, em que os blocos básicos são, por exemplo, transistores, resistores e capacitores. O próximo nível é o nível lógico. Comumente blocos básicos nesse nível incluem portas lógicas, *latches* e *flip-flops*. No nível RTL (microarquitetura), os blocos básicos são: somadores, registradores, multiplexadores, etc. O nível de sistema é o nível mais alto, e seus blocos básicos, frequentemente conhecidos como IP (*Intelectual Property*), são, por exemplo, processadores, memórias e interfaces de barramentos.

Uma atividade de fundamental importância em qualquer metodologia de desenvolvimento é a exploração do espaço de projeto (SCHMITZ; AL-HASHIMI; ELES, 2004; DICK, 2002; WOLF, 2014; JIA et al., 2014). A atividade de exploração do espaço de projeto refere-se ao processo de explorar e avaliar diferentes decisões (opções) de projeto durante o desenvolvimento do sistema. Dessa forma, em geral, o problema de exploração do espaço de projeto é composto por duas questões complementares (JIA et al., 2014; GRIES, 2004; KEMPF; ASCHEID; LEUPERS, 2011): (i) como representar e avaliar uma alternativa de projeto (modelos), e (ii) como percorrer o espaço de projeto (algoritmos), dado que a exploração exaustiva é usualmente inviável. Frequentemente os projetistas usam o conhecimento adquirido em projetos similares para restringir o espaço de projeto e selecionar a melhor a solução a partir de apenas algumas alternativas de projeto. Porém, devido ao aumento da complexidade dos sistemas embarcados, esta abordagem "intuitiva" tem se tornado extremamente propensa a erros e lenta (GAJSKI et al., 2009; GRIES, 2004). Os projetistas, portanto, precisam de métodos e ferramentas automáticas que os auxiliem a eficientemente explorar e selecionar as alternativas ótimas ou quase ótimas do espaço de soluções. Neste contexto, esta tese propõe um novo método de exploração do espaço de projeto. O principal objetivo deste trabalho é prover meios para que o projetista possa escolher uma arquitetura composta por processadores heterogêneos e programáveis para uma dada aplicação de tempo-real não crítica, considerando as seguintes restrições conflitantes de projeto: probabilidades de violação de *deadlines*, potência consumida e custo monetário.

1.1 Projeto baseado em plataformas

Este trabalho segue a metodologia de desenvolvimento baseada em plataformas (*platform-based design*) (FERRARI; SANGIOVANNI-VINCENTELLI, 1999; SANGIOVANNI-VINCENTELLI; MARTIN, 2001; SANGIOVANNI-VINCENTELLI et al., 2004; DENSMORE; PASSERONE; SANGIOVANNI-VINCENTELLI, 2006). Essa metodologia se fundamenta nos seguintes princípios:

- **Reuso de blocos IPs:** Este princípio tem como objetivos reduzir a complexidade de desenvolvimento e os custos de fabricação. A redução dos custos de fabricação se dá através do compartilhamento de um "*hardware* comum" (chamado de plataforma)

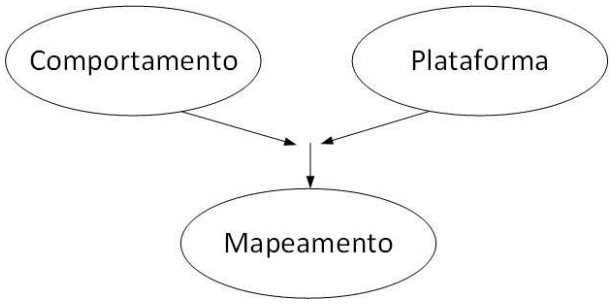


Figura 1.1: Estrutura básica da metodologia de desenvolvimento baseada em plataformas (DENSMORE; PASSERONE; SANGIOVANNI-VINCENTELLI, 2006).

entre um conjunto de produtos. A utilização de uma plataforma comum permite que o *hardware* possa ser produzido em larga escala, o que faz com que o custo total de fabricação seja inferior ao custo de fabricação de um *hardware* especificamente desenvolvido para uma dada aplicação. Neste sentido, a flexibilidade (capacidade de customização) é um importante critério para definir a qualidade de uma plataforma.

- **Separação de interesses:** O objetivo deste princípio é permitir uma exploração mais eficaz das alternativas de projeto. Uma separação fundamental no processo de desenvolvimento é a separação entre funcionalidade (o que o sistema deve fazer) e arquitetura (como ele faz).

A Figura 1.1 apresenta a estrutura básica da metodologia baseada em plataformas (DENSMORE; PASSERONE; SANGIOVANNI-VINCENTELLI, 2006). A parte esquerda da figura apresenta as funcionalidades que o projetista deseja implementar (ex.: um decodificador de vídeo MPEG-2), a parte direita apresenta os elementos que podem ser usados para implementar as funcionalidades, e a parte de baixo identifica os elementos que o projetista usará para implementar as funcionalidades (o mapeamento). Nesse contexto, o lado direito da figura é a plataforma. A plataforma é composta por (FERRARI; SANGIOVANNI-VINCENTELLI, 1999; DENSMORE; PASSERONE; SANGIOVANNI-VINCENTELLI, 2006): (i) uma biblioteca de elementos (ex.: processadores, memórias), e as regras de composição que expressam como e que elementos podem ser combinados, (ii) um método para obter informações sobre as características de cada elemento (ex.: potência consumida, tempo para executar uma dada computação), e (iii) uma API (*Application Program Interface*), formada por sistema operacional e *device drivers*, que objetiva abstrair detalhes da interface entre *software* e *hardware*, assim como facilitar o reuso.

Cada combinação válida dos elementos da plataforma é chamada de instância da plataforma. Dessa forma, no processo de mapeamento, as funcionalidades da aplicação são atribuídas aos elementos da plataforma (i.e., uma instância da plataforma). A saída do processo de mapeamento é uma arquitetura mapeada, que pode ser refinada até a implementação final. O projetista deve otimizar o processo de mapeamento considerando as restrições de projeto envolvidas (ex.: desempenho, potência consumida). São estas restrições que definem a qualidade do mapeamento. O leitor deve observar que a estrutura apresentada na Figura 1.1 pode ser aplicada em qualquer nível de abstração (ex.: nível lógico, nível de microarquitetura). Neste trabalho, nós focamos no nível de abstração de sistema.

1.2 Motivação

Nos últimos anos, a indústria tem adotado sistemas embarcados com múltiplos e heterogêneos processadores como uma resposta viável à demanda por mais desempenho e baixa potência consumida (WOLF; JERRAYA; MARTIN, 2008; WOLF, 2014; MOYER, 2013). A potência consumida (estática e dinâmica) dos processadores baseados na tecnologia CMOS (dominante nos dias de hoje (WOLF, 2012)) está diretamente relacionada à frequência de operação do *clock* do processador (PATTERSON; HENNESSY, 2013). A vantagem dos sistemas multiprocessados em relação aos monoprocessados é que o desempenho pode ser melhorado aumentando o número processadores ao invés de aumentar a frequência do *clock*, o que ajuda a evitar o alto consumo de potência associado a altas frequências de *clock* (BLAKE; DRESLINSKI; MUDGE, 2009). No entanto, programar, depurar, simular e otimizar arquiteturas heterogêneas são atividades complexas (WOLF, 2014; MOYER, 2013), e isso tem forçado as empresas a lidar com diversos novos desafios para aumentar a produtividade de seus projetistas. Um dos desafios proeminentes é disponibilizar métodos eficientes para exploração do espaço de projeto. Devido ao grande número de alternativas de projeto, normalmente, a exploração exaustiva de todas as possibilidades é inviável (GRIES, 2004; JIA et al., 2014). Por exemplo, considere o problema de mapear uma aplicação com t tarefas (ou processos) em uma arquitetura com p processadores de diferentes tipos. Se as tarefas podem ser executadas em qualquer processador, então o número de alternativas de projeto é dado por p^t. Consequentemente, a análise de todas as possibilidades se torna rapidamente inviável com o aumento do número de processadores ou de tarefas.

Esta tese se concentra no problema de achar uma arquitetura ótima (ou próxima da ótima) para uma dada aplicação de tempo-real não crítica, considerando múltiplos objetivos de projeto (potência consumida, desempenho e custo monetário). Este problema tem atraído muita pesquisa nos últimos anos (GRIES, 2004; JIA et al., 2014; GERSTLAUER et al., 2009; KEMPF; ASCHEID; LEUPERS, 2011; HERRERA; SANDER, 2015; NOGUEIRA et al., 2010) e, como consequência, diversos modelos têm sido propostos para representar aplicações e arquiteturas. De maneira geral, tais modelos podem ser classificados como: (i) modelos abstratos, e (ii) modelos executáveis (GRIES, 2004). Nos modelos abstratos de arquitetura, desempenho é apenas simbolicamente representado, por exemplo, associando o tempo de execução em ciclos de *clock* de uma operação sem, de fato, executar uma especificação da arquitetura (GRIES, 2004). Em modelos executáveis de arquitetura, as estimativas são feitas com base na execução (simulação) de uma especificação da arquitetura. Embora geralmente sejam menos exatos que modelos executáveis, a maioria dos métodos de exploração automática do espaço de projeto adota modelos abstratos de arquitetura (GRIES, 2004; GERSTLAUER et al., 2009; KEINERT et al., 2009; JIA et al., 2014; HERRERA; SANDER, 2015; PIMENTEL; ERBAS; POLSTRA, 2006), pois eles permitem explorar mais rapidamente as alternativas de projeto. De modo similar à classificação dos modelos de arquitetura, em modelos abstratos de aplicação, a carga de trabalho do sistema é definida sem que a especificação funcional da aplicação seja executada (simulada). Por outro lado, em modelos executáveis da aplicação, as estimativas são feitas com base na execução da aplicação.

Tradicionalmente, métodos de exploração do espaço de projeto que adotam modelos abstratos tanto para arquitetura quanto para a aplicação assumem que as tarefas do sistema têm tempos constantes (BLICKLE, 1997; DICK, 2002; ERBAS; CERAV-ERBAS; PIMENTEL, 2006; RUGGIERO et al., 2006; TAVARES et al., 2010). Apesar de facilitar a exploração do espaço de projeto, esse modelo de tempos constantes é pouco realista na maioria dos casos, uma vez que variações nos tempos de execução dos sistemas embarcados podem ser causadas por diversos fatores, tais como, dados de entrada e características arquiteturais (*pipeline*, *caches*).

Frequentemente os trabalhos que assumem tempos constantes consideram que eles são sempre iguais aos piores tempos de execução das tarefas (WCET - *Worst-Case Execution Time*) (TAVARES et al., 2010). Embora essa abordagem seja aceitável para sistemas embarcados de tempo-real críticos, pois garante-se que no pior caso o sistema não violará *deadlines*, tal abordagem pode ser demasiadamente pessimista para sistemas de tempo-real não críticos e pode resultar em arquiteturas desnecessariamente caras e ineficientes. Ao invés de considerar tempos constantes, distribuições de probabilidade são preferíveis para modelar o tempo de execução de sistemas embarcados de tempo-real não críticos, uma vez que através delas é possível computar as probabilidades de violação de *deadlines* (MANOLACHE; ELES; PENG, 2008). Diversos métodos têm sido propostos para avaliar analiticamente (ou numericamente) sistemas embarcados em que os tempos de execução são modelados por distribuições de probabilidades (REFAAT; HLADIK, 2010; KIM; LEE, 2009; ABENI; MANICA; PALOPOLI, 2012; MANOLACHE; ELES; PENG, 2002, 2004; MUPPALA; WOOLET; TRIVEDI, 1991; ZAMORA; HU; MARCULESCU, 2007; MANOLACHE; ELES; PENG, 2008). No entanto, a maioria desses trabalhos considera apenas especificações simplificadas do sistema. Por exemplo, grande parte deles considera arquiteturas com apenas um processador, ou sistemas nos quais as tarefas são descritas por distribuições de probabilidade específicas, como exponencial ou *Coxian* (MANOLACHE; ELES; PENG, 2002).

Métodos baseados em modelos executáveis da aplicação (KEMPF; ASCHEID; LEUPERS, 2011; KANGAS et al., 2006; PIMENTEL; ERBAS; POLSTRA, 2006; JIA et al., 2014; HERRERA; SANDER, 2015; DÖMER et al., 2008) são uma alternativa natural para capturar a variabilidade no tempo de execução dos sistemas embarcados. Apesar de normalmente serem mais exatos que modelos abstratos, a desvantagem de modelos executáveis de aplicação é que o tempo de avaliação estará diretamente relacionado à complexidade das funcionalidades da aplicação. Em modelos abstratos, essa dependência é mais fraca, uma vez que os aspectos funcionais da aplicação normalmente são vistos como uma caixa-preta. Considerando que, por exemplo, poucos minutos de um vídeo MPEG-2 podem facilmente demandar a simulação *Gbytes* de dados de entrada (MARCULESCU; PEDRAM; HENKEL, 2004), fica evidente que a exploração do espaço de projeto usando modelos executáveis em alguns casos pode se tornar inviável. Dessa forma, para acelerar o processo de exploração, grande parte dos métodos que se baseiam em modelos executáveis adota uma estratégia de duas fases (JIA et al., 2014). Na primeira fase, um conjunto de soluções é selecionado usando exploração baseada em modelos abstratos (normalmente assume-se que as tarefas têm tempos constantes) e, na segunda fase, as soluções selecionadas pela fase anterior são avaliadas com maior exatidão usando modelos executáveis. Porém, o problema dessa abordagem é que, devido às limitações de representação dos modelos abstratos utilizados, soluções de boa qualidade podem ser erroneamente ignoradas na primeira fase.

Pela análise acima, é possível notar que os modelos abstratos e executáveis atuais não são totalmente adequados para explorar o espaço de projeto de sistemas embarcados de tempo-real não críticos, uma vez que os primeiros possuem diversas limitações em relação à sua capacidade de representação, e os últimos podem demandar longos tempos de simulação.

1.3 Objetivos e contribuições

Este trabalho propõe um novo método de exploração do espaço de projeto de sistemas embarcados de tempo-real não críticos. O trabalho proposto utiliza uma estratégia centrada em simulação estocástica (BOLCH et al., 2006) para evitar os problemas relacionados ao tempo de avaliação e representatividade dos métodos existentes. Os itens a seguir relacionam as principais contribuições do trabalho:

- **Modelos de especificação:** O trabalho propõe modelos de especificação abstratos para que o projetista possa definir as restrições e os atributos da aplicação/plataforma de *hardware*. Diferentemente dos métodos que assumem que as tarefas possuem tempos constantes, os modelos de especificação propostos capturam a variabilidade no tempo de execução dos sistemas embarcados por meio de distribuições de probabilidade.

- **Modelos de desempenho:** Modelos de simulação baseados no formalismo DEVS (*Discrete Event System Specification*) (ZEIGLER; PRAEHOFER; KIM, 2000) são propostos para estimar o desempenho de sistemas embarcados não críticos. Estes modelos são adotados para representar os processadores da arquitetura, o sistema operacional que executa em cada processador, e a aplicação. O formalismo DEVS é usado para garantir que o comportamento concorrente inerente aos sistemas embarcados possa ser representado de forma precisa e sem ambiguidades. Ao contrário da grande maioria dos modelos de simulação adotados pelos métodos atuais (KEINERT et al., 2009; DÖMER et al., 2008; JIA et al., 2014), os modelos concebidos são livres de detalhes funcionais da arquitetura e aplicação. Além dos modelos de simulação, este trabalho também propõe modelos analíticos para representar a potência consumida da plataforma de *hardware*.

- **Mapeamento:** Por meio de uma estratégia de combinação de blocos básicos, este trabalho define um método que automaticamente mapeia a especificação de alto nível em um modelo formal de simulação DEVS. Dessa forma, o projetista não precisa tratar diretamente com os modelos de simulação propostos.

- **Algoritmos de exploração:** Dois algoritmos multiobjetivo são concebidos para exploração do espaço de soluções: MODSES e C-MODSES. Dada a especificação de alto nível, estes algoritmos tentam identificar soluções que otimizem os seguintes objetivos de projeto: probabilidades de violação de *deadlines*, potência consumida e custo monetário. MODSES e C-MODSES utilizam os modelos de desempenho propostos para avaliar a qualidade de uma dada solução. Dessa forma, além de liberar o projetista do trabalho de ter que explorar manualmente o espaço de projeto, os algoritmos propostos oferecem os seguintes benefícios em relação aos métodos existentes: (i) eles não são baseados em tempos constantes e, portanto, permitem que o projetista possa estabelecer a melhor relação entre os requisitos temporais e os outros objetivos de projeto, (ii) eles exploram, por meio de simulação, soluções que os métodos analíticos atuais têm dificuldade em avaliar, e (iii) eles são mais rápidos que os métodos que adotam modelos executáveis de aplicação (KEINERT et al., 2009; DÖMER et al., 2008; JIA et al., 2014), uma vez que as funcionalidades da aplicação são vistas como uma caixa-preta para os modelos de simulação propostos.

- **Métodos de caracterização:** Algumas informações sobre o tempo de execução das tarefas da aplicação e a potência consumida da arquitetura precisam ser associadas aos modelos de especificação. Métodos baseados em medição são propostos para obter estas informações.

- **Biblioteca de comunicação.** O trabalho propõe uma biblioteca para dar suporte ao desenvolvimento de aplicações que executam em arquiteturas compostas por processadores heterogêneos e programáveis. A biblioteca fornece um conjunto de funções de alto nível para gerenciar a troca de mensagem entre tarefas que executam

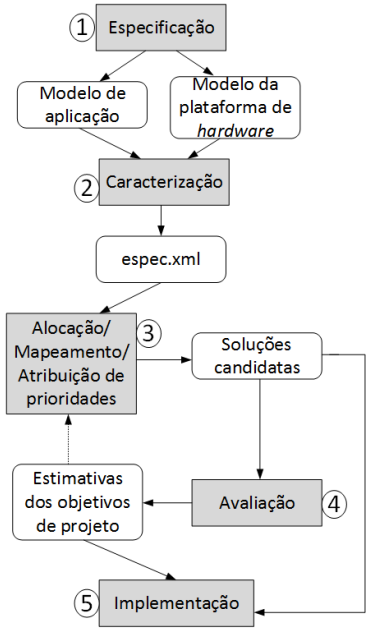


Figura 1.2: Visão geral do método proposto: fluxo de atividades.

em processadores diferentes (ou no mesmo processador). O objetivo principal da biblioteca é dar suporte à semântica do modelo de especificação da aplicação.

1.4 Visão geral do método proposto

A Figura 1.2 apresenta o fluxo de atividades do método proposto. As caixas cinzas representam atividades, e as brancas suas respectivas informações de entrada e saída. Os números representam os passos do método, descritos a seguir:

1. **Especificação:** Dada uma especificação da aplicação (código na linguagem C) e da plataforma de *hardware* na qual ela será implementada, essa atividade consiste na definição desses dois artefatos (aplicação e arquitetura), utilizando os modelos de especificação propostos.

2. **Caracterização:** Nesta atividade, o projetista deve definir informações sobre o tempo de execução das tarefas da aplicação, assim como informações sobre o custo monetário e a potência consumida dos elementos da plataforma de *hardware*. Caso todas essas informações já estejam disponíveis, o projetista pode pular esta atividade.

3. **Alocação, mapeamento e escalonamento:** Depois da atividade de caracterização, um dos algoritmos propostos (MODSES ou C-MODSES) deve ser usado para explorar o espaço de projeto. Esses algoritmos constroem uma solução candidata executando as seguintes subatividades: (i) *alocação*, selecionar que elementos (processadores e barramentos) da plataforma de *hardware* serão utilizados, (ii) *mapeamento*, definir em que elemento de *hardware* cada tarefa da aplicação irá executar, (iii) *atribuição de prioridades*, determinar as prioridades de execução das tarefas, e (iv) *atribuição de frequência*, escolher a frequência de operação da arquitetura.

4. **Avaliação de desempenho:** Nesta atividade, a qualidade de cada alternativa candidata gerada pela atividade anterior é avaliada. A qualidade de uma solução é

avaliada em termos dos seguintes objetivos de projeto: probabilidades de violação de *deadlines*, custo monetário e potência consumida. A atividade de avaliação é conduzida automaticamente pelos algoritmos propostos. É com base nos resultados dessa atividade que os algoritmos decidem se o processo de exploração deve continuar ou não.

5. **Implementação:** Ao final do processo de exploração, os algoritmos propostos retornam um subconjunto representativo do conjunto ótimo de Pareto (DEB, 2008) (ou uma aproximação deste conjunto). Assim, o projetista deve escolher algumas das alternativas de projeto produzidas para implementação final.

1.5 Delimitação

Como mencionado, sistemas embarcados podem ser usados nos mais diversos domínios de aplicação. O alvo deste trabalho são sistemas embarcados que implementam aplicações focadas em dados (*data-oriented*), como aplicações multimídia e de processamento de sinais. Focamos nesta classe de sistemas pois comumente eles possuem apenas *deadlines* não críticos, e, por isso, são classificados como sistemas de tempo-real não críticos (ABENI; MANICA; PALOPOLI, 2012). Ademais, este trabalho considera que a aplicação será implementada apenas em *software*, que executa em processadores programáveis. Essa não é uma restrição muito grande pois, devido a maior flexibilidade das implementações baseadas em *software* em comparação a implementações baseadas em *hardware*, atualmente uma forte tendência é que a grande maioria das funcionalidades de um sistema embarcado seja implementada em *software* (MOYER, 2013; SANGIOVANNI-VINCENTELLI; MARTIN, 2001). Não obstante, faz parte dos nossos objetivos para trabalhos futuros considerar aplicações com foco em controle (*control-oriented*), assim como considerar que partes da aplicação também possam ser implementadas em *hardware* (ex.: FPGA).

1.6 Estrutura da tese

O próximo capítulo apresenta conceitos fundamentais para que esta tese seja melhor compreendida. O Capítulo 3 descreve os trabalhos correlatos. O Capítulo 4 mostra como o projetista deve conduzir as atividades de especificação e caracterização do fluxo proposto. O Capítulo 5 apresenta os modelos propostos para representar sistemas embarcados de tempo-real não críticos. O Capítulo 6 descreve os algoritmos propostos para exploração do espaço de projeto. O Capítulo 7 apresenta os resultados experimentais. Por fim, o Capítulo 8 conclui este trabalho e apresenta direções para futuras pesquisas.

2 Fundamentos

Esse capítulo apresenta um resumo dos conceitos necessários para um melhor entendimento deste trabalho. O capítulo está organizado da seguinte forma: a Seção 2.1 apresenta conceitos importantes sobre as arquiteturas de sistemas embarcados com múltiplos processadores. A Seção 2.2 introduz os conceitos fundamentais sobre otimização que posteriormente serão adotados para definir o problema de exploração tratado e os algoritmos propostos para atacá-lo. A Seção 2.3 apresenta a definição formal de um HSDG (*Homogenous Synchronous Dataflow Graph*), que é o modelo de aplicação adotado por este trabalho. A Seção 2.4 aborda o formalismo DEVS, que é utilizado por este trabalho para avaliar os aspectos temporais dos sistemas embarcados de tempo-real não críticos. Por último, a Seção 2.5 conclui o capítulo.

2.1 Sistemas embarcados com múltiplos processadores

Embora alguns sistemas embarcados possam parecer simples, muitos deles demandam alto poder computacional. Arquiteturas com múltiplos processadores têm sido a melhor forma de se atender aos rígidos requisitos de tempo-real desses sistemas (WOLF, 2014). A utilização de múltiplos processadores é particularmente importante quando o sistema possui restrições de consumo de energia/potência, além das restrições temporais. Uma arquitetura com múltiplos processadores ajuda a evitar o aumento da potência consumida associada a altas frequências do *clock* (BLAKE; DRESLINSKI; MUDGE, 2009). Em sistemas embarcados, a utilização de múltiplos processadores é uma solução natural devido à concorrência inerente a esses sistemas.

Arquiteturas com múltiplos processadores podem ser classificadas em: (i) heterogêneas, e (ii) homogêneas (MOYER, 2013). Nas arquiteturas heterogêneas os processadores são de diferentes tipos e características, já nas arquiteturas homogêneas eles são idênticos. Do ponto de vista do *software* que executa nesses processadores, o sistema pode ser classificado em: (i) AMP - *Asymmetrical Multi-Processing*, e (ii) SMP - *Symmetrical Multi-Processing* (MOYER, 2013) (Figura 2.1). Na configuração SMP, um mesmo sistema operacional (SO) é responsável por gerenciar todos os processadores e, na AMP, cada processador executa seu próprio SO (o processador pode inclusive não ter um SO). Em sistemas que utilizam a configuração SMP, a memória é vista como um grande bloco compartilhado que se parece igual a todos os processadores (MOYER, 2013). Em sistemas AMP, essa restrição não existe e cada processador pode ter uma mistura de memórias locais/compartilhadas.

Esta tese foca em sistemas embarcados projetados para executar um conjunto específico de aplicações. Dado que, neste caso, o projetista sabe de antemão que aplicações irão executar na plataforma de *hardware*, é possível definir uma arquitetura específica e otimizada para atender os requisitos destas aplicações. Nesse contexto, apesar das arquiteturas heterogêneas/AMP serem mais difíceis de programar, elas são bastante eficientes, pois permitem que o projetista possa mapear as diferentes partes da aplicação nos processadores com as características mais

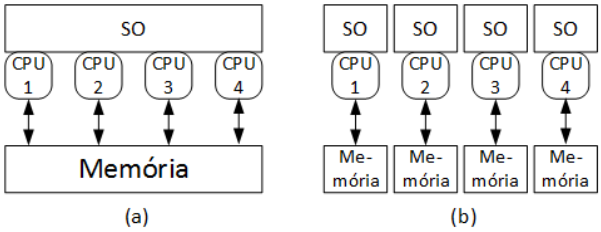


Figura 2.1: (a) *Symmetrical Multi-Processing*. (b) *Asymmetrical Multi-Processing*.

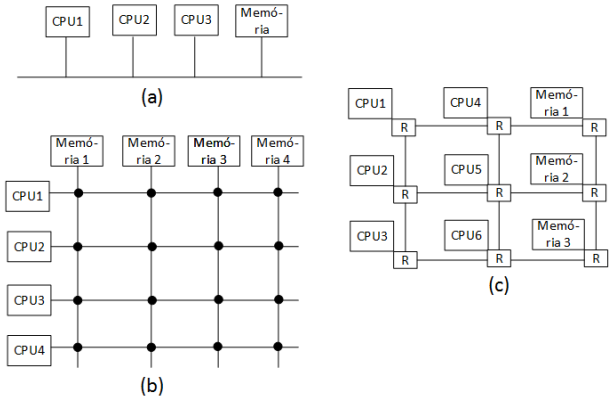


Figura 2.2: (a) Barramento. (b) *Crossbar*. (c) NoC (*network on chip*).

adequadas para executá-las. Além disso, a configuração AMP possibilita que o projetista limite o compartilhamento de memória, dando maior previsibilidade temporal ao sistema, o que é importante para sistemas de tempo-real. Devido a essas características, este trabalho se concentra em sistemas embarcados AMP/heterogêneos ao invés de sistemas SMP.

A comunicação entre os recursos de uma arquitetura multiprocessada pode ser feita usando conexões ponto a ponto, barramentos, *crossbars* e redes de roteamento programável como as NoC (*networks on chip*) (MOYER, 2013) (Figura 2.2). O barramento é uma conexão compartilhada entre um conjunto de emissores e receptores. Um *crossbar* com N linhas de entrada e N linhas de saída provê um caminho de todas as portas de entrada para as portas de saída através de uma matriz N x N (WOLF, 2014). NoCs são interconexões de rede para chips com múltiplos processadores (WOLF, 2014). A ideia principal de uma NoC é usar um conjunto hierárquico de roteadores para permitir mais eficientemente o fluxo de pacotes entre os emissores e receptores (WOLF; JERRAYA; MARTIN, 2008).

2.2 Técnicas de otimização

Otimização é o ato de encontrar a melhor solução nas circunstâncias dadas (RAO; RAO, 2009). As técnicas de otimização possuem uma ampla gama de aplicações, uma vez que quase toda empresa está envolvida na resolução de problemas de otimização. Muitos problemas práticos e teóricos em engenharia, economia e planejamento podem ser modelados convenientemente como problemas de otimização. Por exemplo, empresas de telecomunicações interessadas no melhor projeto das redes de comunicação para otimizar o custo e a qualidade do serviço; empresas que operam no mercado financeiro interessadas na melhor estratégia de alocação de ativos para otimizar o lucro e o risco; empresas de distribuição interessadas na melhor alocação das entregas nos veículos disponíveis de forma a otimizar a distância percorrida pelos veículos.

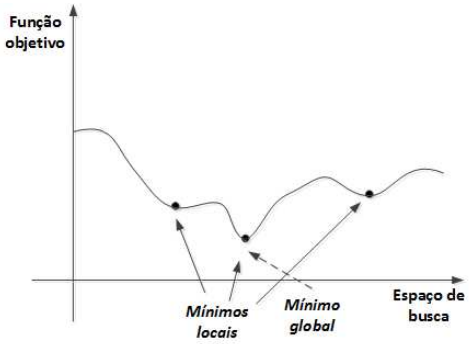


Figura 2.3: Mínimos locais e globais.

2.2.1 Formulação de um problema de otimização

Um problema de otimização pode ser definido de acordo com o seguinte formato[1]:

minimize ou maximize	**Função Objetivo**
sujeito a	**Restrições**

Uma solução é **viável** se ela satisfaz todas as restrições. A função objetivo associa para toda solução viável um número real que indica o valor (qualidade) da solução. Esta função também é chamada de função utilidade ou de aptidão. Considere que S é o conjunto das soluções viáveis e $f : S \to \mathbb{R}$ é a função objetivo a ser otimizada. Define-se:

Definição 2.1 (Ótimo global). Uma solução $s^* \in S$ é um ótimo global se não há nenhuma outra solução que possua melhor valor (maior ou menor) da função objetivo, i.e., considerando um problema de minimização tem-se que $\forall s \in S, f(s^*) \leq f(s)$.

Definição 2.2 (Ótimo local). Seja $N : S \to 2^S$ uma função de vizinhança que mapeia cada solução $s \in S$ em um conjunto de soluções $N(s) \subset S$, então uma solução $s' \in S$ é um ótimo local (ou uma solução sub-ótima) se não há nenhuma outra solução de melhor qualidade em sua vizinhança, i.e., considerando um problema de minimização: $\forall s \in N(s'), f(s') \leq f(s)$ (Figura 2.3). Em particular, a vizinhança de uma solução s em um espaço contínuo é uma bola com centro s e raio igual a ε, sendo $\varepsilon \geq 0$ (TALBI, 2009).

Pela definição, todo ótimo global é também um ótimo local (o inverso pode ser falso), e mais de um ótimo global pode existir. Desta forma, o principal objetivo ao se resolver um problema de otimização é achar um ótimo global ou, caso se queira mais alternativas, todos os ótimos globais.

Diferentes famílias de modelos de otimização podem ser usados para modelar e resolver problemas de otimização. A Figura 2.4 apresenta uma classificação desses modelos (TALBI, 2009). Uma das classes mais difundidas é a **Programação Linear** (LP - *Linear Programming*). Em um problema de LP, a função objetivo, assim como as restrições do problema são funções lineares. Caso seja possível satisfatoriamente modelar um problema utilizando LP, então o analista está em boas mãos pois existem métodos bastante eficientes para resolver esse tipo de problema, como o método simplex e o método de pontos interiores (HILLIER, 1990).

Uma classe de modelos de otimização de particular importância para este trabalho são os **modelos não analíticos**. Apesar de existirem inúmeras formas de se formular um problema

[1]Mais precisamente, essa é a formulação de um problema mono-objetivo. A Subseção 2.2.4 apresenta a formulação de um problema multiobjetivo.

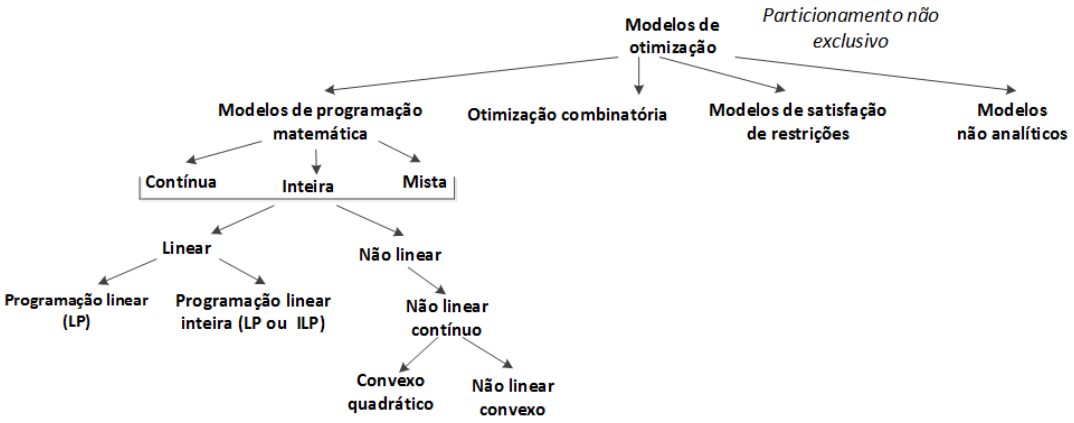


Figura 2.4: Classificação dos modelos de otimização (particionamento não exclusivo) (TALBI, 2009).

matematicamente, para diversos problemas práticos uma formulação matemática explícita é muito difícil, o que faz com que seja impossível modelá-los utilizando modelos de programação matemática ou de satisfação de restrições (TALBI, 2009). Problemas em que é necessário utilizar um modelo de simulação ou físico para determinar a qualidade de uma solução são exemplos proeminentes desse tipo de problema.

Exemplo 2.1 (***Call center*** **(FU, 2002)).** *Considere o projeto de um* call center *simplificado com apenas um operador, e que se deseja escolher o nível ótimo de habilidade do operador considerando os custos envolvidos. Um operador mais habilidoso custa mais caro, porém atende mais rápido melhorando, assim, a qualidade de serviço.*

Assuma que esse call center *pode ser modelado como uma fila (BOLCH et al., 2006) com um servidor, que os intervalos de chegada entre os clientes são independentes e identicamente distribuídos (com uma distribuição arbitrária) e que os tempos de serviço dos clientes também são independentes e identicamente distribuídos (com uma distribuição arbitrária). Além disso, a fila tem capacidade ilimitada e possui política FIFO (primeiro a entrar é o primeiro a sair). Seja x o tempo médio de serviço do atendente (de forma que $1/x$ é a velocidade de atendimento). Então esse problema é geralmente modelado da seguinte forma:*

$$\textit{minimize} \quad f(x) = W(x) + c/x,$$

onde $W(x)$ é o tempo médio que o cliente precisa esperar no sistema até ser atendido, ou seja, é igual ao tempo médio de espera na fila mais o tempo médio de serviço, e c é um fator de custo para a velocidade do atendente. Uma vez que $W(x)$ aumenta com x, a função objetivo captura o trade-off *entre a qualidade do serviço e o custo de fornecer o serviço. Apesar desse modelo de fila ter sido bastante estudado, simulação estocástica ainda é necessária em muitos casos para estimar $W(x)$ (FU, 2002). Para simular esse sistema é necessário informar as distribuições de probabilidade dos tempos de chegada e de serviço.*

Um outro modelo de otimização importante para este trabalho e que não é mostrado na Figura 2.4, são os modelos que modelam problemas com múltiplos objetivos. Esse tipo de modelo será tratado com maiores detalhes na Subseção 2.2.4.

Figura 2.5: Técnicas clássicas de otimização (TALBI, 2009).

2.2.2 Atacando problemas intratáveis

Em geral, considera-se que um problema é tratável (ou fácil) se existe algum algoritmo para resolvê-lo em tempo polinomial, i.e., se o tempo de execução do melhor algoritmo que resolve o problema no pior caso é $O(n^k)$ para alguma constante k, onde n é o tamanho da entrada do problema. Considera-se um problema **intratável** (ou difícil) se não existe nenhum algoritmo capaz de resolvê-lo em tempo polinomial (LEISERSON et al., 2001).

Pesquisadores da área de teoria computacional já sabem há algum tempo que existe uma classe especial de problemas, chamada de **NP-completo**, que, acredita-se, é composta por problemas que não podem ser resolvidos em tempo polinomial, embora isso ainda nunca tenha sido provado. Esta questão, chamada de *P versus NP*, é um dos maiores problemas em aberto da computação. Qualquer problema NP-completo pode ser reduzido em tempo polinomial para qualquer outro problema NP-completo. Isto significa que, caso algum algoritmo de tempo polinomial fosse descoberto para um problema NP-completo, então também seria possível resolver todos os outros problemas NP-completo em tempo polinomial. Existem centenas de problemas NP-completo, e depois de décadas de estudo ninguém foi capaz de desenvolver um algoritmo em tempo polinomial para nenhum deles. Parece razoável, portanto, acreditar que, caso seja possível estabelecer que um dado problema é NP-completo, ou que ele é no mínimo tão complexo (ou tão "difícil" (LEISERSON et al., 2001)) quanto um problema NP-completo, então isto implica que existe uma boa evidência de que ele é intratável.

A Figura 2.5 apresenta uma classificação das técnicas tradicionais que podem ser adotadas para resolver problemas de otimização (TALBI, 2009). Os **métodos exatos** garantem que um ótimo global será encontrado, já os **métodos aproximados** acham soluções de boa qualidade, porém não garantem que a solução encontrada será um ótimo global. Em muitas situações, um algoritmo exato pode demorar muito para encontrar uma solução ótima. Dessa forma, às vezes uma solução de boa qualidade, mesmo que não seja comprovadamente a melhor, é suficiente, considerando o tempo e os recursos computacionais disponíveis. Considere, por exemplo, o problema de encontrar a melhor rota entre dois lugares usando para isso um GPS. Como o grafo que representa o problema é grande e o tempo de busca é restrito (uma vez que o usuário deseja obter resultados rapidamente), na prática, mesmo que esse problema possa ser resolvido em tempo polinomial (ex.: algoritmo de Dijkstra (LEISERSON et al., 2001)), os algoritmos que atualmente executam nos GPS fazem uso de algoritmos aproximados (TALBI, 2009).

A não ser que $P = NP$, a aplicação de métodos exatos em problemas NP-completo provavelmente resultará em algoritmos de tempo não polinomial. Porém, quando é possível garantir que as entradas para o problema serão sempre pequenas, um algoritmo de tempo não

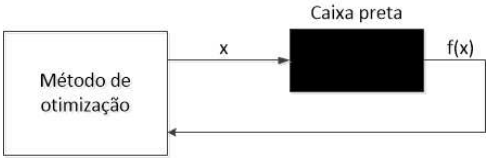


Figura 2.6: Função objetivo vista como uma caixa-preta para método de otimização.

polinomial pode ser viável. Outra possibilidade é tentar isolar casos especiais do problema e, assim, desenvolver algoritmos polinomiais que tratam esses casos. Quando as duas opções anteriores não são aplicáveis, os métodos aproximados são uma boa alternativa.

A qualidade das soluções encontradas por um algoritmo aproximado vai depender, obviamente, da qualidade do algoritmo usado no problema. Porém, dentro da classe dos métodos aproximados é possível diferenciar duas classes: os **algoritmos de aproximação** e as **heurísticas**. Os algoritmos de aproximação oferecem garantias com relação à qualidade das soluções encontradas. Eles são algoritmos que precisam ser desenvolvidos especificamente para o problema a ser tratado. Na prática, porém, as aproximações que são possíveis ficam muito distantes das soluções ótimas, fazendo com que esses algoritmos não sejam muito úteis para muitos problemas reais (TALBI, 2009).

Definição 2.3 (**Algoritmos α-aproximado (WILLIAMSON; SHMOYS, 2011)**). Um algoritmo de α-aproximação é um algoritmo de tempo polinomial que para qualquer instância do problema produz uma solução cujo valor está dentro de um fator α do valor da solução ótima.

Assim, um algoritmo de $\frac{1}{2}$-aproximação para um problema de maximização é um algoritmo que comprovadamente executa em tempo polinomial e retorna soluções que são, no mínimo, metade do valor máximo (ótimo).

As heurísticas não oferecem nenhuma garantia com relação à qualidade das soluções. Dessa forma, tipicamente a qualidade das soluções geradas por esses algoritmos, assim como o seu tempo de execução são avaliados empiricamente por meio de *benchmarks* compostos de instâncias específicas do problema tratado. As heurísticas são divididas em **heurísticas específicas** e **meta-heurísticas**. As primeiras são desenvolvidas especificamente para o problema ou para casos específicos do problema, já as últimas podem ser vistas como *templates* que podem ser usados para auxiliar na criação de heurísticas para resolver problemas específicos.

Nos últimos anos, as meta-heurísticas ganharam bastante popularidade devido à sua eficiência em resolver problemas de grande importância e complexidade. Exemplos de meta-heurísticas incluem os algoritmos genéticos, arrefecimento simulado (*simulated annealing*), otimização por colônia de formigas, otimização por enxame de partículas (*particle swarm optimization*), GRASP, busca tabu, dentre outras.

As meta-heurísticas são particularmente interessantes para atacar problemas de otimização envolvendo simulação, pois não é difícil integrar um modelo de simulação a uma meta-heurística já que maioria delas requer apenas os valores da função objetivo. Assim, para a meta-heurística, o modelo de simulação pode ser visto como uma caixa-preta (Figura 2.6), cujos detalhes ela não precisa saber. Em outras palavras, para a meta-heurística o modelo de simulação é apenas um mecanismo que transforma parâmetros de entrada x em medidas de desempenho $f(x)$ (APRIL et al., 2003). Os procedimentos de otimização de quase todas as ferramentas comerciais de simulação são baseados em meta-heurísticas (FU; GLOVER; APRIL, 2005; OPTTEK SYSTEMS, 2013).

2.2.3 Algoritmos genéticos

Os **algoritmos genéticos** foram propostos por J. H. Holland na década de 70 (HOLLAND, 1975) e fazem parte de uma área da computação, chamada de computação evolucionária, que se inspira nos processos naturais da evolução para resolver problemas (GEN; CHENG, 2000; EIBEN; SMITH, 2003). Nos algoritmos genéticos, um **indivíduo** é uma solução codificada para o problema, e um conjunto de indivíduos é denominado de **população**. O Algoritmo 1 exibe o esquema geral de um algoritmo genético.

```
1  t := 0
2  Initialize(P_t)                                   // cria a população inicial
3  Evaluate(P_t)                                     // avalia as soluções
4  while StopCriterion(t,P_t) == false do // enquanto o critério de parada não for satisfeito
5  |  t := t + 1
6  |  P' := Select(P_{t-1})                          // seleciona as soluções para crossover
7  |  O := Crossover(P')                             // realiza crossover nas soluções selecionadas
8  |  O := Mutation(O)                               // realiza mutação nas soluções
9  |  Evaluate(O)                                    // avalia as novas soluções
10 |  P_t := P_{t-1} ∪ O
11 |  Terminate(P_t)          // remove as soluções que não pertencerão a nova geração
```

Algoritmo 1: Esquema geral de um algoritmo genético.

O algoritmo inicia criando uma população inicial, P_t (linha 1), de indivíduos (ou soluções). Em seguida, cada indivíduo da população tem sua aptidão avaliada (linha 2). Os indivíduos mais aptos, P', são selecionados e recombinados para gerar novas soluções, O (linha 6-8). Dois operadores são usados para gerar os filhos: (i) ***crossover***, na qual um par de indivíduos é mesclado para gerar outros indivíduos, e (ii) **mutação**, na qual um indivíduo é levemente modificado. As novas soluções, chamadas de **filhos**, são então avaliadas (linha 9). Depois, uma nova população (uma nova **geração** da população) é criada, copiando os indivíduos da geração atual e seus filhos (linha 10). Em seguida, os indivíduos menos aptos da nova geração são removidos da população (linha 11). Após diversas gerações, o algoritmo converge para o indivíduo mais apto, que, espera-se, represente um ótimo global ou local para o problema.

Pode-se destacar os seguintes componentes como os mais importantes de um algoritmo genético:

- **Codificação**: A representação genética das soluções do problema;
- **População inicial**: Mecanismo de criação da população inicial;
- **Função de avaliação (ou função de aptidão[2])**: A função que atribui ao indivíduo um valor que determina a sua aptidão;
- **Operadores de recombinação**: Os operadores de mutação e *crossover*;
- **Seleção dos pais**: Mecanismo de seleção dos pais que passarão pelos operadores de recombinação para gerar novas soluções;
- **Seleção dos sobreviventes**: Mecanismo de seleção dos indivíduos que permanecerão na geração seguinte;
- **Critério de parada**: As condições que fazem o algoritmo terminar.

[2]Do inglês *fitness function*.

Indivíduo 1	1	1	1	0	0	1	0
Indivíduo 2	1	0	0	0	1	0	0

Figura 2.7: Representação binária de dois indivíduos.

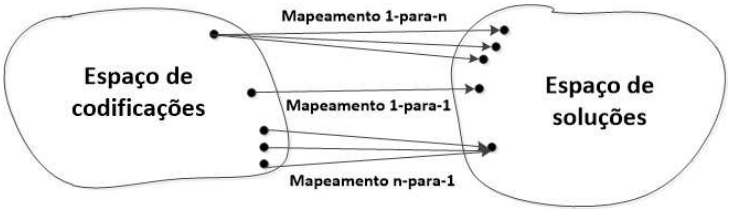


Figura 2.8: Tipos de mapeamento entre códigos e soluções.

Todos esses componentes precisam ser especificados para definir um algoritmo genético em particular. Além disso, é necessário especificar também os parâmetros que controlam o algoritmo, como o tamanho da população e número de filhos que são gerados em cada geração.

2.2.3.1 Codificação

A representação genética é um dos componentes mais importantes para a eficácia de um algoritmo genético. Na proposta inicial de J. H. Holland, os indivíduos eram codificados por *strings* de bits (Figura 2.7). Cada bit representa uma característica da solução, ou uma outra possibilidade é que um conjunto de bits represente um número. Desde a proposição inicial de Holland diversos outros algoritmos genéticos foram propostos usando as mais diversas codificações: números inteiros, número reais, *arrays*, listas, árvores, dentre outras estruturas de dados (SIVANANDAM; DEEPA, 2007).

Considere por exemplo o problema de encontrar o conjunto máximo de vértices independentes em um grafo $G = (V, E)$, onde V é o conjunto dos vértices e E o conjunto dos arcos (FEO; RESENDE; SMITH, 1994). Segue uma possível representação para esse problema utilizando um *string* de binários. Seja sol um *string* de tamanho $|V|$, tal que $sol[i] = 1$ se o vértice i pertence ao conjunto máximo de vértices independentes, e $sol[i] = 0$, caso contrário.

Vários princípios foram propostos para garantir que uma dada codificação resulte em um algoritmo genético eficaz (GEN; CHENG, 2000):

- **Não redundância**: O mapeamento entre um código e uma solução deve ser 1-para-1. A Figura 2.8 apresenta os tipos de mapeamento que podem existir entre os códigos e as soluções. Se é possível um mapeamento n-para-1 então o algoritmo genético pode perder tempo na busca já que existirão indivíduos repetidos no espaço de busca. O pior mapeamento é o 1-para-n, pois terá de haver um procedimento auxilar para determinar uma solução dentre as várias possíveis;

- **Legalidade**: Qualquer permutação de uma codificação deve corresponder a uma solução. Esta propriedade garante que os operadores genéticos podem ser facilmente aplicados nos indivíduos, sem que isso gere soluções inválidas (GEN; CHENG, 2000). Usualmente técnicas de reparo precisam ser usadas para reparar códigos inviáveis e transformá-los em códigos viáveis;

- **Propriedade Lamarckiana**: A codificação deve garantir que as características importantes de um indivíduo possam ser repassadas para seus filhos;

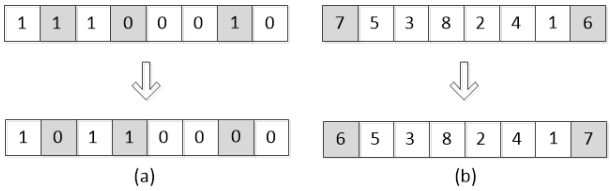


Figura 2.9: Dois exemplos de operadores de mutação: (a) Operador que inverte os bits de um *string*. (b) Operador que permuta os valores de duas posições de um *string*.

- **Completude**: Qualquer solução deve possuir uma codificação. Isto garante que todo o espaço de soluções é acessível pela busca;
- **Causalidade**: Pequenas variações no código devido a mutações também devem gerar pequenas variações na solução resultante. Esta propriedade faz com que a estruturas de vizinhança sejam preservadas.

2.2.3.2 Operadores de recombinação

O papel dos operadores de *crossover* e mutação é criar novos indivíduos a partir dos antigos. O objetivo do operador de *crossover* é mesclar características diferentes mas desejáveis dos pais e passá-las para os filhos que, espera-se, sejam mais aptos que os pais. No caso do operador de mutação, a ideia é adicionar mais diversidade à população com o intuito de fazer com que a busca aponte para lugares inexplorados. Ambos são operadores estocásticos: o código genético dos filhos vai depender de uma série de eventos aleatórios.

A Figura 2.9 exibe dois exemplos de operadores de mutação. O operador da Figura 2.9a inverte os bits de posições aleatórias do *string*, e o operador da Figura 2.9b seleciona aleatoriamente duas posições do *string* e permuta seus valores.

A Figura 2.10 exibe três exemplos de operadores de *crossover*. O primeiro operador é denominado de *crossover* de um ponto (Figura 2.10a). Esse operador seleciona aleatoriamente uma posição do *string*, depois corta os pais nessa posição e cria os filhos permutando os *substrings* criados depois dos cortes. O segundo operador é uma variação do primeiro e é denominado de *crossover* de dois pontos (Figura 2.10b). Nesse operador dois pontos são selecionados aleatoriamente e os pais são cortados nesses pontos, em seguida os filhos são criados permutando os *substrings* do meio.

Os operadores de *crossover* anteriores não são adequados para representações baseadas em permutações, pois note que, para os dois operadores, o *crossover* gerou filhos com valores repetidos em seus *strings*. Considere, por exemplo, o problema do caixeiro viajante, cujo objetivo é encontrar a melhor rota para visitar apenas uma vez uma série de cidades e, no fim, retornar para a cidade de origem (DORIGO; GAMBARDELLA, 1997). Suponha que existam n cidades para serem visitadas, então a seguinte codificação utilizando um *string* de inteiros poderia ser adotada. Seja sol um *string* de tamanho n, tal que $sol[i] = j$ indica que a cidade j deve ser a i-ésima cidade a ser visitada.

O operador chamado de *crossover* parcialmente mapeado pode ser usado para evitar o problema anterior (Figura 2.10c). Esse operador funciona como um *crossover* de dois pontos seguido por um procedimento de reparo que evita que valores repetidos apareçam nos *strings* dos filhos. Na Figura 2.10c, os valores circulados indicam que eles são inválidos pois aparecem mais de uma vez no *string*. Esses valores são substituídos de acordo com uma tabela de referências, que pode ser criada associando os valores que ocupam as mesmas posições nos pais. Por exemplo, os valores 3 e 7 estão associados porque os dois valores ocupam a posição 3 nos pais.

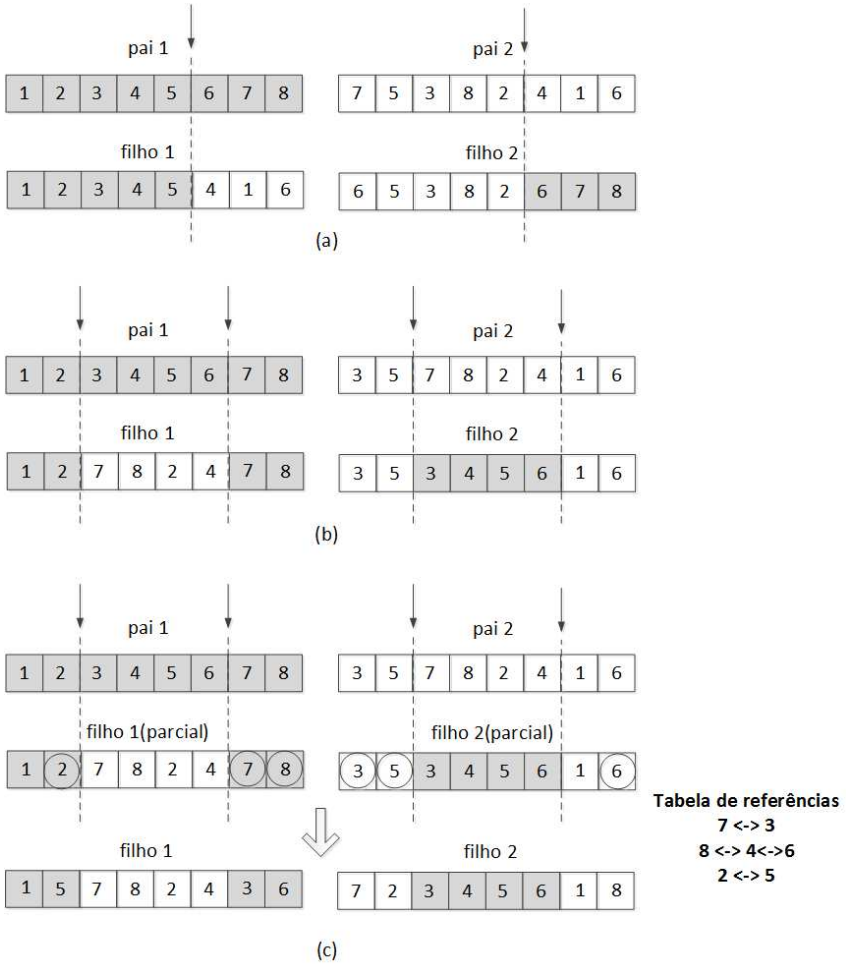


Figura 2.10: Três exemplos de operadores de *crossover*: (a) *Crossover* de um ponto. (b) *Crossover* de dois pontos. (c) *Crossover* parcialmente mapeado.

Indivíduo	Aptidão	Aptidão relativa
i1	2,5	23,81%
i2	2	19,05%
i3	1	9,52%
i4	5	47,62%

Figura 2.11: Indivíduos de uma população e sua respectiva seção na roleta.

Os operadores de *crossover* e mutação são muito importantes para a eficácia de um algoritmo genético. Se esses operadores geram apenas soluções aleatórias sem nenhum vínculo com as soluções anteriores, o algoritmo genético não estará fazendo nada mais que uma busca aleatória (SIVANANDAM; DEEPA, 2007).

2.2.3.3 Seleção e função de aptidão

O papel da função de aptidão é definir uma medida de qualidade para as soluções. Tipicamente ela é igual à função objetivo. A função de aptidão constitui a base para os mecanismos de seleção, determinando, assim, o que significa uma melhora (EIBEN; SMITH, 2003).

Todo algoritmo genético possui dois mecanismos de seleção de indivíduos. O primeiro mecanismo define quais indivíduos serão recombinados para gerar novos indivíduos (linha 6 do Algoritmo 1), e o segundo determina quem serão os indivíduos que passarão para a geração seguinte (linha 11 do Algoritmo 1). Assim como os operadores de recombinação, estas seleções geralmente são feitas de forma estocástica: o princípio adotado é que, embora seja desejável escolher os indivíduos mais aptos, deve existir uma chance pequena para que os indivíduos menos aptos também sejam selecionados. A seleção dos indivíduos mais aptos é responsável por forçar a melhora na qualidade das soluções durante as sucessivas gerações. Deixar que em algumas vezes os indivíduos menos aptos também sejam escolhidos é um principio fundamental das meta-heurísticas, pois isso impede que o algoritmo fique preso em regiões de mínimos locais.

Existem diversas formas de implementar os mecanismos de seleção. Uma das formas mais tradicionais é a seleção por roleta. A seleção por esta abordagem simula uma roleta com a seguinte configuração: o lado de cada seção da roleta é proporcional ao valor de aptidão de cada indivíduo, portanto, quanto maior for esse valor, mais larga a seção (Figura 2.11). Depois que a roleta é repartida, ela é girada e o indivíduo sorteado é selecionado. Uma outra forma de seleção é a seleção por torneio de tamanho k. Nessa abordagem, k indivíduos são selecionados aleatoriamente e o mais apto é escolhido. Caso mais de um tenha o valor máximo de aptidão, então um deles é selecionado aleatoriamente com mesma probabilidade de seleção.

2.2.3.4 População inicial e critério de parada

Tradicionalmente, a geração da população inicial é feita de forma simples: os indivíduos iniciais são aleatoriamente gerados (EIBEN; SMITH, 2003). Porém, também é comum que heurísticas específicas para o problema sejam adotadas para garantir que os indivíduos iniciais tenham boa aptidão.

Vários critérios podem ser adotados como condição de parada para o algoritmo. Eles podem ser adotados isoladamente ou em conjunto (SIVANANDAM; DEEPA, 2007; EIBEN; SMITH, 2003):

- **Número máximo de gerações**: O algoritmo para depois que um dado número máximo de gerações foi atingido.

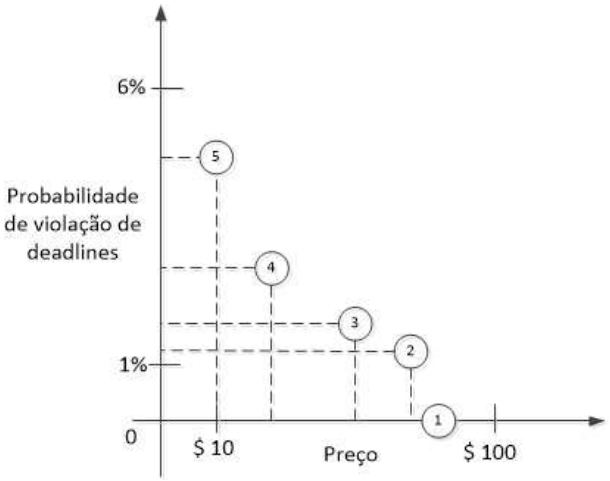


Figura 2.12: Possíveis alternativas de projeto para um sistema embarcado não crítico.

- **Tempo de execução**: Depois que o algoritmo executou por um tempo máximo ele para.
- **Sem mudanças na aptidão da população**: O algoritmo para se depois de um certo número de gerações não houve melhora na aptidão do melhor indivíduo da população.
- **Número de avaliações**: Depois de um dado número de avaliações da função de aptidão o algoritmo para.

2.2.4 Otimização multiobjetivo

A maioria dos problemas reais é composta por diversos objetivos conflitantes que precisam ser otimizados simultaneamente. Considere o problema de escolher a melhor opção de projeto de sistema embarcado não crítico dentre as opções da Figura 2.12. Se a minimização da violação de *deadlines* fosse o único objetivo, então a alternativa 1 seria a solução ótima. No entanto, como discutido anteriormente, a escolha envolve diversos outros objetivos, sendo o custo um desses objetivos. Normalmente um *hardware* mais rápido custa mais caro, porém um *hardware* mais rápido pode fazer com que a violação de *deadlines* diminua. Se o custo também for considerado, então, note que entre quaisquer duas alternativas da figura, uma é melhor em algum objetivo e pior em outro. Desta forma, não é possível afirmar que uma dessas alternativas é melhor que a outra. Esta é uma característica fundamental dos problemas de otimização multiobjetivo: diferentemente dos problemas mono-objetivo, em que o objetivo é encontrar uma solução superior a todas as outras alternativas, em geral para os problemas multiobjetivo não existe uma solução única que seja ótima em todos os objetivos ao mesmo tempo, e sim um conjunto de soluções ótimas que são definidas como o **conjunto ótimo de Pareto**. Uma solução é um ótimo de Pareto se não é possível melhorar algum objetivo sem que outro seja piorado.

Um problema de otimização multiobjetivo pode ser formulado como (TALBI, 2009):

$$\begin{aligned} &\text{minimize/maximize} && F(x) = (f_1(x), f_2(x), \ldots, f_n(x)) \\ &\text{sujeito a} && x \in S, \end{aligned}$$

onde $x = (x_1, x_2, \ldots, x_k)$ é o vetor que representa as variáveis de decisão e S é o conjunto das soluções viáveis, considerando as restrições do problema. $F(x) = (f_1(x), f_2(x), \ldots, f_n(x))$ é a função vetorial cujos elementos representam as n funções objetivo que precisam ser minimizadas

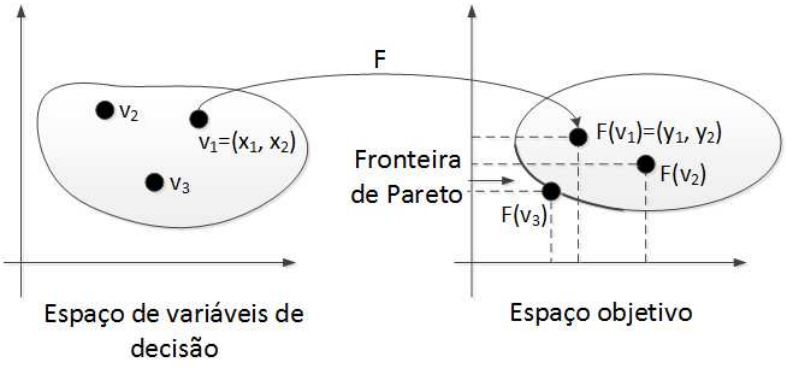


Figura 2.13: Espaços de busca: espaço de variáveis de decisão x espaço objetivo.

ou maximizadas. Sem perda de generalidade, nas definições/conceitos apresentados a seguir assume-se problemas que envolvem a minimização de todos os objetivos.

Outra característica fundamental dos problemas multiobjetivo é que é preciso lidar com dois espaços de busca (DEB, 2008; COELLO; LAMONT, 2007): o **espaço de variáveis de decisão**, S, e o **espaço objetivo**, Λ. Esses dois espaços estão relacionados pela função vetorial $F : S \rightarrow \Lambda$ que mapeia variáveis de decisão ($x = (x_1, x_2, ..., x_k)$) em um vetor ($y = (y_1, y_2, ..., y_n)$) que representa a qualidade da solução (Figura 2.13). A partir da função F, quaisquer duas soluções do espaço de decisão podem ser selecionadas e comparadas.

Definição 2.4 (Dominância de Pareto). Um vetor $u = (u_1, u_2, ..., u_n)$ é menor que outro vetor $v = (v_1, v_2, ..., v_n)$ (denotado por $u < v$) se, e somente se, nenhum componente de u é maior que v e, adicionalmente, existe pelo menos um componente de u que é menor que v, i.e., $\forall i \in \{1, ..., n\},\ u_i \leq v_i \wedge \exists i \in \{1, ..., n\},\ u_i < v_i$. Uma solução x_1 domina outra solução x_2 (denotado por $x_1 \prec x_2$) se, e somente se, $F(x_1) < F(x_2)$.

Na Figura 2.13 é possível observar, por exemplo, que a solução v_3 domina (ela é melhor que) as soluções v_1 e v_2. É possível observar também que a solução v_2 não domina a solução v_1, e que a solução v_1 não domina a solução v_2.

Definição 2.5 (Dominância fraca). Um vetor $u = (u_1, u_2, ..., u_n)$ é menor ou igual a outro vetor $v = (v_1, v_2, ..., v_n)$ (denotado por $u \leq v$) se, e somente se, todos os componentes de u são menores ou iguais a v, i.e., $\forall i \in \{1, ..., n\},\ u_i \leq v_i$. Uma solução x_1 domina fracamente outra solução x_2 (denotado por $x_1 \preceq x_2$) se, e somente se, $F(x_1) \leq F(x_2)$.

Definição 2.6 (Conjunto ótimo de Pareto). Uma solução x pertence ao conjunto ótimo de Pareto, P^*, se não existe nenhuma outra solução no espaço de soluções viáveis que a domine, i.e., $P^* = \{x \in S \mid \nexists x' \in S,\ x' \prec x\}$.

Definição 2.7 (Fronteira de Pareto). Para um dado conjunto ótimo de Pareto, P^*, a fronteira de Pareto é o conjunto $PF^* = \{u = F(x) \mid x \in P^*\}$ (Figura 2.13).

2.2.5 Métodos para resolver problemas multiobjetivo

Assim como acontece nos problemas mono-objetivo, existem métodos exatos e aproximados para resolver problemas multiobjetivo (TALBI, 2009). No entanto, tipicamente métodos exatos não são eficazes pois em geral problemas multiobjetivo são NP-completos (COELLO; LAMONT, 2007).

Embora a fronteira de Pareto seja composta por múltiplas soluções ótimas, do ponto de vista prático, o analista precisa de apenas uma solução. Dessa forma, existem basicamente dois tipos de abordagens que podem ser usadas pelas técnicas que resolvem problemas multiobjetivo (DEB, 2008; GEN; CHENG, 2000): (i) abordagem geracional, e (ii) abordagem baseada em preferências. Na abordagem geracional, múltiplas soluções (ou aproximações) da fronteira de Pareto são geradas para o analista, que então as compara e escolhe uma. Além de encontrar soluções da fronteira de Pareto, também é importante que o conjunto de soluções gerado seja o mais diverso possível, pois isto permite que o analista possa ter um melhor entendimento das relações de compromisso (ou *trade-offs*) existentes entre os objetivos do problema. Nessa abordagem, portanto, não é necessário nenhum conhecimento prévio sobre a importância relativa de cada objetivo. Por outro lado, na abordagem baseada em preferências, algum conhecimento sobre a importância relativa de cada objetivo é usado durante busca. A forma mais simples de se definir matematicamente a importância de cada objetivo seria formular uma função objetivo como sendo a soma ponderada dos objetivos, em que o peso de um objetivo em particular é proporcional a sua importância relativa. A Figura 2.14 apresenta como estas duas abordagens podem ser utilizadas.

A vantagem da abordagem geracional está no fato de que o analista não precisa fornecer qualquer informação previamente. Depois que as soluções ótimas forem geradas, conhecendo a fronteira de Pareto ele pode selecionar aquela que oferece o melhor *trade-off* entre os objetivos. Na abordagem baseada em preferências, o analista precisa fornecer de antemão as importâncias relativas de cada função, o que em muitos casos não é uma tarefa simples principalmente quando não se sabe quais são os possíveis *trade-offs* entre os objetivos. Além disso, outra dificuldade é que pequenas variações nos valores das importâncias relativas podem resultar em grandes mudanças nas solução encontrada. Assim, o analista é obrigado a fornecer esses valores sem nenhum conhecimento das possíveis consequências. Por outro lado, caso valores confiáveis para as importâncias relativas estejam disponíveis, então a abordagem baseada em preferências é mais indicada pois ela evitaria o alto custo computacional da abordagem geracional.

Do ponto de vista das técnicas de solução, a maioria dos métodos tradicionais convertem o problema multiobjetivo para um problema mono-objetivo (ou uma sequência de problemas mono-objetivo), que depois é resolvido usando métodos para problemas mono-objetivo. Os métodos mais modernos, por outro lado, incorporam o conceito de dominância de Pareto na função objetivo, fazendo com que não seja necessário converter o problema multiobjetivo para um problema mono-objetivo. A seguir, algumas dessas técnicas são apresentadas (DEB, 2008; COELLO; LAMONT, 2007; TALBI, 2009).

2.2.5.1 Métodos tradicionais

Soma ponderada: Nesta técnica, o problema é formulado somando e ponderando os múltiplos objetivos:

$$\begin{aligned} \text{minimize} \quad & F(x) = \sum_{i=1}^{n} w_i f_i(x) \\ \text{sujeito a} \quad & x \in S, \end{aligned}$$

onde w_i é o peso de cada função objetivo. Esses pesos devem ser fornecidos pelo analista. A solução da equação acima sempre gera uma solução pertencente à fronteira de Pareto caso $w_i > 0$ para todo i. Porém, quando o espaço objetivo não é convexo esse método não garante que todas as soluções ótimas são alcançáveis.

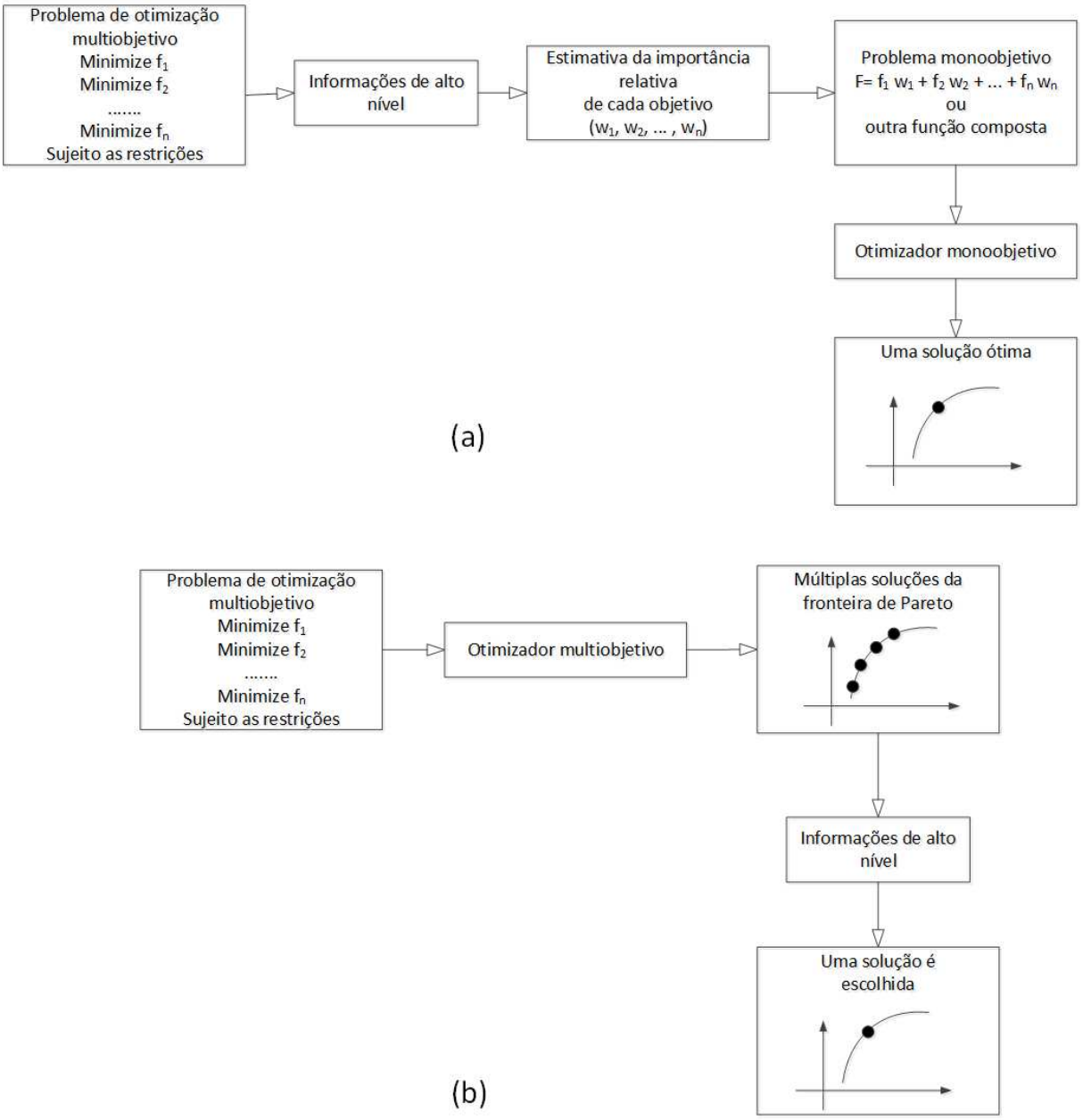


Figura 2.14: (a) Abordagem baseada em preferências. (b) Abordagem geracional.

Método ε-restrito (ε-*Constraint method*): Neste método, o problema multiobjetivo é substituído por um problema mono-objetivo mantendo apenas uma das funções objetivo e tratando as outras como restrições:

$$\begin{aligned} &\text{minimize} && f_w(x) \\ &\text{sujeito a} && x \in S \\ & && f_i(x) \leq \varepsilon_i \quad i = 1,2,\ldots,n \ \text{ e } \ i \neq w, \end{aligned}$$

como no método anterior, os valores de ε_i devem ser fornecidos pelo projetista. A vantagem em relação ao método anterior é que ele garante que todas as soluções da fronteira de Pareto são alcançáveis.

Programação por metas (*Goal programming*): Neste método, o projetista define os pontos que ele espera atingir, $\bar{z}_i$, para cada objetivo, f_i, definindo, assim, as metas $f_i(x) \leq \bar{z}_i$. Busca-se então minimizar a diferença entre os objetivos e as metas, δ_i. Uma variação clássica desta abordagem é programação por metas ponderada (*weighted goal programming*), na qual as diferenças são ponderadas:

$$\begin{aligned} &\text{minimize} && \sum_{i}^{n} w_i \delta_i \\ &\text{sujeito a} && x \in S \\ & && f_i(x) - \delta_i \leq \bar{z}_i \quad i = 1,2,\ldots,n \\ & && \delta_i \geq 0 \qquad\quad i = 1,2,\ldots,n \end{aligned}$$

2.2.5.2 Métodos modernos

Nas últimas duas décadas, métodos baseados nos algoritmos genéticos (ou evolucionários) emergiram como uma das formas mais eficientes para resolver problemas multiobjetivo. O uso dos algoritmos evolucionários para resolver esse tipo de problema foi motivado principalmente por eles serem baseados em populações de soluções (COELLO; LAMONT, 2007). Como a cada iteração uma população de soluções é processada, o resultado final do algoritmo também é uma população de soluções. Se o problema possui apenas um ótimo, então espera-se que os membros da população convirjam para esta solução ótima. Porém, quando o problema possui diversos ótimos, o algoritmo pode ser usado para fazer com que população final seja composta por várias soluções ótimas (DEB, 2008). Os algoritmos genéticos, portanto, podem ser usados para tentar encontrar múltiplas soluções ótimas em apenas uma execução. Nos métodos tradicionais, para achar múltiplas soluções ótimas a busca precisa ser repetida diversas vezes, variando, por exemplo, os pesos e restrições da função objetivo.

Basicamente, a única diferença dos algoritmos genéticos multiobjetivo para os algoritmos genéticos mono-objetivos está na forma como o valor de aptidão das soluções é computado e atribuído (COELLO; LAMONT, 2007). Nos algoritmos genéticos multiobjetivo, ao invés de uma é necessário computar k funções de aptidão, onde k é o número de objetivos do problema. Adicionalmente, como nos algoritmos genéticos a aptidão de um indivíduo é definida por um valor único, é necessário converter esse vetor de aptidões para um escalar.

Existem diversos métodos de conversão do vetor de aptidões. O objetivo desses métodos de conversão é guiar o algoritmo para que a busca: (i) encontre o real conjunto ótimo de Pareto, e (ii) garanta uma boa diversidade nas soluções encontradas. Dentre os métodos para garantir o primeiro objetivo está a abordagem chamada de contagem de dominância (*dominance*

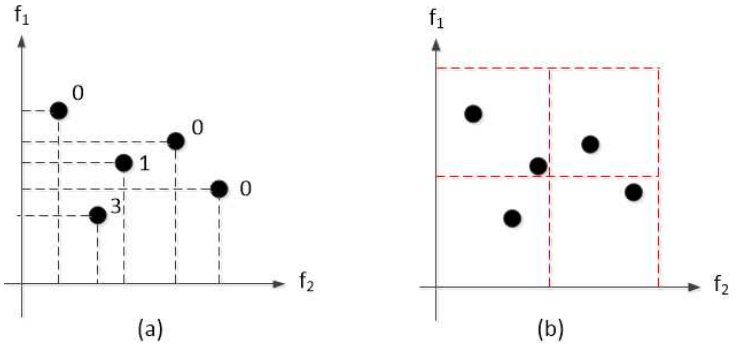


Figura 2.15: (a) Contagem de dominância. (b) Abordagem por histogramas.

count), na qual a aptidão de uma solução está relacionada à quantidade de soluções que ela domina (Figura 2.15a). Um exemplo de método que pode ser usado para garantir o segundo objetivo é a abordagem por histogramas, na qual o espaço objetivo é particionado em diversos *hipergrides*, definindo a vizinhança (Figura 2.15b). Nessa abordagem, a aptidão de uma solução está relacionada à densidade de soluções em cada *grid*: uma solução sem muitos vizinhos recebe aptidão maior que uma solução com muitos vizinhos.

NSGA-II (DEB et al., 2002) e SPEA2 (ZITZLER; LAUMANNS; THIELE, 2001) são exemplos proeminentes de algoritmos genéticos multiobjetivo.

2.2.5.3 Medidas de qualidade

Para avaliar o desempenho de diferentes algoritmos evolucionários multiobjetivo, diversas medidas de desempenho têm sido propostas. Estas medidas tentam quantificar o quão próximo o algoritmo está de satisfazer os dois objetivos principais da otimização multiobjetivo: descobrir as soluções da real fronteira de Pareto, e garantir que a soluções não dominadas encontradas tenham o máximo de diversidade possível. Neste trabalho, foram utilizadas duas medidas de qualidade para comparar as soluções encontradas pelos algoritmos propostos com as soluções encontradas pelas abordagens existentes na literatura (ZITZLER, 1999): (i) diferença de cobertura de dois conjuntos, e (ii) cobertura de dois conjuntos. Embora existam outras medidas na literatura, em geral, duas desvantagens são observadas: (i) para utilizá-las é necessário saber qual é o real conjunto de Pareto, que muitas vezes não é conhecido (como nos problemas adotados por este trabalho), e/ou (ii) elas requerem que os valores das funções objetivo tenham a escala ajustada, caso a magnitude dos valores para diferentes objetivos seja diferente (ZITZLER, 1999).

Definição 2.8 (Diferença de cobertura de dois conjuntos). Sejam A e B dois conjuntos de soluções não dominadas, então a diferença de cobertura de dois conjuntos, denotada por $\mathbf{D}(\mathbf{A},\mathbf{B})$, é o tamanho do espaço objetivo fracamente dominado por A e não dominado por B. A função $D(A,B)$ é definida da seguinte forma:

$$D(A,B) = S(A+B) - S(B),$$

onde a função $S(\beta)$, chamada de tamanho do espaço dominado (ZITZLER, 1999), mede quanto do espaço objetivo é dominado fracamente por um dado conjunto de soluções β.

Um exemplo de espaço dominado para um problema de minimização de dois objetivos é mostrado na Figura 2.16a. Os valores de referência para cada objetivo ($max1$ e $max2$, no exemplo) são os valores máximos que cada objetivo pode assumir. A Figura 2.16b mostra como a a função D é calculada. Nesse exemplo, $S(A+B) = W+U+V$, e $S(B) = W+V$, portanto,

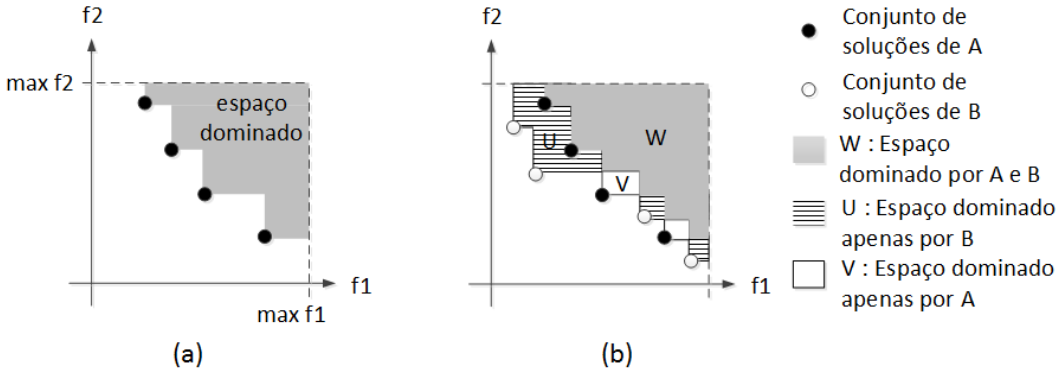


Figura 2.16: (a) Tamanho do espaço dominado para um conjunto de soluções quando o objetivo é minimizar dois objetivos. (b) Diferença de cobertura de dois conjuntos.

$D(A,B) = U$. Se $D(A,B) > D(B,A)$, então, de acordo com essa medida, o conjunto A é melhor que B.

Definição 2.9 (Cobertura de dois conjuntos). Sejam A e B dois conjuntos de soluções não dominadas, então a cobertura de dois conjuntos, denotada pela função $\mathbf{C}(\mathbf{A},\mathbf{B})$, é definida a seguir:

$$C(A,B) = \frac{|\{b \in B | \exists a \in A : a \preceq b\}|}{|B|}$$

Caso $C(A,B) = 1$, isto significa que todas as soluções em B são fracamente dominadas por A. O oposto, $C(A,B) = 0$, representa a situação em que nenhuma solução em B é fracamente dominada por A. Note que as duas direções precisam ser verificadas, uma vez que $C(A,B)$ não necessariamente significa $C(A,B) = 1 - C(B,A)$. Na Figura 2.16b, $C(A,B) = 0$ e $C(B,A) = 0,5$.

2.3 *Homogenous Synchronous Dataflow Graph*

Como será mostrado posteriormente, este trabalho adota um modelo *Homogenous Synchronous Dataflow Graph* (HSDG) (LEE; MESSERSCHMITT, 1987; GHAMARIAN et al., 2006) para explicitamente representar a concorrência que pode ser explorada na aplicação.

Definição 2.10 (HSDG). Um HSDG $G_{hsdg} = (A, C, s_0)$ é um grafo direcionado com um conjunto de vértices (também chamados de atores) A, e um conjunto de arcos C conectando os atores, tal que $C \subseteq A \times A$. $s_i : C \to \mathbb{N}$ é uma função de estado que associa a cada arco uma quantidade de dados (também chamados de *tokens*), e s_0 denota o estado inicial do HSDG.

Os atores representam atividades (ou computações) e os arcos representam filas FIFO (primeiro a entrar é o primeiro a sair) de capacidade infinita. Atores trocam *tokens* (dados) entre si por meio dos arcos. A Figura 2.17a apresenta um exemplo de HSDG. Atores, arcos e *tokens* são desenhados, respectivamente, como círculos, setas e pontos pretos nas setas. Este HSDG é denotado por G_{hsdg} = ($A = \{a_1, a_2, a_3, a_4, a_5, a_6\ a_7\}$, $C = \{(a_1, a_2), (a_1, a_3), (a_2, a_4), (a_3, a_5), (a_4, a_6), (a_5, a_6), (a_6, a_1), (a_6, a_7)\ \}$, $s_0 = \{(a_1, a_2, 0), (a_1, a_3, 0), (a_2, a_4, 0), (a_3, a_5, 0), (a_4, a_6, 0), (a_5, a_6, 0), (a_6, a_1, 1), (a_6, a_7, 0)\}$.

Em um HSDG, um ator está pronto para disparar (executar sua atividade) sempre que houver ao menos um *token* em cada um de seus arcos de entrada. Quando um ator dispara, ele remove um *token* de cada um de seus arcos de entrada, e adiciona um *token* a cada um de seus

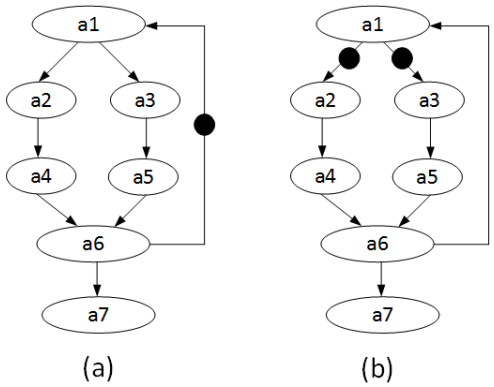


Figura 2.17: Exemplo de HSDG.

arcos de saída (LEE; MESSERSCHMITT, 1987). Na Figura 2.17a, apenas o ator a_1 está pronto para disparar, uma vez que apenas esse ator possui pelo menos um *token* em cada um de seus arcos de entrada. O disparo de a_1 remove um *token* do arco (a_6, a_1), e adiciona um *token* aos arcos (a_1, a_2) e (a_1, a_3) (Figura 2.17b). Após o disparo de a_1, apenas os atores a_2 e a_3 estão prontos para disparar. Comunicação e sincronização entre os atores só podem ocorrer pelos arcos (LEE; MESSERSCHMITT, 1987). Como o interesse deste trabalho é apenas representar a concorrência que pode ser explorada na aplicação, e não, por exemplo, analisar as propriedades funcionais da aplicação, nesta tese nós abstraímos os valores de fato representados pelos *tokens* (GEILEN; BASTEN; STUIJK, 2005). Em outras palavras, consideramos que os *tokens* são indistinguíveis.

Sejam $IC(a) = \{(a_w, a) \mid (a_w, a) \in C, \forall a_w \in A\}$ e $OC(a) = \{(a, a_z) \mid (a, a_z) \in C, \forall a_z \in A\}$, respectivamente, o conjunto de arcos de entrada e o conjunto de arcos de saída do ator $a \in A$. Então, o comportamento dinâmico do modelo (mudanças de estado) é determinado pelas definições a seguir.

Definição 2.11 (Atores prontos para disparar). Um ator está pronto para disparar sempre que houver ao menos um *token* em cada um de seus arcos de entrada. O conjunto de atores prontos para disparar no estado s_i é denotado pelo conjunto $EA(s_i) = \{a \in A \mid s_i(c) \geq 1, \forall c \in IC(a)\}$.

Definição 2.12 (Regra de disparo). Quando o ator $a \in EA(s_i)$ dispara, um *token* é removido de cada um de seus arcos de entrada $c_w \in IC(a)$, e um *token* é adicionado a cada um de seus arcos de saída $c_z \in OC(a)$. O novo estado s_j alcançado após disparo de a é dado pela equação abaixo.

$$s_j(c) = \begin{cases} s_i(c) - 1, & \text{se } c \in IC(a), \\ s_i(c) + 1, & \text{se } c \in OC(a), \\ s_i(c), & \text{caso contrário.} \end{cases}$$

Definição 2.13 (Execução). Uma execução σ de um HSDG $G_{hsdg} = (A, C, s_0)$ é uma sequência (finita ou infinita) de estados s_0, s_1, s_2, ... tal que s_{i+1} é o resultado do disparo de um ator a que está pronto para disparar em s_i ($a \in EA(s_i)$), para todo $i \geq 0$.

Definição 2.14 (Estados alcançáveis). Um estado s é alcançável no HSDG $G_{hsdg} = (A, C, s_0)$, se existe uma execução s_0, s_1, s_2, ... , $s_n = s$. $Reach(G_{hsdg})$ representa o conjunto de todos os estados alcançáveis em G_{hsdg}.

As regras de disparo acima implicam que mais de um ator pode estar pronto para disparar ao mesmo tempo e, portanto, o modelo consegue representar concorrência. É possível observar no HSDG G_{hsdg} da Figura 2.17 que, por exemplo, os atores a_4 e a_5 podem estar prontos para disparar ao mesmo tempo ($\exists s \in Reach(G_{hsdg}) \mid a_4, a_5 \in EA(s)$). Por outro lado, os atores a_3 e a_5 não podem estar prontos para disparar ao mesmo tempo ($\nexists s \in Reach(G_{hsdg}) \mid a_3, a_5 \in EA(s)$).

Definição 2.15 (Arco estritamente limitado). Considere o HSDG $G_{hsdg} = (A, C, s_0)$, então um arco $c \in C$ é dito estritamente limitado se, e somente se, $\exists B \mid s(c) \leq B, \forall s \in Reach(G_{hsdg})$.

Definição 2.16 (Capacidade máxima de um arco). A capacidade máxima de um arco $c \in C$ de um HSDG $G_{hsdg} = (A, C, s_0)$ é igual a K se $\exists s \in Reach(G_{hsdg}) \mid s(c) = K$ e, adicionalmente, $\forall s \in Reach(G_{hsdg}) \mid s(c) \leq K$.

Considere o HSDG $G_{hsdg} = (A, C, s_0)$ e um arco que não é estritamente limitado $c = (a_i, a_j) \in C$. Para transformar c em um arco estritamente limitado, com capacidade máxima K, basta adicionar ao HSDG G_{hsdg} um novo arco $c' = (a_j, a_i)$ cujo estado inicial é igual a n, tal que $K = n + s_0(c)$. Pelas regras de disparo, sempre depois de um disparo de a_i, um *token* precisa ser removido do arco $c' = (a_j, a_i)$ e um *token* precisa ser adicionado ao arco $c = (a_i, a_j)$. Da mesma forma, sempre que a_j disparar, um *token* precisa ser removido do arco $c = (a_i, a_j)$ e um *token* precisa ser adicionado ao arco $c' = (a_j, a_i)$. Desta maneira, o somatório da quantidade de *tokens* em c e c' será sempre igual ao somatório da quantidade de *tokens* nos estados iniciais de c e c'. Nesta tese, arcos cujo propósito é apenas limitar a capacidade máxima de *tokens* em outros arcos, são chamados de arcos limitantes.

Definição 2.17 (Predecessor, Sucessor). O conjunto de atores predecessores do ator $a_i \in A$ de um HSDG $G_{hsdg} = (A, C, s_0)$ é dado por $Prede(a_i) = \{a_j \in A | (a_j, a_i) \in C\}$. O conjunto de atores sucessores de a_i é dado por $Suce(a_i) = \{a_j \in A | (a_i, a_j) \in C\}$.

Definição 2.18 (Caminho não direcionado). Um caminho não direcionado p é uma sequência $a_1, a_2, a_3, \ldots, a_l$ tal que $a_{i+1} \in Prede(a_i) \cup Suce(a_i)$, para todo a_i, $0 \leq i < l$.

Definição 2.19 (Componente fracamente conectado). Um componente fracamente conectado de um HSDG $G_{hsdg} = (A, C, s_0)$ é um HSDG $G'_{hsdg} = (A', C', s'_0)$ que satisfaz as seguintes condições: (i) $A' \subseteq A$, $C' \subseteq C$, e $s'(c) = s(c), \forall c \in C'$, (ii) existe um caminho não direcionado de um ator $a_i \in A'$ para qualquer outro ator $a_j \in A'$, e (iii) não existe um outro HSDG $G^*_{hsdg} = (A^*, C^*, s^*_0)$ que satisfaça as duas condições anteriores e que satisfaça também uma das duas seguintes condições: $A' \subset A^*$ ou $C' \subset C^*$. A Figura 2.18 mostra um HSDG com dois componentes fracamente conectados.

2.4 *Parallel* DEVS

O formalismo DEVS (*Discrete Event System Specification*) (ZEIGLER; PRAEHOFER; KIM, 2000) foi desenvolvido por Bernard Zeigler para descrever os aspectos fundamentais dos sistemas de eventos discretos. Devido a sua capacidade de modelar sistemas complexos de maneira concisa e sem ambiguidades, o DEVS tem sido usado em diversas aplicações da engenharia (ex.: sistemas de manufatura, sistemas de comunicação, projeto de *hardware*) e ciência (ex.: biologia e ciência). Desde a formulação inicial de Zeigler, diversas variantes foram propostas para facilitar a modelagem e expandir as classes de sistemas que podem ser representadas pelo DEVS. Este trabalho adota a variante denominada P-DEVS (ZEIGLER;

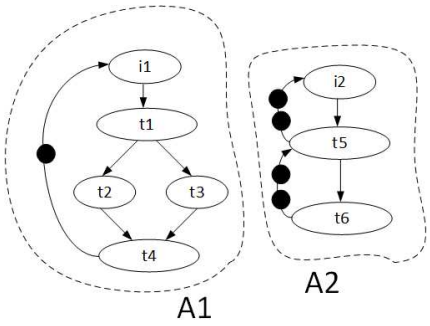


Figura 2.18: HSDG com dois componentes fracamente conectados: A1 e A2.

PRAEHOFER; KIM, 2000; CHOW; ZEIGLER, 1994), que foi desenvolvida para facilitar a descrição de sistemas paralelos e distribuídos. Apesar de existirem outros formalismos para descrever e simular sistemas de eventos discretos, como as redes de Petri (MOLLOY, 1982; LINDEMANN, 1998; MURATA, 1989), a escolha pela utilização do DEVS neste trabalho se deu principalmente: (i) pela disponibilidade de ferramentas de código aberto para esse formalismo (NUTARO, 2012), (ii) pela facilidade de integração dessas ferramentas com uma heurística de otimização, e (iii) pela simplicidade em se modelar os sistemas abordados no trabalho.

O formalismo P-DEVS consiste de duas partes, que são descritas nas subseções seguintes: modelo atômico e modelo acoplado.

2.4.1 Modelo atômico

O modelo atômico é dado pela seguinte tupla.

$$P-DEVS = (X, Y, S, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta), \tag{2.1}$$

onde

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ é um conjunto de eventos de entrada, $IPorts$ é um conjunto de portas de entrada e X_p é a faixa valores para a porta p;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ é o conjunto dos eventos de saída, $OPorts$ é o conjunto das portas de saída e Y_p é a faixa de valores para a porta p;

S é o conjunto dos estados parciais e $s_0 \in S$ é o estado parcial inicial;

$Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ é o conjunto dos estados e $ta : S \to \mathbb{R}^+ \cup \infty$ é a função de avanço do tempo;

$\delta_{int} : S \to S$ é a função de transição interna;

$\delta_{ext} : Q \times X^b \to S$ é a função de transição externa, X^b é o conjunto de todos os *bags*[3] sobre X, e $\delta_{ext}(s,e,\varnothing) = (s,e)$;

$\delta_{con} : Q \times X^b \to S$ é a função de transição confluente;

$\lambda : S \to Y^b$ é a função de saída.

A semântica de um modelo atômico é descrita seguir. A qualquer momento o sistema está no estado (s,e). Se nenhum evento externo chegar ao sistema, ele irá ficar no estado parcial s pelo tempo $ta(s)$, enquanto o tempo decorrido, e, irá acumular. Quando $e = ta(s)$ o sistema irá emitir um ou mais valores descritos por $\lambda(s)$, e irá mudar para o estado $(\delta_{int}(s), 0)$. Por outro lado, se um ou mais eventos externos, $x^b \in X^b$, chegarem enquanto o sistema estiver no estado (s,e) com $e < ta(s)$, o sistema então irá mudar para o estado $(\delta_{ext}(s,x^b,e),0)$. Uma ambiguidade

[3]Em matemática, um *bag* (ou *multiset*) é um conjunto em que múltiplas instâncias de um mesmo elemento podem ocorrer.

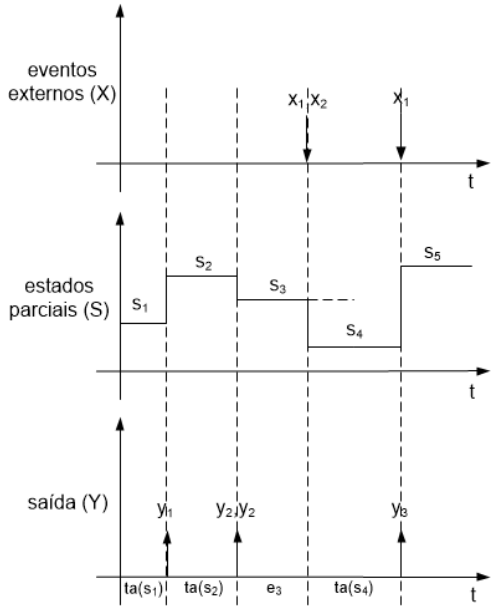


Figura 2.19: Transições de estado de um modelo P-DEVS atômico.

pode surgir na situação em que um ou mais eventos externos, x^b, chegam no mesmo momento em que o sistema estiver prestes a mudar para o estado parcial $\delta_{int}(s)$ (quando $e = ta(s)$). Essa ambiguidade é removida pela função δ_{con}, que faz com que o sistema mude para o estado $(\delta_{con}(s,x^b),0)$, quando a transição descrita anteriormente acontece. A Figura 2.19 exemplifica esses conceitos exibindo uma trajetória de um modelo atômico P-DEVS. O leitor deve notar que para esse modelo $s_2 = \delta_{int}(s_1)$, $s_3 = \delta_{int}(s_2)$, $s_4 = \delta_{ext}(s_3,\{x_1,x_2\},e_3)$, $s_5 = \delta_{con}(s_4,\{x_1\})$, $\lambda(s_1) = \{y_1\}$, $\lambda(s_2) = \{y_2,y_2\}$, e $\lambda(s_4) = \{y_3\}$.

Exemplo 2.2 (Lâmpada). *A Figura 2.20 exibe um exemplo de modelo atômico que representa o comportamento de uma lâmpada. O modelo não possui entradas e nem saídas (linha 2). A lâmpada possui dois estados: "ligada" e "desligada" (linha 3). Ela inicia no estado "ligada" (linha 4) e a cada 10 unidades de tempo (linhas 6 e 7) transiciona para o estado oposto ao estado atual (linhas 9-13). A Figura 2.21 exibe a trajetória desse modelo.*

```
1  M_lampada = (X, Y, S, s_0, δ_int, δ_ext, δ_con, λ, ta)
2  X = {}, Y = {}
3  S = {"ligada","desligada"}
4  s_0 = "ligada"
5
6  σ = ta(s):
7      σ := 10
8
9  s' = δ_int(s):
10     if s == "ligada" then
11         s' := "desligada"
12     else
13         s' := "ligada"
```

Figura 2.20: Exemplo de modelo atômico: lâmpada.

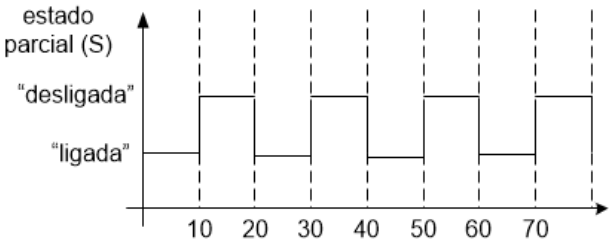


Figura 2.21: Trajetória do Exemplo 2.2 (lâmpada).

Exemplo 2.3 (**Acumulador**). *O modelo atômico na Figura 2.22 representa um acumulador. Esse modelo possui uma porta de entrada ("in"), que recebe os valores a serem acumulados, e uma porta de saída ("out"), que exibe o valor atual do acumulador (linhas 3 e 4). Como, para qualquer estado, a função de avanço do tempo (ta) é igual a 0 (linhas 7 e 8), a saída (linhas 10 e 11) assim como o estado atual (linhas 12 e 13) do acumulador são instantaneamente atualizados. A Figura 2.23 exibe uma trajetória desse modelo.*

```
1  M_acumulador = (X, Y, S, s_0, δ_int, δ_ext, δ_con, λ, ta)
2
3  X = {(p, v) | p ∈ IPorts, v ∈ X_p} tal que IPorts ∈ {"in" }, X_in ∈ ℝ
4  Y = {(p, v) | p ∈ OPorts, v ∈ X_p} tal que OPorts ∈ {"out" }, X_out ∈ ℝ
5  S = ℝ, s_0 = 0
6
7  σ = ta(s):
8      σ := 0
9
10 y^b = λ(s):
11     y^b := {("out", s)}
12 s' = δ_int(s):
13     s' := s
14
15 s' = δ_ext(s, e, x^b):
16     s' := s
17     foreach x in x^b do
18         s' := x.v + s'
```

Figura 2.22: Exemplo de modelo atômico: acumulador.

Exemplo 2.4 (**Processador com *buffer***). *Um processador com um* buffer *é apresentado na Figura 2.24. Ele tem duas portas: "in" e "out" (linhas 3 e 4). O estado do processador é representado por uma tupla (linha 5) que determina se o processador está ocioso ou não (idle), o tempo restante para terminar o* job *em execução (timeLeft) e uma fila com os* jobs *que estão esperando para serem executados (buffer). Quando o processador recebe um ou mais* jobs*, ele os armazena no* buffer *(linhas 8-11). Depois da inserção, caso o processador esteja ocioso, ele passa para o estado ocupado (idle = false) e inicia a execução do primeiro* job *da fila, caso contrário, ele continua a executar o* job *atual (linhas 12-16). Os* jobs *são identificados por um número natural e seus tempos de execução são determinados pela função RandomServiceTime(). Quando o processador termina a execução de um* job*, ele informa qual* job *foi finalizado por meio da porta "out" (linhas 18 e 19) e passa a executar o próximo* job *da fila (linhas 21-26).*

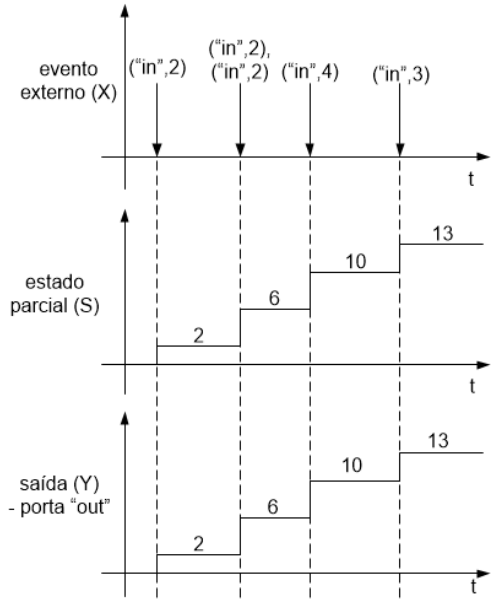


Figura 2.23: Trajetória do Exemplo 2.3 (acumulador).

Caso não exista nenhum job *na fila, o processador vai para o estado ocioso (linhas 27-29). A função* δ_{con} *(linhas 31 e 32) determina que na situação em que ao mesmo tempo um* job *chega e outro* job *termina, o processador deve finalizar a execução do* job *atual para só então receber o* job *que acabou de chegar. As funções* Pop *e* $Front$*, respectivamente, removem e retornam o primeiro* job *da fila.*

2.4.2 Modelo acoplado

Modelos atômicos podem ser acoplados entre si para formar o que é denominado de um modelo acoplado. O formalismo DEVS permite que modelos acoplados possam ser usados como modelos atômicos em modelos maiores, possibilitando, assim, a construção de modelos hierárquicos. A estrutura de um modelo DEVS acoplado é dada a seguir.

$$N = (X_{self}, Y_{self}, D, \{M_d\}, \{I_d\}, \{Z_{i,j}\}) \tag{2.2}$$

onde

X_{self} é o conjunto dos eventos de entrada;
Y_{self} é o conjunto dos eventos de saída;
D é o conjunto dos nomes dos componentes;
para cada $i \in D$,
  M_i é um modelo (componente) DEVS, atômico ou acoplado;
  I_i são os componentes influenciados por i;
para cada $i, j \in D \cup \{self\}$,
  $Z_{self,j} : X_{self} \to X_j, \forall j \in D$ é um acoplamento de entrada externa;
  $Z_{i,self} : Y_i \to X_{self}, \forall i \in D$ é um acoplamento de saída externa;
  $Z_{i,j} : Y_i \to X_j, \forall i, j \in D$ é um acoplamento interno i-para-j.

Exemplo 2.5 (**Pipeline**)**.** *Neste exemplo, um modelo acoplado é construído colocando três processadores em série para formar um* pipeline *(Figura 2.25). Os processadores são réplicas do processador definido no Exemplo 2.4. A porta "out" do primeiro processador é acoplada*

$M_{proc} = (X, Y, S, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ tal que $IPorts \in \{$"in" $\}$, $X_{in} \in \mathbb{N}$
$Y = \{(p,v) \mid p \in OPorts, v \in X_p\}$ tal que $OPorts \in \{$"out" $\}$, $X_{out} \in \mathbb{N}$
$S = \{idle \in \{$"$true$","$false$"$\}$, $timeLeft \in \mathbb{R}$, $buffer \in \mathbb{N}*\}$
$s_0 = (idle =$"$true$"$, timeLeft = +\infty, buffer = ())$

$\mathbf{s' = \delta_{ext}(s, e, x^b)}$:
    $s'.buffer = s.buffer$
    foreach x **in** x^b **do**
        $Insert(s'.buffer, x.v)$
    if $s.idle ==$"$true$" **then**
        $s'.timeLeft := RandomServiceTime()$
    else
        $s'.timeLeft := s'.timeLeft - e$
    $s'.idle :=$"$false$"

$\mathbf{y^b = \lambda(s)}$:
    $y^b := \{($"out"$, Front(s.buffer))\}$

$\mathbf{s' = \delta_{int}(s)}$:
    $s'.buffer = s.buffer$
    $Pop(s'.buffer)$
    if $SizeOf(s'.buffer) >= 1$ **then**
        $s'.timeLeft := RandomServiceTime()$
        $s'.idle :=$"$false$"
    else
        $s'.timeLeft := +\infty$
        $s'.idle :=$"$true$"

$\mathbf{s' = \delta_{con}(s, e, x^b)}$:
    $s' := \delta_{ext}(\delta_{int}(s), 0, x^b)$

$\mathbf{\sigma = ta(s)}$:
    $\sigma := s.timeLeft$

Figura 2.24: Exemplo de modelo atômico: processador com *buffer*.

a porta "in" do segundo, e o mesmo acontece para o segundo e terceiro processadores. Esse tipo de acoplamento é chamado de acoplamento interno. O pipeline *tem uma porta de entrada ("in"), que é acoplada a porta "in" do primeiro processador, e uma porta de saída ("out"), que é acoplada a porta "out" do terceiro processador. Esses dois tipos de acoplamento são chamados, respectivamente, de acoplamento de entrada externa e acoplamento de saída externa.*

A definição do *pipeline* é dada a seguir.

$$N_{pipeline} = (X_{self}, Y_{self}, D, \{M_d\}, \{I_d\}, \{Z_{i,j}\}),$$

onde

$X_{self} = \{($"in"$, \mathbb{N})\}$
$Y_{self} = \{($"out"$, \mathbb{N})\}$
$D = \{processador0, processador1, processador2\}$

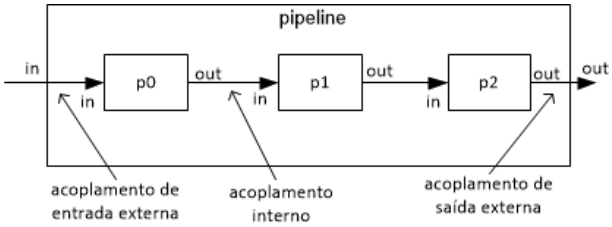


Figura 2.25: Exemplo de modelo acoplado (Pipeline - Exemplo 2.5).

$M_{processador0} = M_{processador1} = M_{processador2} = M_{proc}$

$I_{self} = \{processor0\}, I_{processor0} = \{processor1\}, I_{processor1} = \{processor2\}, I_{processor2} = \{self\}$

$Z_{self,processor0}(\text{“}in\text{”},x) = (\text{“}in\text{”},x), Z_{processor0,processor1}(\text{“}out\text{”},x) = (\text{“}in\text{”},x),$
$Z_{processor1,processor2}(\text{“}out\text{”},x) = (\text{“}in\text{”},x), Z_{processor2,self}(\text{“}out\text{”},x) = (\text{“}out\text{”},x), \forall x \in \mathbb{N}$

2.4.3 Clausura

Como mencionado anteriormente, modelos acoplados podem ser usados como modelos atômicos em modelos maiores. Isto é possível devido à propriedade denominada clausura (do inglês *closure*) (ZEIGLER; PRAEHOFER; KIM, 2000; CHOW; ZEIGLER, 1994), que faz com que seja possível transformar qualquer modelo acoplado em um modelo atômico equivalente. A propriedade da clausura pode ser demonstrada mostrando que o modelo acoplado $(X_{self}, Y_{self}, D, \{M_d\}, \{I_d\}, \{Z_{i,j}\})$, resultante do acoplamento dos modelos atômicos M_d, $d \in D$, é um modelo atômico P-DEVS $(X, Y, S, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$, onde

$X = X_{self}$

$S = \times Q_i, i \in D$

$Y = Y_{self}$

$ta(s) = minimum\{\sigma_i | i \in D\}$, $s \in S$, $\sigma_i = ta(s_i) - e_i$

Seja $s = (..., (s_i, e_i), ...)$, $IMM(s) = \{i | \sigma_i = ta(s)\}$, $INF(s) = \{j | j \in \bigcup_{i \in IMM(s)} I_i\}$, $CONF(s) = IMM(s) \cap INF(s)$, $INT(s) = IMM(s) - INF(s)$, $EXT(s) = INF(s) - IMM(s)$, onde $INT(s)$ contém os subcomponentes que estão prontos para realizar uma transição interna sem eventos de entrada, $EXT(s)$ contém os subcomponentes que vão receber eventos mas não possuem transição interna agendada, $CONF(s)$ contém os subcomponentes que, ao mesmo tempo, vão receber eventos e possuem transições internas agendadas. Então,

$\lambda(s) = \{z_{i,self(\lambda_i(s_i))} | i \in IMM(s) \wedge z_{i,self} \in Z\}$

$\delta_{int}(s) = (..., (s_i', e_i)', ...)$, onde

$(s_i', e_i') = (\delta_{inti}(s_i), 0)$, para $i \in INT(s)$,

$(s_i', e_i') = (\delta_{exti}(s_i, e_i + ta(s), x_i^b), 0)$, para $i \in EXT(s)$,

$(s_i', e_i') = (\delta_{coni}(s_i, x_i^b), 0)$, para $i \in CONF(s)$,

$(s_i', e_i') = (s_i, e_i + ta(s))$, caso contrário, e

$x_i^b = \{z_{o,i}(\sigma_o(s_o)) | o \in IMM(s) \wedge z_{o,i} \in Z\}$;

$\delta_{ext}(s, e, x^b) = (..., (s_i', e_i)', ...)$, onde

$(s_i', e_i') = (\delta_{exti}(s_i), e_i + e, x_i^b, 0)$, para $i \in \{i | i \in I_{self} \wedge z_{self,i} \in Z\}$,

$(s_i', e_i') = (s_i, e_i + e)$ caso contrário, e

$x_i^b = \{z_{self,i}(x) | x \in x^b\}$;

Seja $INF'(s) = \{j | j \in \bigcup_{i \in (IMM(s) \cup self)} I_i\}$, $CONF'(s) = IMM(s) \cap INF'(s)$, $INT'(s) = IMM(s) - INF'(s)$, $EXT'(s) = INF'(s) - IMM(s)$. Então,

$\delta_{con}(s, e, x^b) = (..., (s_i', e_i)', ...)$, onde

$(s'_i, e'_i) = (\delta_{inti}(s_i), 0)$, para $i \in INT'(s)$,
$(s'_i, e'_i) = (\delta_{exti}(s_i, e_i + ta(s), x_i^b), 0)$, para $i \in EXT'(s)$,
$(s'_i, e'_i) = (\delta_{coni}(s_i, x_i^b), 0)$, para $i \in CONF'(s)$,
$(s'_i, e'_i) = (s_i, e_i + ta(s))$, caso contrário, e
$x_i^b = \{z_{o,i}(\sigma_o(s_o)) | o \in IMM(s) \wedge z_{o,i} \in Z\} \cup \{z_{self,i}(x) | x \in x^b\}$.

2.5 Considerações finais

Esse capítulo apresentou conceitos fundamentais sobre: (i) sistemas embarcados com múltiplos processadores (Seção 2.1), (ii) algoritmos de otimização (Seção 2.2), (iii) HSDG (Seção 2.3) e, por último, (iv) o DEVS (Seção 2.4), que é um formalismo para especificação e avaliação de sistemas de eventos discretos. Neste trabalho, algoritmos genéticos multiobjetivo são propostos para explorar o espaço de projeto de sistemas embarcados multiprocessados. Para avaliar cada alternativa de projeto, um modelo formal de simulação DEVS é automaticamente construído e avaliado pelo método proposto.

3 Trabalhos correlatos

A literatura sobre exploração do espaço de projeto de sistemas embarcados é bastante vasta, e um dos motivos para isso é que a exploração pode ser feita considerando diferentes níveis de abstração (ex.: sistema, lógico e circuito), diferentes variáveis de decisão (ex.: mapeamento, escalonamento e tamanho das *caches*), e diferentes objetivos de projeto (ex.: tempo, custo e potência consumida). Em geral, o problema de exploração do espaço de projeto é um problema multiobjetivo (objetivos de projeto conflitantes), e para resolvê-lo é preciso lidar com duas questões complementares (GRIES, 2004; JIA et al., 2014; KEMPF; ASCHEID; LEUPERS, 2011): (i) como avaliar uma única alternativa de projeto (modelos), e (ii) como eficientemente explorar o espaço de projeto (algoritmos), dado que normalmente a exploração exaustiva é inviável.

A Figura 3.1 apresenta a classificação dos métodos de exploração proposta em (JIA et al., 2014) (outras classificações são apresentadas em (GRIES, 2004) e (KEMPF; ASCHEID; LEUPERS, 2011)). Essa figura ilustra o fato de que as restrições de projeto, assim como as decisões tomadas em níveis de abstração superiores restringem o espaço de opções de projeto nos níveis inferiores. Para cada nível de abstração, a figura mostra quais as variáveis de decisão normalmente exploradas, e como uma solução candidata é avaliada pelos métodos atuais. Por exemplo, a exploração do espaço de mapeamento pode compreender o particionamento da aplicação entre *hardware*/*software* e o escalonamento da aplicação. A exploração no domínio da arquitetura pode incluir a definição de quantos e que tipos de processadores serão usados, e a exploração da aplicação pode estabelecer de que forma extrair a concorrência da aplicação. Além disso, a exploração da microarquitetura normalmente se concentra na configuração dos parâmetros (ex.: tamanho das *caches*, política de arbitragem dos barramentos) de cada componente individualmente. Nesta tese, conforme a classificação de (JIA et al., 2014), o foco é a exploração do espaço de mapeamento e da arquitetura. Dessa forma, este capítulo se concentra em métodos que tratam da exploração nesses dois domínios.

O restante do capítulo está organizado da seguinte forma: a Seção 3.1 apresenta métodos de exploração de uma única alternativa de projeto. A Seção 3.2 aborda métodos para cobrir o espaço de soluções. A Seção 3.3 compara as abordagens existentes com o trabalho proposto. Por último, a Seção 3.4 conclui o capítulo.

3.1 Métodos para avaliar uma única alternativa de projeto

Na análise de uma alternativa de projeto, o nível de abstração dos modelos de avaliação está diretamente relacionado à exatidão dos resultados obtidos (GRIES, 2004). Em modelos de baixo nível, os resultados são, na maioria das vezes, mais exatos que em modelos de alto nível, uma vez que mais detalhes da arquitetura são incorporados nesse tipo de modelo. No entanto, a avaliação através de modelos de baixo nível é demorada e menos flexível, dificultando

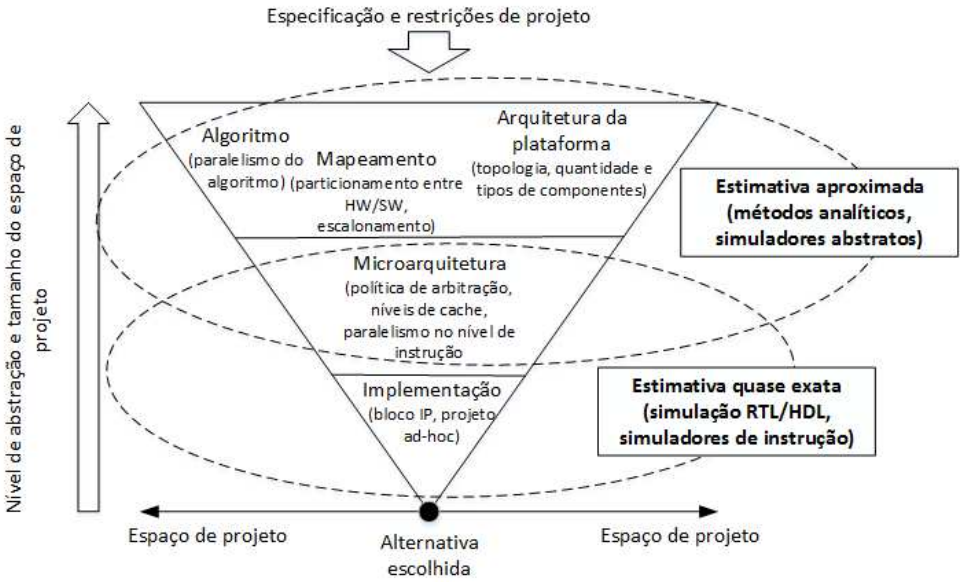


Figura 3.1: Classificação das abordagens de exploração do espaço de projeto (adaptado de (JIA et al., 2014)).

a exploração automática do espaço de projeto. Além disso, para se construir um modelo de baixo nível é preciso ter acesso a detalhes de implementação da arquitetura, que nem sempre estão disponíveis, devido a restrições de propriedade intelectual.

Nos últimos anos, diversos modelos têm sido propostos para representar arquiteturas e aplicações (GRIES, 2004; PIMENTEL et al., 2001; GERSTLAUER et al., 2009; TEICH, 2012; JIA et al., 2014). Conforme apresentado anteriormente, os modelos de arquitetura podem ser classificados em dois tipos (GRIES, 2004): (i) modelos abstratos, e (ii) modelos executáveis. Nos modelos abstratos, nenhuma especificação funcional do *hardware* é executada, e o tempo de execução é apenas simbolicamente representado (ex.: associando os tempos de duração às tarefas do sistema, sem, de fato, executar uma especificação do *hardware*). Ainda assim, quando recursos compartilhados precisam ser modelados, os modelos devem ser capazes de modelar a contenção desses recursos. Nos modelos executáveis (BENINI et al., 2005; AUSTIN; LARSON; ERNST, 2002; BROOKS; TIWARI; MARTONOSI, 2000; PEES et al., 1999), por outro lado, as estimativas são feitas com base na execução (simulação) de uma especificação funcional da arquitetura. Dessa forma, normalmente, modelos abstratos de arquitetura são mais rápidos de se avaliar, porém, modelos executáveis permitem modelar mais precisamente o comportamento que depende do estado da arquitetura como, por exemplo, os aspectos temporais das *caches* e *pipeline* (GRIES, 2004). O foco deste capítulo será em métodos que adotam modelos abstratos para a arquitetura, pois a grande maioria das abordagens para exploração automática do espaço de projeto adota este tipo de modelo (GERSTLAUER et al., 2009; GRIES, 2004; JIA et al., 2014).

Assim como os modelos utilizados para representar a arquitetura, a aplicação também pode ser representada em diferentes níveis de abstração (GRIES, 2004). Ela pode, por exemplo, ser representada de maneira abstrata ou executável. Em um modelo abstrato de aplicação, a carga de trabalho do sistema é definida sem que a especificação funcional da aplicação seja executada. Dessa forma, uma vantagem dos modelos executáveis de aplicação em relação aos modelos abstratos de aplicação é que eles permitem que as estimativas sejam feitas com base no real estado do fluxo de execução da aplicação. Por outro lado, o tempo de avaliação dos modelos executáveis, em muitos casos, é fortemente dependente da complexidade da especificação

funcional da aplicação.

Métodos que adotam modelos abstratos para a aplicação e modelos abstratos para a arquitetura: Estes métodos são muito utilizados para desenvolver sistemas embarcados de tempo-real críticos. O projeto de um sistema crítico deve garantir que o sistema funciona corretamente (sem violar *deadlines*), mesmo nos piores cenários. Dessa forma, métodos baseados em simulação não são muito atraentes para avaliar sistemas críticos, pois geralmente a simulação não consegue garantir isso (THIELE; PERATHONER, 2010). Real-time Calculus (THIELE; PERATHONER, 2010), SymTA/S (HENIA et al., 2005) e o *framework* MAST (GONZÁLEZ et al., 2001) são exemplos de métodos que recebem como entrada uma representação abstrata da aplicação e verificam se *deadlines* críticos do sistema podem ser violados.

A maneira mais comum de representar abstratamente a aplicação é considerar que: (i) os tempos de execução das tarefas são constantes e iguais ao WCET (ou outro valor constante, como o tempo médio de execução das tarefas), e/ou (ii) que os intervalos de ativação das tarefas (também chamados intervalos de chegada) são constantes (DICK, 2002; HA et al., 2007; PIMENTEL; ERBAS; POLSTRA, 2006; RUGGIERO et al., 2006; SCHMITZ; AL-HASHIMI; ELES, 2004; TAVARES et al., 2010; QUAN; PIMENTEL, 2014). Embora facilite a exploração do espaço de projeto, este modelo com tempos constantes não consegue representar adequadamente a variabilidade causada, por exemplo, pelos diferentes dados de entrada da aplicação e/ou pelos aspectos arquiteturais do *hardware* (ex.: *cache*, *pipeline*). Além disso, considerar que o tempo de execução de cada tarefa é igual ao seu WCET pode ser muito pessimista e levar a sistemas desnecessariamente caros. Assumir que os intervalos de ativação das tarefas são sempre constantes é outra grande limitação. Considere, por exemplo, uma tarefa que é ativada (fica pronta para executar) sempre que o usuário aperta algum botão da interface do sistema. Como os intervalos de tempo entre as interações do usuário com a interface são variáveis, trabalhos que assumem intervalos constantes de ativação não irão conseguir modelar adequadamente essa variabilidade. Dessa forma, para sistemas embarcados de tempo-real não críticos, distribuições de probabilidade são preferíveis para descrever os tempos de ativação e de execução das tarefas, uma vez que através dessas distribuições é possível estimar as probabilidades de violação de *deadlines*. O leitor deve notar que quando distribuições de probabilidade são adotadas, o pior caso também é coberto pela avaliação, ou seja, é um caso particular dentre cenários avaliados.

Dado um sistema cujos tempos de execução das tarefas são descritos por distribuições de probabilidades, o cálculo das probabilidades de violação de *deadlines* de maneira analítica (ou numérica) não é uma tarefa simples, e é um problema que tem atraído bastante atenção nos últimos anos. Apesar de algum progresso ter sido feito nessa área, os tipos de sistemas que podem ser avaliados pelos métodos existentes continuam muito restritos (KIM; LEE, 2009; ABENI; MANICA; PALOPOLI, 2012; MANOLACHE; ELES; PENG, 2002, 2004; MUPPALA; WOOLET; TRIVEDI, 1991; ZAMORA; HU; MARCULESCU, 2007). Por exemplo, a maioria dos métodos conseguem avaliar apenas arquiteturas com um único processador. Embora os métodos baseados em cadeias de Markov propostos em (MANOLACHE; ELES; PENG, 2002; MUPPALA; WOOLET; TRIVEDI, 1991; ZAMORA; HU; MARCULESCU, 2007) consigam avaliar arquiteturas com múltiplos processadores, eles só podem garantir resultados exatos se os tempos de execução e ativação das tarefas forem exponencialmente distribuídos. Quando estas suposições não são satisfeitas, as distribuições de probabilidade precisam ser aproximadas, por exemplo, por distribuições *Coxian* (MANOLACHE; ELES; PENG, 2002). Porém, o processo de aproximação por essas distribuições pode ser uma tarefa complexa de se automatizar (ZAMORA; HU; MARCULESCU, 2007; MANOLACHE; ELES; PENG, 2002).

Simulação estocástica é uma alternativa aos métodos analíticos descritos no parágrafo anterior, e é a abordagem adotada nesta tese e também pelos seguintes trabalhos (SONNTAG;

GRIES; SAUER, 2007; SATISH; RAVINDRAN; KEUTZER, 2008). Modelos de simulação permitem que as avaliações sejam feitas sem as restrições impostas pelos modelos analíticos (ex.: distribuições exponenciais). Apesar de normalmente a simulação estocástica ser mais lenta que alguns dos métodos analíticos, se corretamente aplicada (PAWLIKOWSKI; JEONG; LEE, 2002), ela pode ser usada para garantir um *feedback* relativamente rápido e confiável para o projetista.

Métodos que adotam modelos executáveis para a aplicação e modelos abstratos para a arquitetura: Nesses métodos, as operações da aplicação são anotadas com seus respectivos tempos de duração. Em seguida, a aplicação é simulada, e as estimativas são feitas a partir das operações executadas durante a simulação. Uma operação pode ser definida como sendo, por exemplo, uma instrução, conjunto de instruções, linha de código, etc.

Os métodos adotados pelas metodologias Deadalus (NIKOLOV et al., 2008; PIMENTEL; ERBAS; POLSTRA, 2006) e SystemCoDesigner (KEINERT et al., 2009), são exemplos proeminentes desta maneira de avaliar uma alternativa de projeto. Na metodologia Deadalus, KPNs (*Kahn Process Networks*) (GILLES, 1974) são usadas como modelo de computação. O comportamento funcional da aplicação é capturado por meio da instrumentação de cada processo Kahn com anotações que descrevem as ações do processo. Quando o modelo *Kahn* é executado, cada processo gera seu respectivo *trace*, que é usado para simular um modelo abstrato de arquitetura. Para simular os efeitos dos *traces*, os elementos da arquitetura são parametrizados com uma tabela com os tempos de execução de cada operação que pode ser gerada pela aplicação. A metodologia SystemCoDesigner utiliza uma abordagem semelhante, no entanto, a aplicação é especificada usando um subtipo da linguagem SystemC, chamada de SysteMoC (KEINERT et al., 2009).

Devido à lentidão em se avaliar (simular) modelos executáveis de aplicação, muitos métodos de exploração do espaço de projeto adotam uma estratégia de duas fases (PIMENTEL; ERBAS; POLSTRA, 2006; KANGAS et al., 2006; KEMPF; ASCHEID; LEUPERS, 2011; JIA et al., 2014; HERRERA; SANDER, 2015). Na primeira fase, um conjunto de soluções é selecionado usando exploração baseada em modelos abstratos e, na segunda fase, as soluções selecionadas pela fase anterior são avaliadas usando modelos executáveis. Porém, o problema dessa estratégia é que, devido às limitações de representação dos modelos abstratos utilizados, boas soluções podem ser erroneamente ignoradas na primeira fase.

3.2 Métodos de exploração das alternativas de projeto

Mesmo considerando que as tarefas do sistema possuem tempos constantes, mapeamento e escalonamento ótimos são ambos problemas intratáveis (GAREY; JOHNSON, 1979; LEUNG, 1989). Assim, a maioria dos algoritmos para cobrir o espaço de projeto são baseados em heurísticas (GAJSKI et al., 2009). Diversos algoritmos genéticos têm sido propostos (BLICKLE, 1997; DICK, 2002; SCHMITZ; AL-HASHIMI; ELES, 2004; THIELE et al., 2002; PURNAPRAJNA; REFORMAT; PEDRYCZ, 2007). Porém, no melhor do nosso conhecimento, nenhum deles focou em sistemas embarcados em que os tempos de execução e ativação das tarefas são descritos por distribuições de probabilidade.

Dick e Jha (DICK, 2002) propuseram o algoritmo genético multiobjetivo EMOGAC, que ataca um problema muito similar ao desta tese. No entanto, EMOGAC foca em sistemas embarcados de tempo-real críticos, e por isso os tempos de execução e ativação das tarefas são considerados constantes e baseados nos piores cenários de execução do sistema. O objetivo de EMOGAC é encontrar uma alocação, mapeamento e escalonamento que minimize custo e potência consumida sem que nenhum *deadline* crítico seja violado. Esse algoritmo também permite a definição de *deadlines* não críticos: nesse caso, além das outras medidas de projeto, o

algoritmo tenta minimizar o somatório da quantidade de tempo em que o sistema ultrapassa os *deadlines* não críticos.

Em (MANOLACHE; ELES; PENG, 2008), Manolache et al. propuseram um algoritmo baseado na meta-heurística busca tabu para minimizar violações de *deadlines*. No algoritmo proposto, os tempos de execução das tarefas são modelados por distribuições de probabilidades, porém, os tempos de ativação são considerados constantes. Ele ataca o problema de mapeamento e atribuição de prioridade a tarefas em arquiteturas com múltiplos processadores. Esse algoritmo considera que a arquitetura é fixa e por isso não trata do problema de alocação. Uma abordagem mono-objetivo é adotada cuja função objetivo é modelada como o somatório das probabilidades de se violar cada *deadline*. A aplicação é modelada como um conjunto de grafos de tarefas e um método analítico aproximado é proposto para avaliar as probabilidades de violação de *deadlines*. No entanto, cada tarefa só pode ter uma instância ativa. Esta restrição limita a concorrência que pode ser explorada durante a execução da aplicação.

Em (SATISH; RAVINDRAN; KEUTZER, 2008), um método baseado em simulação estocástica foi proposto para tentar otimizar o escalonamento de arquiteturas com processadores heterogêneos. Dado um conjunto de grafos de tarefas cujos tempos de execução são descritos por distribuições de probabilidade, o objetivo do método é encontrar um escalonamento que minimize o *makespan* (tempo de término de execução de todas as tarefas). O método não permite a representação de interrupções. Esta restrição limita a capacidade de modelar aplicações que possuem comportamento dependente do tempo (ex.: tarefas que precisam ser executadas periodicamente). É importante ressaltar também que minimizar o *makespan* nem sempre implica que as violações de *deadlines* também irão diminuir.

Na abordagem proposta por Saraswat et al. (SARASWAT; POP; MADSEN, 2010), os tempos de execução das tarefas que possuem *deadlines* não críticos são modelados por distribuições de probabilidade, ao passo que os tempos das tarefas com *deadlines* críticos são descritos pelo WCET. O objetivo do algoritmo proposto, que é baseado na meta-heurística busca tabu, é minimizar as probabilidades das tarefas não críticas violarem *deadlines* e garantir que nenhum *deadline* das tarefas críticas seja violado. Tarefas críticas são escalonadas usando a política EDF (*Earliest Deadline First*) e as não críticas com a política CBS (*Constant Bandwidth Server*). Diferentemente dos outros trabalhos apresentados nesta seção, nessa abordagem uma aplicação é modelada como um conjunto independente de tarefas, o que é uma grande desvantagem, pois isso impede que relações de precedência entre tarefas sejam especificadas.

Nos algoritmos propostos em (HUA; QU; BHATTACHARYYA, 2007; QIU et al., 2007; YALDIZ; DEMIR; TASIRAN, 2008), o objetivo é aproveitar a flexibilidade das restrições temporais dos sistemas de tempo-real não críticos para minimizar o consumo de energia utilizando DVS (*Dynamic Voltage and Frequency Scaling*). Dado que a potência consumida é proporcional ao quadrado da tensão de alimentação, a técnica DVS reduz o consumo de energia ajustando dinamicamente a frequência do *clock* e a tensão de alimentação do sistema (HUA; QU; BHATTACHARYYA, 2007). Esses algoritmos recebem como entrada um conjunto de grafos de tarefas, as distribuições de probabilidade dos tempos de execução de cada tarefa, e o percentual máximo que cada *deadline* pode ser violado, e retornam como saída uma política de atribuição de tensão de alimentação para os processadores.

A metodologia de desenvolvimento SCE (DÖMER et al., 2008; GAJSKI et al., 2009) adota uma abordagem baseada no paradigma especificar-explorar-refinar. O processo inicia com um modelo de especificação das funcionalidades (passo especificar). Em seguida, o projetista analisa as opções de projeto possíveis (passo explorar), e toma as decisões necessárias (ex.: decidir onde cada tarefa será mapeada). Depois, o método proposto automaticamente gera um novo modelo de mais baixo nível integrando as decisões que foram tomadas (passo refinar),

e volta para o passo explorar. Dessa forma, a abstração do modelo diminui a cada decisão tomada pelo projetista, uma vez que mais informações estão presentes no modelo. A grande limitação desta metodologia é que a exploração do espaço de opções de projeto precisa ser feita manualmente.

Na metodologia SystemCoDesigner (KEINERT et al., 2009; TEICH, 2012), um simulador é integrado a um algoritmo de exploração. Para cada solução gerada pelo algoritmo, um modelo em que a aplicação é representada de maneira executável é avaliado. O processo de avaliação (simulação) desta metodologia foi apresentado na seção anterior. Como a metodologia se baseia em modelos executáveis, os tempos de simulação são fortemente dependentes da complexidade da aplicação. Para explorar 7600 arquiteturas candidatas para uma aplicação M-JPEG, o algoritmo proposto levou aproximadamente três dias em um processador Intel Core 2 2400 MHz com 3 GB de RAM (KEINERT et al., 2009).

Na metodologia Deadalus (NIKOLOV et al., 2008; PIMENTEL; ERBAS; POLSTRA, 2006), a exploração do espaço de projeto é feita em duas fases. Inicialmente um problema multiobjetivo é formulado e resolvido usando o algoritmo SPEA2 (ZITZLER; LAUMANNS; THIELE, 2001). O problema é estabelecido considerando tempos constantes para as tarefas. Em seguida, o projetista deve selecionar algumas das soluções encontradas pelo algoritmo SPEA2 para avaliá-las com maior exatidão usando simulação. O método de simulação adotado por Deadalus foi descrito na seção anterior. Uma grande desvantagem da metodologia proposta é que o modelo de especificação adotado para as aplicações não permite a representação de interrupções (PIMENTEL; ERBAS; POLSTRA, 2006). Em (JIA et al., 2014), uma abordagem de duas fases muito similar a adotada por Deadalus é apresentada.

Embora SystemCoDesigner, Deadalus e o método apresentado em (JIA et al., 2014) sejam abordagens promissoras, elas são mais adequadas para o desenvolvimento de sistemas que executam uma única aplicação, e não um conjunto de aplicações. Isto acontece porque o processo de exploração do espaço de projeto dessas metodologias não considera a exploração do escalonamento das tarefas do sistema, apenas alocação e mapeamento são considerados.

Em (HERRERA; SANDER, 2015), outra abordagem de duas fases é apresentada. Na primeira fase um problema de satisfação de restrições é formulado e resolvido para eliminar as soluções que violam restrições temporais críticas (assume-se que as tarefas possuem tempos constantes). Em seguida, para cada solução que não viola restrições críticas, um modelo em que a aplicação é representada de maneira executável é construído e avaliado. O objetivo dessa última fase é tentar otimizar as restrições temporais não críticas, que não são consideradas na primeira fase. A grande limitação de (HERRERA; SANDER, 2015) é que interrupções não podem ser representadas no modelo de aplicação proposto.

3.3 Análise comparativa dos trabalhos

A partir da descrição dos trabalhos correlatos apresentados neste capítulo, é possível notar que as abordagens de exploração para sistemas embarcados podem ser classificadas pela maneira como uma alternativa de projeto é avaliada: (i) simulação, (ii) métodos analíticos (ou numéricos), e (iii) uma combinação dos dois. O grande problema dos métodos que adotam simulação é que eles podem demandar longos tempos de avaliação. Por outro lado, o problema dos métodos analíticos é que ou eles são baseados em suposições pouco realistas (ex.: tarefas com tempos constantes) ou consideram especificações muito simplificadas (ex.: arquiteturas com apenas um processador). Nos métodos que combinam as duas abordagens, inicialmente métodos analíticos são adotados para rapidamente eliminar soluções de qualidade inferior e, em seguida, métodos baseados em simulação são usados para analisar com maior exatidão as

soluções não eliminadas no passo anterior. Porém, o principal problema dessa estratégia é que, devido às limitações de representação dos modelos abstratos utilizados, boas soluções podem ser erroneamente ignoradas no primeiro passo.

No trabalho proposto por esta tese, um novo método de exploração que combina as melhores características da simulação (representatividade) e dos métodos analíticos (rapidez) é apresentado. Neste trabalho, adotamos uma abordagem centrada em simulação estocástica (BOLCH et al., 2006), na qual os modelos de simulação propostos são livres de detalhes funcionais da arquitetura/aplicação, o que evita os longos tempos de simulação dos métodos baseados em modelos executáveis. Além disso, o método proposto adota técnicas do estado da arte para analisar estatísticamente as saídas das simulações (EWING; PAWLIKOWSKI; MCNICKLE, 1999), e, assim, garantir resultados confiáveis. Embora os modelos de simulação propostos sejam mais lentos para serem avaliados que alguns dos modelos analíticos da literatura, eles não possuem as limitações de representatividade dos modelos analíticos (ex.: tempos de execução com distribuição exponencial).

O trabalho proposto possui algumas semelhanças com os trabalhos de Sonntag et al. (SONNTAG; GRIES; SAUER, 2007) e Satish et al. (SATISH; RAVINDRAN; KEUTZER, 2008), dado que eles também usam simulação estocástica. No entanto, (SONNTAG; GRIES; SAUER, 2007) não inclui nenhum mecanismo para exploração automática do espaço de projeto. O trabalho de (SATISH; RAVINDRAN; KEUTZER, 2008), como já mencionado, não permite que interrupções sejam modeladas. Além disso, (SATISH; RAVINDRAN; KEUTZER, 2008) objetiva otimizar apenas o *makespan*, o que não necessariamente significa que as violações de *deadlines* também serão otimizadas. Por último, como os modelos de simulação apresentados em (SONNTAG; GRIES; SAUER, 2007) e (SATISH; RAVINDRAN; KEUTZER, 2008) não foram comparados a medições em uma arquitetura real (ou um simulador de baixo nível), fica difícil determinar a exatidão desses modelos.

A Tabela 3.1 apresenta um resumo da comparação entre o trabalho proposto e outros trabalhos representativos que abordam o problema de exploração automática do espaço de projeto. A primeira coluna apresenta o trabalho considerado na comparação, a segunda coluna descreve se o método considera a variabilidade no tempo de execução da aplicação, a terceira coluna indica se o método permite modelar interrupções, a quarta coluna indica se o método considera a otimização do escalonamento das tarefas da aplicação, a quinta coluna mostra se o método modela a aplicação de maneira abstrata, executável ou se uma estratégia de duas fases é adotada, a sexta coluna indica se o método de exploração adotado é multiobjetivo, a sétima coluna mostra como os modelos são avaliados, e a última coluna apresenta se os resultados apresentados pelos modelos propostos foram comparados aos resultados produzidos por um sistema real.

	Tarefas com tempos variáveis	Interrupções	Escalonamento	Modelo de aplicação	Multi-objetivo	Avaliação	Validação
Metodo proposto	S	S	S	Abstrato	S	Simulação estocástica	S
(DICK, 2002)	N	S [3]	S	Abstrato	S	Analítica	N
(MANOLACHE; ELES; PENG, 2008)	S	S [3]	S	Abstrato	N	Analítica	N
(SATISH; RAVINDRAN; KEUTZER, 2008)	S	N	S	Abstrato	S	Simulação estocástica	N
(SARASWAT; POP; MADSEN, 2010)	S	S [3]	S	Abstrato	N	Analítica	N
(YALDIZ; DEMIR; TASIRAN, 2008)	S	S	S	Abstrato	N	Analítica	N
(NIKOLOV et al., 2008)[1]	S[2]	N	N	Duas fases	S	Analítica/Simulação determinística	S
(KEINERT et al., 2009)	S	S	N	Executável	S	Simulação determinística	S
(JIA et al., 2014)[1]	S[2]	N	N	Duas fases	S	Analítica/Simulação determinística	N
(HERRERA; SANDER, 2015)[1]	S[2]	N	S	Duas fases	S	Analítica/Simulação determinística	N

[1] O método possui duas fases de exploração: a primeira fase adota métodos analíticos e a segunda fase adota simulação.

[2] O método considera tempos variáveis apenas em sua segunda fase de exploração.

[3] O método permite modelar interrupções. No entanto, diferentemente do trabalho proposto, o método considera que os intervalos entre as interrupções são sempre constantes.

Tabela 3.1: Comparação com os métodos da literatura.

3.4 Considerações finais

Inicialmente, este capítulo apresentou uma revisão dos métodos existentes na literatura para avaliação de uma única alternativa de projeto (Seção 3.1). Em seguida, métodos (algoritmos e metodologias) para cobrir o espaço de projeto foram descritos (Seção 3.2). Por fim, uma comparação entre os trabalhos da literatura e o trabalho proposto foi apresentada (Seção 3.3). Este trabalho propõe um novo método de exploração do espaço de projeto de sistemas embarcados de tempo-real não críticos. Como apresentado, o método se baseia em simulação estocástica para evitar os problemas relacionados ao tempo de avaliação e representatividade dos modelos adotados pelos métodos atuais.

4
Especificação e caracterização

Esse capítulo detalha as duas primeiras atividades do método proposto: especificação e caracterização. A atividade de especificação consiste na definição da aplicação e plataforma de *hardware* utilizando os modelos de especificação propostos. Na atividade de caracterização, o projetista deve definir informações sobre o tempo de execução das tarefas da aplicação, assim como informações sobre o custo monetário e a potência consumida dos elementos de *hardware*. Conforme é apresentado na Figura 4.1, durante o processo de exploração do espaço de projeto, as informações geradas nas atividades de especificação e caracterização serão usadas para: (i) identificar quais são as possíveis alternativas de projeto (espaço de soluções candidatas), e (ii) avaliar a qualidade de uma dada alternativa de projeto.

Como será mostrado neste capítulo, este trabalho propõe modelos abstratos para especificar a aplicação e a plataforma de *hardware*. Porém, diferentemente de grande parte dos trabalhos que também adotam modelos abstratos, os modelos propostos capturam a variabilidade existente no tempo de execução dos sistemas embarcados. O capítulo também apresenta o ambiente proposto para dar suporte às atividades de especificação e caracterização. Este ambiente é composto por: (i) uma biblioteca para dar suporte ao desenvolvimento de aplicações que executam em arquiteturas compostas por processadores heterogêneos e programáveis, e (ii) um conjunto de métodos para capturar informações sobre tempo de execução da aplicação e potência consumida da arquitetura.

O restante do capítulo está organizado da seguinte forma: a próxima seção introduz os modelos propostos de aplicação e plataforma de *hardware*. A Seção 4.2 apresenta a biblioteca desenvolvida para dar suporte ao modelo de aplicação. A Seção 4.3 detalha quais informações devem ser determinadas pelo projetista durante a atividade de caracterização, e os métodos propostos para obtê-las. Por fim, a Seção 4.4 conclui o capítulo.

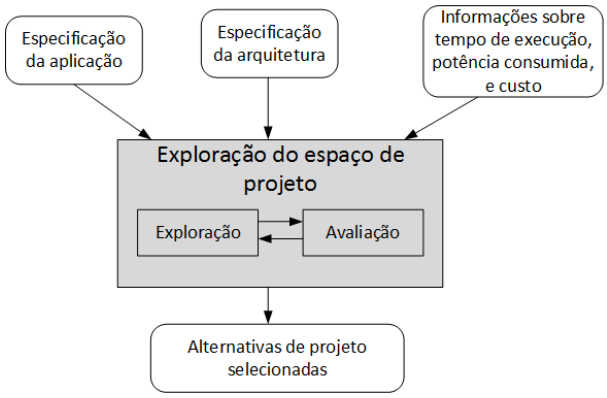


Figura 4.1: Exploração do espaço de projeto.

4.1 Especificação

As próximas subseções descrevem os modelos propostos para especificar a aplicação e a plataforma de *hardware* (NOGUEIRA et al., 2013).

4.1.1 Especificação da aplicação

A aplicação é especificada através de um *Homogenous Synchronous Dataflow Graph* - HSDG $G_{hsdg} = (A, C, s_0)$ (recordar na Seção 2.3 a definição de um HSDG), onde:

- A é o conjunto dos atores. Neste trabalho, o conjunto A é composto por dois conjuntos disjuntos: T e I, tal que $T \cup I = A$:
  - T é o conjunto dos atores que representam as tarefas (ou processos) que executam concorrentemente no sistema.
  - I é o conjunto dos atores que representam os processos do ambiente externo. Mais especificamente, estes atores modelam a interface da aplicação com o mundo externo. Quando um ator deste conjunto dispara, isto indica que uma interrupção foi gerada. Estes atores podem ser usados para modelar, por exemplo, interrupções geradas quando um pacote de dados da internet é recebido pela interface de rede do sistema. Cada componente fracamente conectado do HSDG (recordar a definição de componente fracamente conectado na Definição 2.19) deve ter um único ator $i \in I$ em seu conjunto de atores.
- C é o conjunto dos arcos. O conjunto C neste trabalho é composto por três conjuntos disjuntos: C_{int}, C_{lim} e C_d, tal que $C = C_{int} \cup C_{lim} \cup C_d$:
  - C_d é o conjunto dos arcos que representam canais de comunicação entre as tarefas da aplicação. Um canal de comunicação é uma fila FIFO armazenada na memória do sistema.
  - $C_{int} = \{(i,t) \in C \mid i \in I,\ t \in T\}$ é o conjunto dos arcos que representam a comunicação entre os processos do ambiente externo e as tarefas da aplicação.
  - C_{lim} é o conjunto dos arcos limitantes. Em um modelo HSDG, a capacidade máxima de um arco é ilimitada. No entanto, isso é inviável em uma implementação real, pois implicaria em uma memória com capacidade infinita. A Seção 2.3 mostrou que qualquer arco de um HSDG pode ter sua capacidade máxima limitada adicionando novos arcos, chamados de arcos limitantes. Arcos limitantes não representam de fato comunicação de dados. O objetivo deles é apenas limitar capacidade máxima de outros arcos.
- $s_0 : C \to \mathbb{N}$ é a função que determina a quantidade inicial de *tokens* em cada arco. *Tokens* representam mensagens entre tarefas ou interrupções geradas por processos do ambiente externo.

A Figura 4.2a apresenta um exemplo de especificação de aplicação, denotada por $G_{hsdg} = (A = T \cup I,\ C = C_d \cup C_{int} \cup C_{lim}, s_0 = \{\ (i_1,t_1,0), (t_1,i_1,1), (t_1,t_2,0), (t_2,t_1,1), (t_1,t_3,0),$

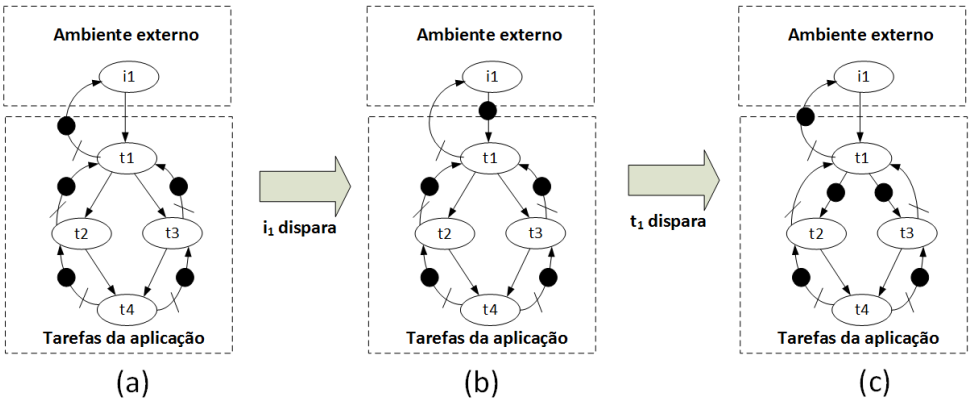


Figura 4.2: Exemplo de especificação da aplicação.

$(t_3,t_1,1)$, $(t_2,t_4,0)$, $(t_4,t_2,1)$, $(t_3,t_4,0)$, $(t_4,t_3,1)$, $\}$), onde $T = \{t_1,t_2,t_3,t_4\}$, $I = \{i_1\}$, $C_d = \{$ (t_1,t_2), (t_1,t_3), (t_2,t_4), (t_3,t_4) $\}$, $C_{int} = \{$ (i_1,t_1) $\}$ e $C_{lim} = \{$ (t_2,t_1), (t_3,t_1), (t_4,t_2), (t_4,t_3), (i_1,t_1) $\}$. Atores, arcos e *tokens* são desenhados, respectivamente, como círculos, setas e pontos pretos nas setas. Arcos limitantes (conjunto C_{lim}) são desenhados como uma seta com um traço. No estado inicial s_0, apenas o ator (processo do ambiente externo) i_1 está pronto para disparar, uma vez que apenas este ator possui um *token* em cada um de seus arcos de entrada (recordar a regra de disparo de um HSDG na Definição 2.12). O disparo deste ator remove um *token* do arco (t_1,i_1) e adiciona um *token* ao arco (i_1,t_1) (Figura 4.2b). No estado alcançado após o disparo de i_1, apenas o ator (tarefa) t_1 está pronto para disparar. O disparo de t_1 remove um *token* dos arcos (i_1,t_1), (t_2,t_1) e (t_3,t_1), e adiciona um *token* aos arcos (t_1,t_2) e (t_1,t_3). Neste momento, i_1, t_2 e t_3 estão prontos para disparar.

Nós escolhemos utilizar HSDGs como modelo de computação, principalmente, porque eles possuem as seguintes características: (i) esses modelos conseguem representar adequadamente aplicações concorrentes e focadas em dados (*data-oriented*), e (ii) como as comunicações entre atores (tarefas) só podem ocorrer pelos arcos (canais), problemas como inversão de prioridades e *race conditions* (TANENBAUM; BOS, 2014) são evitados, o que facilita o escalonamento.

Especificação funcional: A Figura 4.3a apresenta o esquema geral de como este trabalho considera que a especificação funcional (código executável) de uma tarefa está estruturada. Nesta figura, apresentamos o pseudocódigo de uma tarefa genérica, chamada de *Tarefa*, que recebe mensagens de k tarefas diferentes utilizando k canais de entrada (*in_chan_1*, *in_chan_2*, ... , *in_chan_k*), e envia mensagens para l tarefas diferentes através de l canais de saída (*out_chan_1*, *out_chan_2*, ... , *out_chan_l*). A tarefa executa repetidamente o seguinte comportamento:

1. Primeiro, ela verifica se existem mensagens em cada um de seus k canais de entrada (linhas 4-7 da Figura 4.3a). Caso o canal de entrada verificado tenha uma mensagem, a tarefa recebe o endereço da primeira mensagem na fila do canal correspondente. Caso contrário, a tarefa é bloqueada, isto é, a tarefa tem seu fluxo de execução suspenso pelo sistema operacional. Ela só será desbloqueada quando uma mensagem for inserida no canal vazio. O leitor deve notar que este comportamento é o mesmo apresentado por um ator em um HSDG, no sentido de que o ator só pode executar (disparar) caso haja ao menos um *token* em cada um de seus arcos de entrada.

2. Depois de verificar a existência de mensagens, a tarefa tenta obter espaço de memória para escrever uma mensagem em cada um de seus l canais de saída (linhas 8-11). Caso a tarefa tente obter espaço em um canal que está completamente cheio, a tarefa

é bloqueada. Neste caso, ela só será desbloqueada quando houver espaço suficiente para escrever uma nova mensagem no canal que a fez ser bloqueada. Em um modelo HSDG, este comportamento é representado pelos arcos limitantes (ex.: arco (t_2, t_1) da Figura 4.2).

3. Logo após obter espaço para escrever suas mensagens de saída, a tarefa copia as mensagens recebidas para uma memória local (linha 12), para que ela possa trabalhar localmente (mais rapidamente) com as mensagens recebidas.

4. Depois, a tarefa executa suas atividades (linha 13), e escreve as mensagens de saída nos canais de saída (linha 14).

5. Em seguida, ela remove as mensagens lidas dos canais de entrada (linhas 15-18), liberando espaço para que outras mensagens possam ser escritas nesses canais.

6. Finalmente, a tarefa notifica cada uma das l tarefas que receberão as mensagens de saída que novas mensagens estão disponíveis (linhas 19-22). O comportamento apresentado nas linhas 15-22 reflete o comportamento de um ator após o disparo, isto é, depois que um ator dispara, ele remove um *token* de cada um de seus arcos de entrada, e adiciona um *token* a cada um de seus arcos de saída.

Assumindo que a capacidade máxima dos canais *out_chan_1*, ... , *out_chan_l*, *in_chan_1*, ... , *in_chan_k* é igual a dois, a Figura 4.3b mostra como esta especificação funcional poderia ser mapeada em um HSDG: (i) o ator t modela a tarefa, (ii) os arcos (t_{11}, t), (t_{12}, t), ... , (t_{1k}, t) modelam os canais *in_chan_1*, *in_chan_2*, ... , *in_chan_k*, (iii) os arcos (t_{21}, t), (t_{22}, t), ... , (t_{2l}, t) modelam os canais *out_chan_1*, *out_chan_2*, ... , *out_chan_l*, (iv) os arcos limitantes (t, t_{11}), (t, t_{12}), ... , (t, t_{1k}), (t, t_{21}), (t, t_{22}), ... , (t, t_{2l}) modelam a capacidade máxima dos canais, (v) os atores t_{11}, t_{12}, ... , e t_{1k} representam as k diferentes tarefas que enviam mensagens para a tarefa, e (vi) os atores t_{21}, t_{22}, ... , e t_{2l} modelam as l diferentes tarefas que recebem as mensagens enviadas pela tarefa.

A Figura 4.4 mostra como este trabalho considera que as rotinas de tratamento de interrupções (ISR - *interrupt service routine* (WOLF, 2012)) do sistema estão estruturadas. Uma rotina de tratamento de interrupções é uma rotina que é chamada sempre que uma interrupção que está habilitada a ocorrer é recebida pelo sistema. Cada interrupção deve ter sua própria rotina de tratamento específica. No código da Figura 4.4a, quando uma interrupção é recebida (ex.: o usuário do sistema apertou um botão da interface), a rotina de tratamento correspondente utiliza os canais *out_chan1*, *out_chan2*, ... , *out_chank* para avisar as k tarefas que estão esperando a interrupção (linhas 2-5). A Figura 4.4b mostra como esta especificação funcional poderia ser mapeada em um HSDG, em que: (i) o disparo do ator $i \in I$ modela a execução da rotina de interrupção, (ii) os arcos (i, t_{11}), (i, t_{12}), ... , (i, t_{1k}) modelam os canais entre a rotina de tratamento de interrupção e as tarefas esperando pela interrupção, e (iii) t_{11}, t_{12}, ... , $t_{1k} \in T$ representam as tarefas esperando pela interrupção. A Figura 4.4c mostra um outro estilo de implementação, no qual, sempre que uma interrupção é gerada, a rotina de tratamento de interrupções desabilita a interrupção e, dessa forma, uma nova interrupção só poderá ser gerada depois que a interrupção for reabilitada. O modelo da Figura 4.4d mostra como esta especificação poderia ser representada. De acordo com este modelo, a interrupção é sempre desabilitada pela rotina de tratamento de interrupções, e apenas depois que a tarefa representada pelo ator t_{11} executa é que ela é reabilitada. Neste trabalho, assumimos que toda a especificação funcional da aplicação deve estar implementada apenas nas tarefas, isto é, as rotinas de tratamento devem ser usadas apenas para avisar as tarefas sobre eventos do mundo externo.

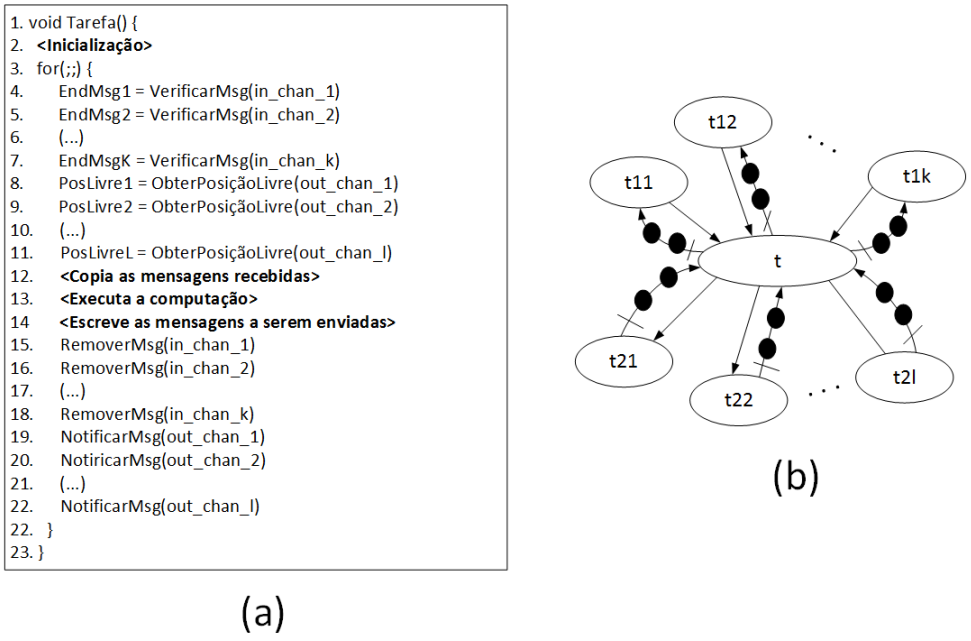


Figura 4.3: Esquema conceitual de como o código funcional de uma tarefa está estruturado.

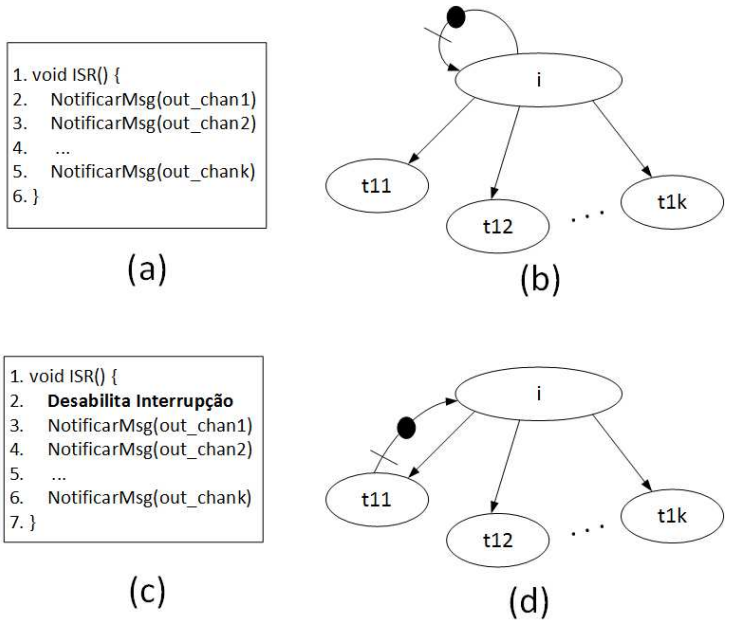


Figura 4.4: Esquema conceitual de como o código funcional de uma rotina de tratamento de interrupções está estruturado.

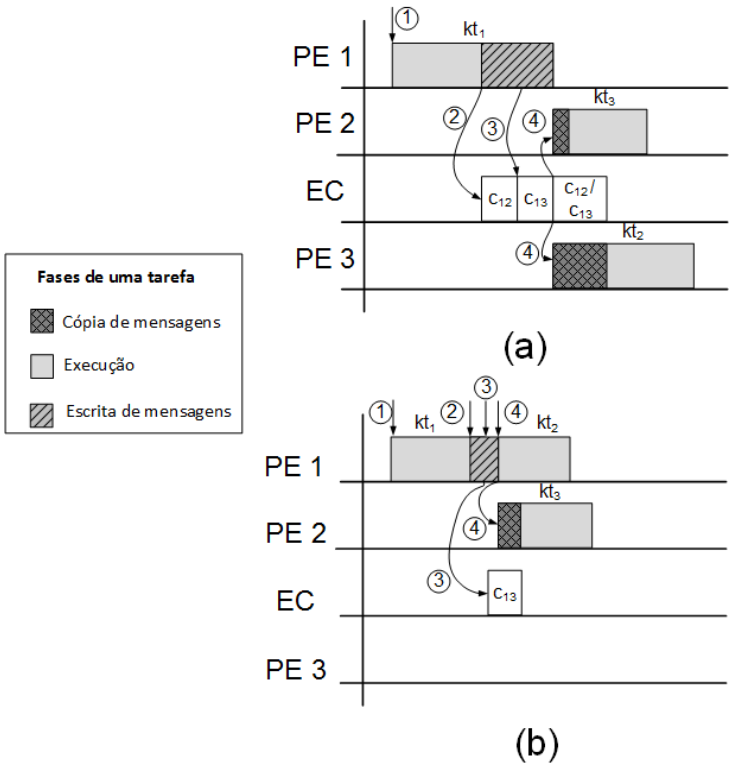


Figura 4.5: Diagramas de Gantt para ilustrar para ilustrar as fases de execução de uma tarefa. *PE1*, *PE3* e *PE2* são processadores, e *EC* é um barramento.

A partir do comportamento descrito na Figura 4.3, é possível dividir a execução de uma tarefa em três fases principais: (i) cópia das mensagens de entrada (linha 12 da Figura 4.3), (ii) execução (linha 13 da Figura 4.3), e (iii) escrita das mensagens de saída (linha 14 da Figura 4.3). Para ilustrar essas fases, a Figura 4.5 mostra diagramas de Gantt, considerando dois cenários de execução diferentes para as tarefas representadas pelos atores t_1, t_2 e t_3 do HSDG da Figura 4.2. A seguir, estas tarefas serão chamadas, respectivamente, de kt_1, kt_2 e kt_3. As mensagens enviadas através dos canais de comunicação representados pelos arcos (t_1,t_2) e (t_1,t_3) do HSDG da Figura 4.2 serão chamadas, respectivamente, de $c_{1,2}$ e $c_{1,3}$.

No cenário apresentado na Figura 4.5a, as tarefas kt_1, kt_2 e kt_3 estão mapeadas, respectivamente, em processadores denominados de *PE1*, *PE3* e *PE2*. Os canais de comunicação entre estas tarefas estão implementados como filas em uma memória compartilhada. Essa memória compartilhada é acessível a todos os processadores, e os processadores a acessam por meio de um barramento denominado de *EC*. Os círculos com número nos diagramas de Gantt representam eventos, que serão descritos a seguir. No evento 1, uma interrupção foi recebida pelo sistema, e isso fez com que kt_1 começasse a executar. No evento 2, kt_1 sai da fase de execução e começa a escrever uma mensagem para a kt_2. No evento 3, kt_1 termina de escrever a mensagem para kt_2, e começa a escrever uma mensagem para kt_3. No evento 4, kt_1 termina de escrever a mensagem para kt_3, e avisa kt_2 e kt_3 que novas mensagens foram enviadas. Neste momento, kt_2 e kt_3 começam a executar, iniciando também a cópia das mensagens recebidas. Note que, durante a cópia e escrita de mensagens, o barramento *EC* é ocupado pelo tráfego gerado por estas atividades.

O cenário da Figura 4.5b descreve o mesmo cenário apresentado anteriormente (Figura 4.5a), exceto que, na Figura 4.5b, kt_1 e kt_2 estão ambas mapeadas no processador *PE1*. A descrição dos eventos (círculos com número) nos dois cenários é a mesma. Como kt_1 e kt_2

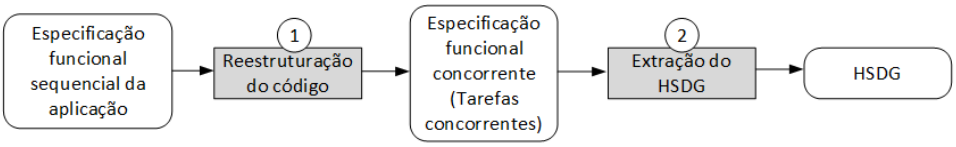


Figura 4.6: Fluxo de atividades do processo de especificação da aplicação.

executam no mesmo processador, o canal de comunicação entre elas é implementado como uma fila na memória local do processador. Portanto, as mensagens enviadas de kt_1 para kt_2 não geram tráfego no barramento *EC*. Ademais, como essas mensagens ficam na memória local, note que kt_2 não possui a fase de cópia de mensagens de entrada.

Fluxo de especificação: A Figura 4.6 apresenta o fluxo de atividades proposto para especificar uma aplicação. As caixas cinzas representam atividades, e as brancas suas respectivas informações de entrada e saída. Os números nas caixas cinzas são os passos que devem ser seguidos pelo projetista, e são descritos a seguir. Partindo de uma especificação funcional sequencial (ex.: código na linguagem C), o projetista deve reestruturar a aplicação para que ela tenha sua concorrência exposta (passo 1). A reestruturação deve transformar o código sequencial inicial em um código composto por um conjunto de tarefas concorrentes que seguem a implementação conceitual apresentada na Figura 4.3. Depois disso, o projetista deve extrair o HSDG correspondente desta nova especificação (passo 2). Este trabalho provê uma biblioteca de funções para facilitar o processo de implementação das tarefas concorrentes em uma arquitetura multiprocessada. Mais especificamente, a biblioteca fornece um conjunto de funções de alto nível (API) para gerenciar a troca de mensagens entre tarefas. Esta biblioteca será apresentada na Seção 4.2.

4.1.2 Especificação da plataforma de *hardware*

A especificação da arquitetura é composta por duas informações: (i) a estrutura da plataforma de *hardware*, e (ii) o tipo de escalonamento adotado nos processadores.

Estrutura da plataforma de *hardware*: A estrutura da plataforma é especificada por um grafo denominado de *Architecture Template Graph* (ATG). Esse grafo contém todos os elementos da plataforma de *hardware* que podem ser alocados (escolhidos) pelo processo de exploração do espaço de projeto (tarefas só podem ser mapeadas em processadores que tenham sido alocados).

Definição 4.1 (ATG). Um ATG $G_{atg} = (V, E)$ é um grafo direcionado com um conjunto de vértices $v \in V$ representando os elementos da plataforma de *hardware* e um conjunto de arcos $e \in E$ representando conexões direcionadas, onde $E \subseteq V \times V$. O conjunto de vértices é formado por dois conjuntos disjuntos, o conjunto de vértices que representam elementos de processamento V_{ep} e o conjunto de vértices que representam elementos de comunicação V_{ec}, tal que $V = V_{ep} \cup V_{ec}$.

Os elementos de comunicação (ECs) são responsáveis pela comunicação entre os elementos de processamento (EPs). A Figura 4.7 mostra exemplos de ATG (lado esquerdo), e as plataformas de *hardware* em que eles foram baseados (lado direito). As seguintes formas de conexão entre os elementos são apresentadas na figura: conexão ponto a ponto (Figura 4.7a), barramento compartilhado (Figura 4.7b) e *crossbar* (Figura 4.7c). Durante a exploração do espaço de projeto, as tarefas são mapeadas nos EPs, e os canais de comunicação são mapeados nos ECs ou nos EPs. Neste trabalho, assumimos que cada EP possui sua própria memória local de instruções/dados, que não é acessível aos outros EPs. Além disso, considera-se que cada EC

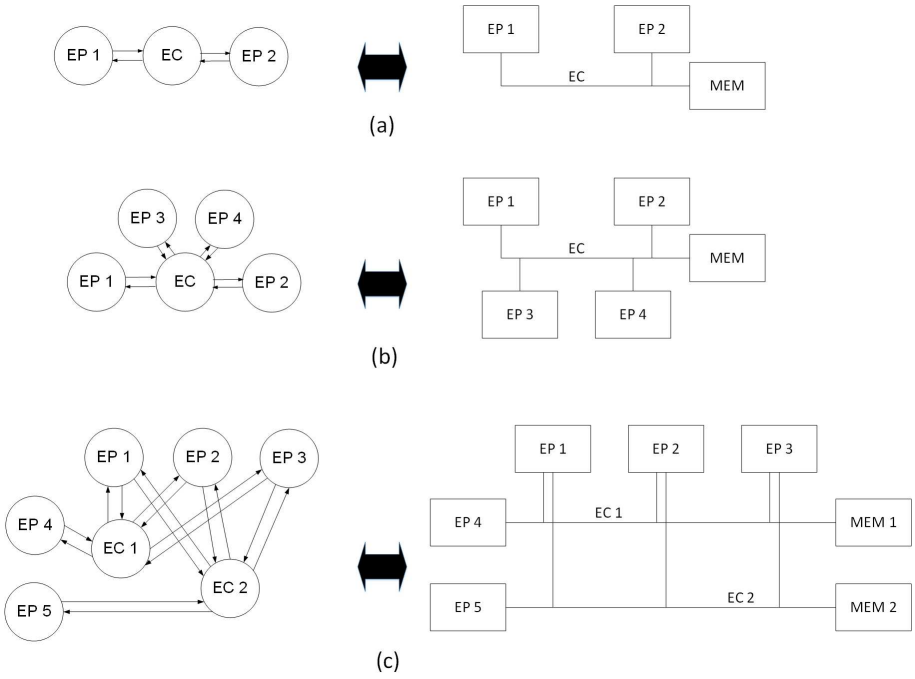


Figura 4.7: Exemplos de ATG (lado esquerdo), considerando diferentes formas de conectar os elementos: (a) conexão ponto a ponto, (b) barramento compartilhado, (c) *crossbar*

está associado a uma memória compartilhada (chamadas de MEM na Figura 4.7), que é usada para armazenar as mensagens enviadas através dos canais de comunicação mapeados no EC.

Escalonamento: Quando mais de uma tarefa é mapeada em um mesmo EP, uma política de escalonamento deve ser adotada. Este trabalho considera a política de escalonamento de prioridade fixa, devido à sua simplicidade e porque a maioria dos sistemas operacionais de tempo-real a suportam (ex.: FreeRTOS (BARRY, 2010), RTAI/Linux (MANTEGAZZA; DOZIO; PAPACHARALAMBOUS, 2000)). Nessa abordagem, caso duas tarefas estejam prontas para executar, a próxima a executar será sempre a que tiver maior prioridade. Neste trabalho, as prioridades das tarefas são definidas em tempo de projeto (durante o processo de exploração do espaço de projeto). Caso em um dado momento nenhuma tarefa esteja pronta para executar, assumimos que o sistema operacional coloca o processador em estado ocioso, fazendo com que a potência consumida por ele seja menor.

O projetista deve definir que elementos de processamento suportam escalonamento com preempções. A definição sobre o tipo de escalonamento adotado por cada EP é representada da seguinte forma: $sched : V_{ep} \rightarrow \{Preemp, NPreemp\}$. Caso o elemento suporte preempções (*Preemp*), então, neste elemento, a qualquer momento uma tarefa de menor prioridade pode ser interrompida pelo sistema operacional para que outra tarefa de maior prioridade possa executar. Caso o elemento não suporte preempções (*NPreemp*), então para que a tarefa de maior prioridade possa executar, ela precisa esperar que a de menor prioridade seja bloqueada pelo sistema operacional quando a última tenta ler um canal vazio (linhas 4-7 da Figura 4.3a) ou escrever em um canal cheio (linhas 8-11 da Figura 4.3a).

Quando dois ou mais elementos de processamento acessam ao mesmo tempo um elemento de comunicação, este trabalho assume que uma política de escalonamento *round-robin* é adotada (TANENBAUM; BOS, 2014). De acordo com essa política, cada elemento de processamento recebe autorização para acessar o elemento de comunicação por uma fração de tempo igual, e os acessos são feitos de forma circular.

4.2 Biblioteca de funções para comunicação entre tarefas

Esta seção apresenta a biblioteca desenvolvida para dar suporte ao desenvolvimento de aplicações que executam em arquiteturas compostas por processadores heterogêneos e programáveis. A biblioteca fornece um conjunto de funções para gerenciar a comunicação entre tarefas mapeadas no mesmo processador ou entre tarefas mapeadas em processadores diferentes, permitindo que o projetista possa abstrair detalhes de implementação de baixo nível, como semáforos e gerenciamento de interrupções. Usando a biblioteca, o projetista pode rapidamente desenvolver aplicações cujas tarefas seguem a implementação conceitual descrita na Figura 4.3.

A biblioteca proposta foi construída sobre o sistema operacional de tempo-real FreeRTOS (BARRY, 2010), ou seja, algumas das funções da biblioteca fazem chamadas a API desse sistema operacional. Este sistema operacional foi escolhido dentre as outras opções de sistemas operacionais de tempo-real por três motivos: (i) ele tem o código aberto, (ii) ele exige pouco espaço na memória, e (iii) ele suporta diversos tipos de processadores para sistemas embarcados. Como consideramos arquiteturas heterogêneas, uma instância do FreeRTOS é executada em cada processador da arquitetura. A biblioteca em conjunto com o sistema operacional FreeRTOS fornece todo o suporte necessário para escalonar e gerenciar as tarefas da aplicação.

4.2.1 Comunicações intraprocessador

Na biblioteca proposta, quando duas tarefas que se comunicam estão mapeadas no mesmo processador, as comunicações entre elas são realizadas através de canais de comunicação chamados de canais locais. Canais locais são implementados como filas armazenadas na memória local do processador. A Tabela 4.1 apresenta as funções fornecidas para gerenciar um canal local, e a Figura 4.8 apresenta um exemplo de como canais locais são inicializados e manipulados. Neste exemplo, uma tarefa (*task2*) faz uso de dois canais locais (*ichan* e *ochan*). O canal *ichan* é usado pela tarefa para ler mensagens, e o canal *ochan* para escrever mensagens. A Figura 4.8a mostra como os canais *ichan* e *ochan* são inicializados. A inicialização do canal *ichan* indica que a fila desse canal tem capacidade máxima de duas mensagens, e o tamanho de cada mensagem nesse canal é igual ao tamanho ocupado pela estrutura *MsgType1*. A inicialização do canal *ochan* indica que a fila associada a esse canal tem capacidade máxima de uma mensagem, e o tamanho das mensagens do canal é igual ao tamanho ocupado pela estrutura *MsgType2*.

O código da tarefa *task2* (Figura 4.8b) mostra que ela executa repetidamente o seguinte comportamento: primeiro, ela usa a função *vLocalChannelRead* para verificar se existe uma mensagem na fila do canal *ichan* (linha 5). Caso existam mensagens na fila deste canal, essa função retorna o endereço da primeira mensagem na fila. Caso contrário, a tarefa é bloqueada (ela é suspensa pelo sistema operacional) e, neste caso, a tarefa *task2* só será desbloqueada quando uma mensagem for adicionada à fila deste canal por outra tarefa. Em seguida, a função *xLocalChannelGetFreeSpace* é chamada (linha 6). Esta função retorna a próxima posição livre para escrita na fila do canal *ochan*. Caso a fila do canal *ochan* esteja cheia, essa função faz com que a tarefa seja bloqueada. Neste caso, a tarefa só será desbloqueada quando alguma mensagem for removida da fila do canal *ochan*. Em seguida, a tarefa executa sua atividade (linha 7). Depois, o resultado da computação é escrito na mensagem de saída (linha 8). Por último, a tarefa remove a primeira mensagem da fila do canal *ichan* (linha 9), e notifica o canal *ochan* que uma mensagem foi escrita (linha 10). É possível observar que a especificação funcional da tarefa segue implementação conceitual descrita na Figura 4.3. A Figura 4.9 mostra como essa tarefa poderia ser representada em um HSDG, em que o ator t_2 representa a tarefa *task2*, os arcos (t_1, t_2) e (t_2, t_3) representam, respectivamente, os canais *ichan* e *ochan*, e os arcos limitantes (t_2, t_1) e

Função	Descrição	Argumentos
xLocalChannelInit	Inicia um novo canal.	- Capacidade máxima da fila de mensagens do canal. - Tamanho de uma mensagem (em *bytes*).
xLocalChannelGetFreeSpace	Disponibiliza um ponteiro para a próxima posição livre na fila do canal. Caso não haja espaço na fila, a tarefa que chamou a função é bloqueada (tem seu fluxo de execução suspenso pelo sistema operacional). A tarefa só será desbloqueada quando houver espaço na fila.	- Canal que deve fornecer uma posição livre.
vLocalChannelRemoveToken	Remove do canal a mensagem no início da fila.	- Canal que terá uma mensagem removida.
vLocalChannelAddToken	Avisa ao canal que uma mensagem foi escrita.	- Canal que deve ser avisado.
vLocalChannelRead	Lê o endereço da mensagem no início da fila. Caso não haja mensagens na fila do canal, a tarefa que chamou a função é bloqueada. A tarefa só será desbloqueada quando houver pelo menos uma mensagem na fila do canal.	- Canal a ser lido. - *Buffer* onde o endereço da mensagem no início da fila será escrito.

Tabela 4.1: Funções para comunicação entre tarefas mapeadas em um mesmo processador.

Inicialização dos canais *ichan* e *ochan*

```
1. LocalChannelHandle_t ichan;
2. LocalChannelHandle_t ochan;
3. ichan = xLocalChannelInit(2, sizeof(struct MsgType1));
4. ochan = xLocalChannelInit(1, sizeof(struct MsgType2));
```

(a)

Definição de *task2*

```
1.  void task2() {
2.     struct MsgType1 *imsg;
3.     struct MsgType2 *omsg;
4.      for(;;) {
5.        vLocalChannelRead(ichan, &imsg);
6.        omsg = (struct MsgType1 *) xLocalChannelGetFreeSpace(ochan);
7.        IMDCT(imsg->mp3DecInfo, imsg->gr, 0);
8.        memcpy(omsg, imsg->mp3DecInfo, sizeof(struct MsgType2));
9.        vLocalChannelRemoveToken(ichan);
10.       vLocalChannelAddToken(ochan);
11.     }
12. }
```

(b)

Figura 4.8: Exemplo de uso das funções para comunicação intraprocessador.

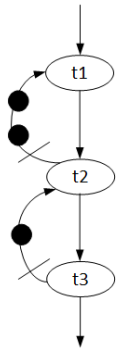


Figura 4.9: HSDG contendo o ator t_2, que poderia ser usado para representar a tarefa *task2*, mostrada na Figura 4.8.

(t_3, t_2) representam, respectivamente, a capacidade máxima dos canais *ichan* e *ochan*.

4.2.2 Comunicações interprocessador

Na biblioteca proposta, canais de comunicação entre tarefas mapeadas em processadores diferentes são implementados como filas na memória compartilhada. Cada fila armazenada na memória compartilhada possui um identificador único, e é gerenciada por um par de canais: canal emissor e canal receptor. A tarefa emissora das mensagens no canal de comunicação deve estar associada ao canal emissor, e a tarefa receptora deve estar associada ao canal receptor. A Tabela 4.2 apresenta as funções fornecidas para gerenciar comunicações interprocessador, e a Figura 4.10 apresenta um exemplo de uso dessas funções. O exemplo mostra uma tarefa (*task2*) que está associada a um canal receptor (*ichan*) e a um canal emissor (*ochan*). A Figura 4.10a mostra como esses dois canais são inicializados. Esta figura indica que: a fila associada ao canal receptor *ichan* tem capacidade máxima de duas mensagens, e o identificador único desta fila é dado pela constante *ID_1*. Além disso, a Figura 4.10a mostra que: a fila associada ao canal emissor *ochan* tem capacidade máxima de uma mensagem, o identificador único da fila é dado pela constante *ID_2*, o tamanho das mensagens na fila deste canal é igual ao tamanho ocupado pela estrutura *MsgType2*, e o endereço de memória compartilhada onde a fila está armazenada é dado pela constante *BUFFER_END*.

A descrição funcional da tarefa é apresentada na Figura 4.10b. Essa descrição indica que a tarefa tem o seguinte comportamento: primeiro, ela chama a função *vReceiverChannelRead* para verificar se há uma mensagem na fila do canal *ichan* (linha 6), e depois chama a função *xSenderChannelGetFreeSpace* para tentar obter um espaço livre na fila do canal *ochan* (linha 7). Caso não existam mensagens na fila do canal *ichan*, a função *vReceiverChannelRead* bloqueia a tarefa. Caso contrário, ela retorna o endereço da primeira mensagem na fila de mensagens associada a *ichan*. De maneira similar, a função *xSenderChannelGetFreeSpace* bloqueia a tarefa caso não haja espaço livre na fila do canal *ochan*. Caso esta fila não esteja totalmente ocupada, a função retorna o endereço do próximo espaço livre na fila. Depois de obter espaço para uma nova mensagem, a tarefa copia a mensagem recebida da memória compartilhada para a memória local (linha 8). Em seguida, a tarefa executa sua atividade (linha 9). Depois, a mensagem de saída é escrita (linha 10), e a primeira mensagem da fila de *ichan* é removida (linha 11). Finalmente, o canal *ochan* é notificado de que uma nova mensagem foi escrita (linha 12). O leitor deve notar que a especificação funcional apresentada na Figura 4.10b está de acordo com a implementação conceitual apresentada na Figura 4.3.

O fluxo (protocolo) de execução do envio e recebimento de mensagens em que a tarefa

Função	Descrição	Argumentos
xSenderChannelInit	Inicia um novo canal emissor.	- Identificador único da fila associada ao canal. - Capacidade máxima da fila do canal. - Tamanho de uma mensagem (em *bytes*). - Endereço na memória compartilhada onde a fila do canal será armazenada.
vSenderChannelAddToken	Avisa ao canal emissor que uma mensagem foi escrita. Esta função irá gerar uma interrupção para o processador que tem a tarefa receptora. A interrupção é usada para avisar ao canal receptor que uma nova mensagem foi escrita.	- Canal emissor que deve ser avisado.
xSenderChannelGetFreeSpace	Disponibiliza um ponteiro para a próxima posição livre na fila associada ao canal. Caso não haja espaço na fila, a tarefa que chamou a função é bloqueada (tem seu fluxo de execução suspenso pelo sistema operacional). A tarefa só será desbloqueada quando houver espaço na fila.	- Canal emissor que deve fornecer uma posição livre.
xReceiverChannelInit	Inicia um novo canal receptor.	- Identificador único da fila que deve ser associada ao canal. - Capacidade máxima da fila associada ao canal.
vReceiverChannelRead	Lê o endereço da mensagem no início da fila do canal receptor. Caso não haja mensagens na fila do canal, a tarefa que chamou a função é bloqueada. A tarefa só será desbloqueada quando houver pelo menos uma mensagem na fila do canal.	- Canal receptor a ser lido. - *Buffer* onde o endereço da mensagem no início da fila será escrito.
vReceiverChannelRemoveToken	Remove uma mensagem do início da fila do canal receptor, e gera uma interrupção para o processador que tem a tarefa emissora para avisar que uma mensagem foi excluída.	- Remove do canal receptor uma mensagem no início da fila.

Tabela 4.2: Funções para comunicação entre tarefas mapeadas em processadores diferentes.

Inicialização dos canais *ichan* e *ochan*

```
1. ReceiverChannelHandle_t ichan;
2. SenderChannelHandle_t ochan;
3. ichan = xReceiverChannelInit(ID1, 2);
4. ochan = xSenderChannelInit(ID2, 1, sizeof(struct MsgType2), BUFFER_END);
```

(a)

Definição de *task2*

```
1.  void TaskT2() {
2.    struct MsgType1 *imsg;
3.    struct MsgType1 *imsg_copy = (struct MsgType1 *)
        pvPortMalloc(sizeof (struct MsgType1));
4.    struct MsgType2 *omsg;
5.    for(;;) {
6.      vReceiverChannelRead(ichan, &imsg);
7.      omsg = (struct MsgType2 *) xSenderChannelGetFreeSpace(ochan);
8.      memcpy(imsg_copy, imsg, sizeof (struct MsgType1));
9.      IMDCT(imsg_copy->mp3DecInfo, imsg_copy->gr, 0);
10.     memcpy(omsg, imsg_copy->mp3DecInfo, sizeof(struct MsgType2));
11.     vReceiverChannelRemoveToken(ChannelMainIMDCT1);
12.     vSenderChannelAddToken(ChannelIMDCT1Main);
13.   }
14. }
```

(b)

Figura 4.10: Exemplo de uso das funções para comunicação interprocessador.

emissora e a tarefa receptora estão em processadores diferentes é apresentado na Figura 4.11. A biblioteca adota um semáforo local para cada canal emissor e para cada canal receptor. O semáforo associado ao canal emissor indica a quantas posições livres existem na fila do canal correspondente, ou seja, caso esse semáforo seja igual a zero, então a fila do canal está cheia. De maneira similar, o semáforo associado ao canal receptor indica quantas mensagens estão presentes na fila do canal correspondente, isto é, caso esse semáforo seja igual a zero, então não existem mensagens na fila. Os semáforos são implementados através da API do sistema operacional. Como pode ser observado na Figura 4.11: (i) se dois processadores precisam se comunicar, é preciso que um possa enviar interrupções para o outro, e (ii) quando uma interrupção causada por uma mensagem interprocessador é recebida, esta interrupção é gerenciada por uma tarefa específica da biblioteca proposta.

4.2.3 Conjunto de canais

Com o intuito de aumentar a expressividade do modelo de aplicação proposto, foram desenvolvidas também as funções da Tabela 4.3, cujo objetivo é gerenciar um conjunto de canais. Considere o HSDG da Figura 4.12a. Suponha que as tarefas representadas pelos atores t_2 e t_4 possuem aspectos funcionais em comum e que por uma questão de modularização e/ou economia de memória, deseja-se que a implementação delas seja feita por uma única tarefa, chamada de *task24*. Neste caso, *task24* deve ficar pronta para executar sempre que houver uma mensagem nos canais representados pelo arcos (t_1,t_2) ou (t_3,t_4). A Figura 4.12b mostra como a implementação poderia ser feita, usando as funções da Tabela 4.3.

No exemplo apresentado, os canais representados pelos arcos (t_1,t_2) e (t_3,t_4) são chamados de *Channel1* e *Channel2*, respectivamente. A Figura 4.12b mostra que a inicialização do conjunto de canais, *GuiChannelSet*, é feita adicionando os canais *Channel1* e *Channel2* ao conjunto. *Channel1* é um canal local (comunicações intraprocessador), e *Channel2* é um canal receptor (comunicações interprocessador). A definição da tarefa *task24* mostra que, inicialmente,

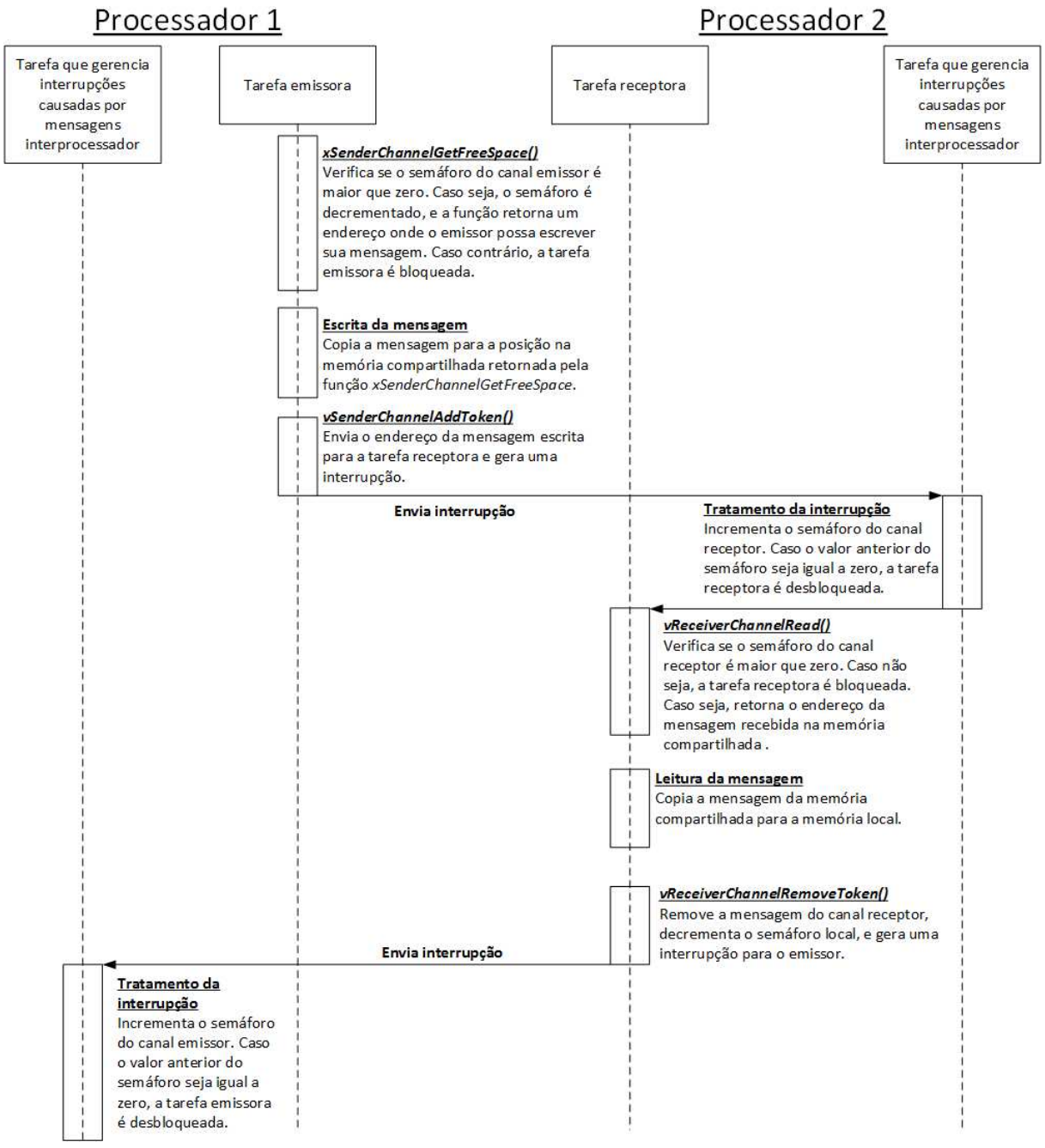


Figura 4.11: Fluxo de envio e recebimento de mensagens em que as tarefas estão em processadores diferentes.

Função	Descrição	Argumentos
xChannelCreateSet	Cria um novo conjunto de canais.	- Somatório do número máximo de mensagens que pode ser armazenado em cada canal do conjunto.
vLocalChannelAddToSet	Adiciona um canal local (intraprocessador) ao conjunto.	- Conjunto de canais em que o canal será adicionado. - Canal local a ser adicionado.
vReceiverChannelAddToSet	Adiciona um canal receptor (interprocessador) ao conjunto.	- Conjunto de canais em que o canal será adicionado. - Canal receptor a ser adicionado.
xChannelSelectFromSet	Seleciona do conjunto de canais um canal que possua uma mensagem em sua fila.	- Conjunto de canais em que o canal será selecionado.
xLocalChannelActivatedMember	Verifica se um dado canal local foi o canal escolhido pela função *xChannelSelectFromSet*.	- Membro do conjunto a ser testado. - Canal local a ser testado.
xReceiverChannelActivatedMember	Verifica se um dado canal receptor foi o canal escolhido pela função *xChannelSelectFromSet*.	- Membro do conjunto a ser testado. - Canal receptor a ser testado.

Tabela 4.3: Funções para gerenciamento de um conjunto de canais.

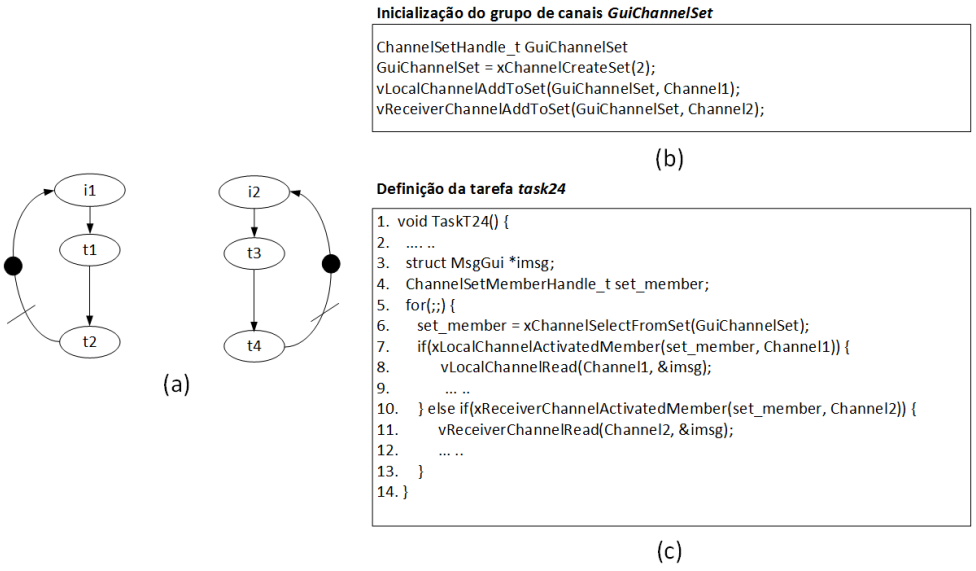


Figura 4.12: Exemplo de uso das funções para gerenciamento de um conjunto de canais.

ela executa a função *xChannelSelectFromSet* (linha 6). Esta função retorna um canal do conjunto de canais que possua ao menos uma mensagem em sua fila, e, caso não haja mensagens na fila de nenhum deles, a tarefa é bloqueada. A tarefa só será desbloqueada caso uma mensagem seja inserida na fila de *Channel1* ou *Channel2*. Depois que uma mensagem é inserida na fila de um desses canais, as funções *xLocalChannelActivatedMember* (linha 7) e *xReceiverChannelActivatedMember* (linha 10) são usadas para verificar qual dos canais foi retornado pela função *xChannelSelectFromSet*. Por fim, dependendo do canal retornado, ações especificas são tomadas (linhas 8-9 e linhas 11-12). Para que o modelo HSDG seja representativo em relação à decisão de implementar as tarefas representadas pelos atores t_2 e t_4 em uma mesma tarefa, é preciso adicionar restrições para garantir que, nas soluções encontradas pelo processo de exploração do espaço de projeto, as tarefas representadas por t_2 e t_4 estejam mapeadas no mesmo processador, e que elas tenham a mesma prioridade. Como será mostrado, os algoritmos de exploração propostos permitem que esse tipo de restrição seja adicionado.

4.3 Caracterização

Como pode ser observado na Seção 4.1, os modelos de especificação propostos para a aplicação e plataforma de *hardware* não possuem informações sobre custo monetário, potência consumida e tempo de execução. Durante a atividade de caracterização, essas informações devem ser obtidas e associadas a esses modelos. Nesta seção, inicialmente serão apresentados os métodos propostos para que o projetista possa obter as informações sobre o tempo de execução e potência consumida, e, em seguida, serão descritas as informações que é preciso obter durante a atividade de caracterização.

4.3.1 Infraestrutura de medição

Os métodos de caracterização propostos se baseiam em medições no sistema real. Os seguintes artefatos são necessários para executar a atividade de caracterização (Figura 4.13): (i) a especificação funcional da aplicação, (ii) a plataforma de *hardware*, e (iii) um conjunto representativo de entradas da aplicação. Caso um sistema real (ou um similar) não esteja disponível, existem ao menos quatro outras maneiras para que o projetista possa obter as informações necessárias: (i) métodos analíticos (GAUTAMA; GEMUND, 2000), (ii) documentos técnicos, (iii) com base na experiência do projetista em projetos anteriores, ou (iv) simulação de baixo nível (BENINI et al., 2005; AUSTIN; LARSON; ERNST, 2002; BROOKS; TIWARI; MARTONOSI, 2000). A seguir, são apresentados os métodos propostos para medir tempo de execução e potência consumida.

Tempos de execução: Na abordagem proposta, para medir o tempo de execução de um trecho de código, esse trecho deve ser instrumentado com as seguintes macros de medição (Figura 4.14): *MES_START(id)* e *MES_END(id)*, em que *MES_START(id)* indica o início do código a ser medido, *MES_END(id)* indica o fim, e *id* é o identificador único que deve estar associado ao trecho. Existem diversas formas de se implementar as macros *MES_START* e *MES_END*. A forma adotada por este trabalho consiste em zerar um temporizador (*timer*) do *hardware* na chamada de *MES_START*, e na chamada de *MES_END* capturar o valor atual deste *timer*. O valor capturado indica o tempo de execução do trecho de código compreendido entre as macros *MES_START* e *MES_END*. Esse valor e o identificador do trecho de código medido são enviados pela porta serial para um *desktop* para que os dados possam ser posteriormente tratados (Figura 4.15). Como cada trecho de código tem seu próprio identificador, vários trechos podem ser medidos em uma mesma execução da aplicação, bastando garantir que, durante a execução

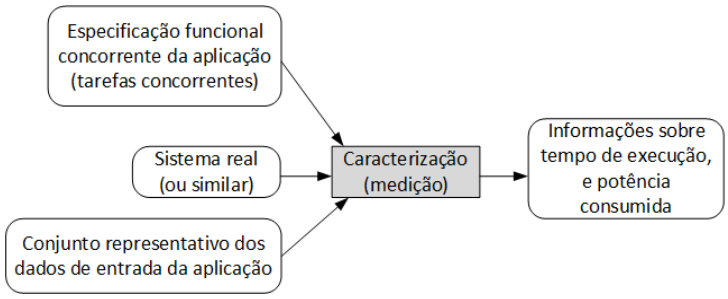


Figura 4.13: Caracterização dos tempos de execução e potência consumida dos elementos de *hardware*. Para esta atividade, os seguintes artefatos são necessários: (i) a especificação funcional da aplicação, (ii) a plataforma de *hardware*, e (iii) um conjunto representativo de entradas da aplicação.

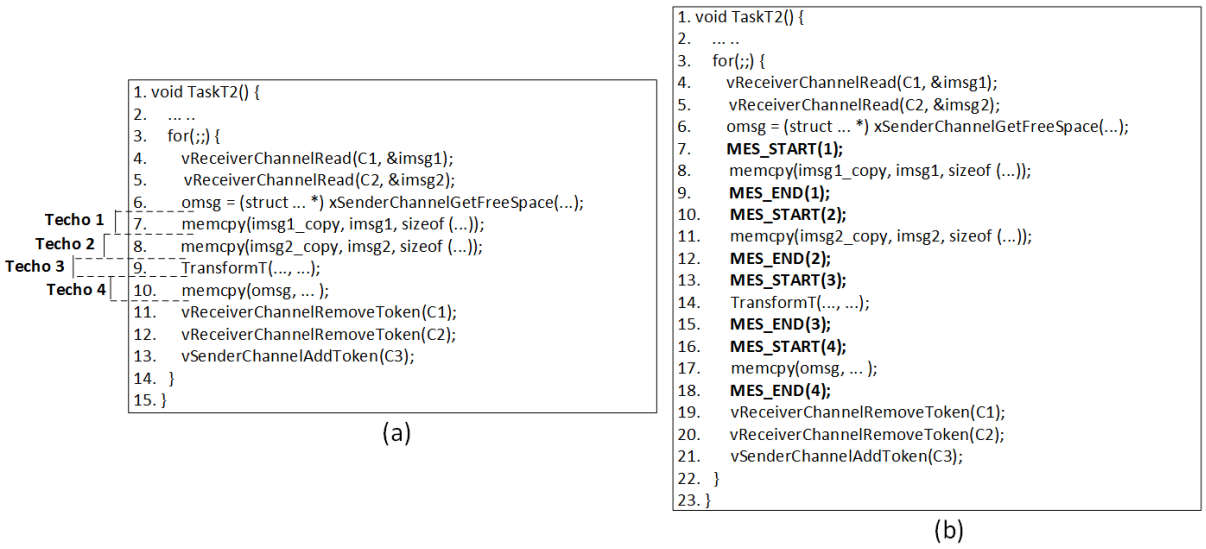


Figura 4.14: Instrumentação do código.

do trecho a ser medido, essa execução não seja interrompida pelo sistema operacional ou por outras tarefas da aplicação. Essa estratégia de medição permite que o tempo de execução dos trechos de código sejam medidos com precisão de aproximadamente 10 s.

Potência consumida: A Figura 4.16 mostra a infraestrutura de medição adotada para medir a potência consumida de uma plataforma de *hardware*. Na figura, um osciloscópio captura (Canal 2) a corrente consumida através da medição da queda de tensão em um resistor de alta precisão. É importante que a resistência do resistor seja baixa para que não interfira no correto funcionamento do *hardware*. Neste trabalho, um resistor de 1 Ohm foi suficiente. O osciloscópio também é conectado a uma porta de I/O do *hardware* (Canal 1), que é usada para indicar os tempos de início e fim de uma medição. Os dados capturados pelo osciloscópio são enviados para um computador executando a ferramenta AMALGHMA (*Advanced Measurement Algorithms for Hardware Architectures*) (SILVA et al., 2014; NOGUEIRA et al., 2010). Essa ferramenta, que foi desenvolvida pelo autor desta tese em conjunto com os membros de seu grupo de pesquisa, adota uma série de técnicas estatísticas e de amostragem para garantir dados com confiança.

Como será mostrado na próxima subseção, o projetista precisa medir a potência média consumida pelos processadores da plataforma, considerando diferentes frequências de operação da plataforma, e diferentes níveis de utilização dos processadores. A Figura 4.17 apresenta um exemplo de código (tarefa) para caracterização da potência consumida de um processador.

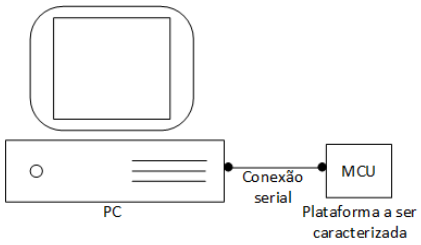


Figura 4.15: Configuração utilizada para capturar os tempos de execução das tarefas.

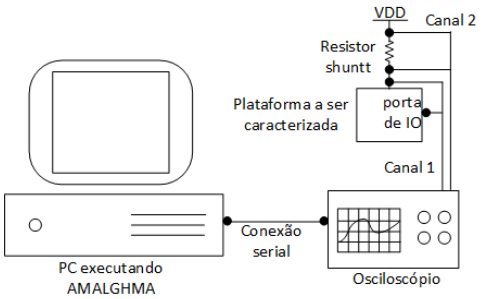


Figura 4.16: Configuração utilizada para capturar as informações sobre potência consumida.

A tarefa com o código de caracterização deve ser a única tarefa em execução pelo sistema operacional. Nas linhas 4 e 9, o código escreve em uma porta de I/O do *hardware* para avisar à ferramenta AMALGHMA sobre o início e o fim de uma medição. Desta maneira, esta ferramenta irá medir a potência média consumida pelo trecho de código compreendido entre essas duas linhas. Este código faz uso da função *Delay* (linhas 7 e 10), que faz parte da API do sistema operacional. Essa função faz com que o sistema operacional suspenda a tarefa em execução por um período de tempo dado pelo argumento da função. Depois desse período, a tarefa volta a executar. Essa função é usada na linha 6 para controlar o nível de utilização do processador, e na linha 8 ela é usada para delimitar a janela de medição. A Figura 4.18 mostra a tela do osciloscópio durante uma medição, em que é possível observar a janela de medição capturada pelo Canal 1.

O tempo da janela de medição (T) pode ser obtido pela equação abaixo:

$$T = (X_f - X_i) \times UT,$$

onde X_i e X_f são os pontos de início e fim da janela de medição (Figura 4.18), e UT é a unidade

```
1. void TaskT2() {
2.    ... ..
3.    for(;;) {
4.      PINUP();   // Levanta pino da porta de I/O
5.      for(i = 0; i < 10; i++) {
6.           Workload_Function();  // Ativo
7.           Delay(IDLE_TIME); // Repouso
8.      }
9.      PINDOWN(); // Abaixa pino da porta de I/O
10.     Delay(PERIOD);   // Guarda
11.   }
12. }
```

Figura 4.17: Exemplo de código para caracterização da potência consumida.

Figura 4.18: Sinais capturados pelo osciloscópio durante uma medição.

de tempo adotada no osciloscópio. Considerando que se sabe o tempo da janela de medição, e o tempo dentro desta janela em que nada estava sendo executado (linha 6 da Figura 4.17), é possível identificar e controlar o nível de utilização do processador. Além disso, como a queda de tensão (V_r), a resistência do resistor (R), e a tensão de alimentação da plataforma (V_{cc}) são conhecidas, então a potência consumida (P) pode ser calculada de acordo as seguintes equações:

$$I = Vr/R$$

$$P = Vcc \times I$$

4.3.2 Informações a serem obtidas

As seguintes informações sobre tempo de execução precisam ser obtidas pelo projetista: (i) tamanho das mensagens de cada canal, (ii) os tempos de execução das tarefas (incluindo os tempos de comunicação), (iii) os intervalos de tempo de chegada dos eventos do ambiente externo, (iv) os requisitos temporais (*deadlines*) da aplicação, e (v) o *overhead* do sistema operacional.

Tamanho das mensagens: O projetista deve informar o tamanho (em *bytes*) das mensagens de cada canal de comunicação da aplicação. Na biblioteca proposta, o tamanho das mensagens em cada canal é constante (recordar as Tabelas 4.1 e 4.2).

Tempos de execução das tarefas: Como apresentado na Figura 4.5, a execução de uma tarefa pode ser dividida em três fases principais: fase de execução, fase de cópia das mensagens recebidas, e fase de escrita das mensagens. Assim, informações sobre essas três fases precisam ser obtidas. Para representar o tempo da fase de execução, o projetista precisa definir as distribuições de probabilidade das seguintes variáveis aleatórias:

- $ET_{i,j}$ = variável aleatória que representa tempo da fase de execução da tarefa t_i no elemento de processamento v_j.

Para representar as fases de cópia das mensagens recebidas, e fase de escrita das mensagens, o projetista precisa definir os seguintes tempos:

- $\mathbb{E}(COPY_{i,z,j,w})$ = tempo médio para o elemento de processamento v_j copiar da memória compartilhada para a memória local a mensagem enviada pela tarefa t_i para a tarefa t_z, quando o canal de comunicação (i,z) está mapeado no elemento de comunicação v_w.

- $\mathbb{E}(COPY^{*}_{i,z,j})$ = tempo médio para o elemento de processamento v_j copiar de uma posição da memória local para outra posição da local a mensagem enviada pela tarefa t_i para a tarefa t_z, quando o canal de comunicação (i, z) está mapeado no elemento de processamento v_j.

Os tempos representados por $COPY_{i,z,j,w}$ e $COPY^{*}_{i,z,j}$ são aproximadamente constantes, uma vez que o tamanho das mensagens também é constante. Se, para uma dada combinação i, z, j e w, alguma das informações acima não estiver definida, os algoritmos de exploração assumem que o mapeamento correspondente é inválido.

No método proposto, a definição das distribuições de probabilidade das variáveis aleatórias $ET_{i,j}$ pode ser feita por meio de arquivos de *trace* contendo amostras reais obtidas por medição, ou utilizando distribuições de probabilidade clássicas (ex.: exponencial, normal, uniforme). Quando uma abordagem por *traces* é adotada, os algoritmos de exploração automaticamente constroem uma tabela de probabilidades (distribuição empírica (ROBINSON, 2004)) cujas entradas têm o seguinte formato: (*val*, *prob*), onde *prob* é a frequência relativa com que *val* aparece no *trace*. Depois de criar a distribuição probabilidade empírica, essa distribuição é associada a variável aleatória correspondente.

Caso os tempos da i-ésima vez que uma tarefa executa estejam correlacionados com os tempos da i-ésima execução de outras tarefas, é preciso construir a distribuição de probabilidade conjunta desses tempos. Os tempos de execução de duas tarefas k e l estão correlacionados positivamente se, quando o tempo de execução de k aumenta (diminui), o tempo de execução de l também aumenta (diminui). Os tempos de k e l estão correlacionados negativamente se, quando o tempo de execução de k aumenta, o tempo de execução de l diminui. Portanto, é preciso capturar essa correlação para garantir estimativas precisas. Para isso, o projetista deve informar que tarefas possuem seus tempos correlacionados, e assegurar que haja uma correspondência linha a linha entre os valores presentes nos arquivos de *trace* associados a essas tarefas. A partir dos arquivos de *trace* das tarefas com tempos correlacionados, os algoritmos de exploração constroem uma tabela conjunta de probabilidades com o seguinte formato (val_1, val_2, ... , val_n, *prob*), onde *prob* é a frequência relativa com que val_1 (presente no *trace* da tarefa 1), val_2 (presente no *trace* da tarefa 2), ... , e val_n (presente no *trace* da tarefa n) aparecem simultaneamente na mesma linha de cada *trace*.

Como descrito na Subseção 4.3.1, através do método de medição proposto, *traces* para mais de um trecho de código podem ser obtidos em uma mesma execução da aplicação. Antes de iniciar a medição, o projetista deve reunir um conjunto representativo dos dados de entrada da aplicação. A Figura 4.14 mostra um exemplo de como *traces* para as fases de transferências de mensagens (variáveis aleatórias $COPY_{i,z,j,w}$ e $COPY^{*}_{i,z,j}$) e execução (variáveis aleatórias $ET_{i,j}$) podem ser obtidos. Neste exemplo, os dois primeiros trechos de código (linhas 7 e 8 da Figura 4.14a), copiam uma mensagem da memória compartilhada para a memória local. No terceiro trecho (linha 9), a tarefa entra na fase de execução. No último trecho, a tarefa copia uma mensagem da memória local para a memória compartilhada. Os tempos das chamadas de função da biblioteca (linhas 4-6 e 11-13 da Figura 4.14a) são muito pequenos e aproximadamente constantes e, portanto, podem ser medidos separadamente e posteriormente adicionados ao tempo da fase de execução da tarefa. Como exemplo, a Figura 4.19 mostra um código para medir o tempo de execução da chamada a função *vReceiverChannelRead*. O código chama duas funções da API do sistema operacional: *vTaskSuspend*, que suspende a execução de uma dada tarefa, e *vTaskResume*, que tira a suspensão de uma dada tarefa. Como uma chamada a função *vReceiverChannelRead* pode bloquear a tarefa caso não haja mensagens para serem lidas, é importante garantir que, durante a medição, a fila do canal não fique vazia. Dessa forma, no código exemplo, o propósito da tarefa *Task1* é garantir que sempre haverá mensagens para serem

```
1. void Task1() {
2.   for(;;) {
3.     (...)
4.     vSenderChannelAddToken(C1);
5.     vTaskResume(task2); // Tira a suspensão de Task2, que tem maior prioridade, fazendo
                           // com que ela execute imediatamente
6.   }
7. }
8.
9.   void Task2() {
10.     (...)
11.     for(;;) {
12.       (...)
13.       MES_START(1); // inicia a medição
14.       vReceiverChannelRead(C1, &imsg1);
15.       MES_END(1); // termina a medição
16.       vTaskSuspend(NULL); // Task2 se suspende, fazendo com que Task1 volte a executar
17.     }
18. }
```

Figura 4.19: Medição de uma chamada a uma função da biblioteca de funções proposta.

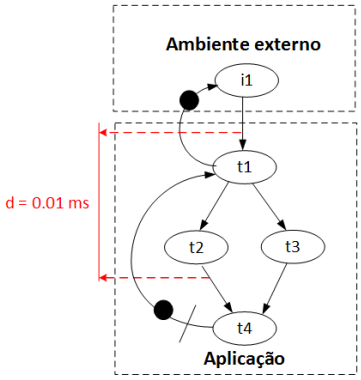


Figura 4.20: Exemplo de HSDG com um *deadline*.

lidas quando *vReceiverChannelRead* for chamada por *Task2* (*Task2* tem maior prioridade que *Task1*).

Intervalos de tempo dos eventos externos: Considere o HSDG da Figura 4.20. Como descrito na Seção 4.1, esse HSDG modela a situação em que quando a interrupção representada pelo disparo do ator $i_1 \in I$ é gerada, novas interrupções são desabilitadas pela rotina de tratamento de interrupções, e a reabilitação da interrupção só é feita após a execução da tarefa representada pelo ator t_1. Neste trabalho, o intervalo de tempo compreendido entre a reabilitação da interrupção e a geração de uma nova interrupção é descrito por uma variável aleatória chamada de IT_i, $i \in I$. Caso a interrupção nunca seja desabilitada (ex.: Figura 4.4b), essa variável aleatória descreve o intervalo de tempo entre a geração de duas interrupções consecutivas. Para cada ator $i \in I$ do HSDG, o projetista deve associar uma variável aleatória IT_i. Intuitivamente, IT_i representa o intervalo de tempo entre as chegadas das cargas de trabalho (*workload*) ao sistema. Assim como a definição das distribuições de probabilidade das variáveis aleatórias de tempo de execução $ET_{i,j}$, as distribuições de probabilidade das variáveis aleatórias IT_i pode ser feita por meio de *traces* contento intervalos reais de medição, ou por meio de distribuições de probabilidade clássicas. Caso uma abordagem por *traces* seja utilizada, a partir do *trace*, a distribuição de probabilidade empírica é automaticamente construída pelos algoritmos e associada a variável aleatória IT_i correspondente.

***Deadlines*:** O projetista deve fornecer, em tempo de projeto, os requisitos temporais da aplicação, que são definidos em termos de *deadlines* $d : C \times C \rightarrow \mathbb{R}^+$, onde C é o conjunto de arcos do HSDG. A Figura 4.20a mostra um exemplo de *deadline*, onde $d((i_1,t_1),(t_2,t_4)) = 0,01$ ms. Um *deadline* $d(c_j,c_l)$ representa o tempo máximo que uma mensagem no canal representado

```
void Task1() {
 for(;;) {
      MES_END(1); // termina a medição
     vTaskSuspend(NULL);  // suspende Task1
 }
}

void Task2() {
    for(;;) {
       Delay(100);
       MES_START(1); // Inicia a medição
       vTaskResume(task1); // força o início de Task1, que tem maior prioridade que Task2
    }
}
```

Figura 4.21: Código adotado para medição do tempo de mudança de contexto do sistema operacional.

pelo arco c_j demora para chegar ao canal representado pelo arco c_l. Caso a mensagem demore mais que o *deadline* para ir de um canal ao outro, considera-se que o sistema violou o *deadline*.

***Overhead* do sistema operacional:** O *overhead* do sistema operacional é composto por: (i) tempo de mudança de contexto, e (ii) tempo de execução da tarefa que trata interrupções geradas por mensagens interprocessador (Figura 4.11). Para cada processador, o valor médio desses tempos deve ser medido e informado. A Figura 4.21 mostra o código utilizado por este trabalho para medir o tempo de execução de uma mudança de contexto. O código mede o intervalo de tempo entre a suspensão de uma tarefa (*Task*2) e o início da execução de outra tarefa (*Task*1) de maior prioridade.

Custo monetário: Na atividade de caracterização, o projetista deve fornecer o custo monetário de todos os elementos que podem ser alocados da plataforma. O custo monetário de uma alternativa candidata é dado pela equação seguinte.

$$R(\Pi) = \sum_{i \in \Pi} r_i, \tag{4.1}$$

onde Π é o conjunto de elementos alocados da plataforma e r_i é o custo do elemento i.

Potência consumida: O modelo adotado para calcular a potência média consumida por uma arquitetura é dado a seguir:

$$P(\mu_1, ..., \mu_n, \mathfrak{f}) = b_0 + \mu_1 \times b_1 + \mu_2 \times b_2 + ... + \mu_n \times b_n + \mathfrak{f} \times b_{n+1}, \tag{4.2}$$

onde μ_1, ... , μ_n representam as utilizações dos processadores 1, 2, ... , n, $\mathfrak{f}$ é a frequência de operação da arquitetura e b_0, ... , b_{n+1} são os coeficientes que precisam ser estimados durante a atividade de caracterização. Neste trabalho, estes coeficientes são estimados usando regressão linear múltipla (WEISBERG, 2014). Dessa forma, o projetista deve medir a potência consumida pela arquitetura, considerando diferentes valores de utilização dos processadores e diferentes valores de frequência de operação da arquitetura. Seja $pw = (pw_1, pw_2, \ldots, pw_k)$ o vetor com as k medições de potência consumida, e seja $uw_i = (uw_{i,1}, uw_{i,2}, \ldots, uw_{i,n}, uw_{i,n+1})$ o vetor em que $uw_{i,1}$, ... , $uw_{i,n}$ são as utilizações de cada processador na i-ésima medição e $uw_{i,n+1}$ é a frequência da plataforma na i-ésima medição. Então, o modelo de regressão linear múltipla pode ser formulado da seguinte forma:

$$pw_i = b_0 + uw_{i,1} \times b_1 + uw_{i,2} \times b_2 + ... + uw_{i,n} \times b_n + uw_{i,n+1} \times b_{n+1} + \varepsilon_i \tag{4.3}$$

Neste trabalho, através da ferramenta Matlab (MATHWORKS, 2014), os coeficientes

$b_0, b_1, \ldots, b_{n+1}$ são obtidos pelo método dos mínimos quadrados. Este método foi escolhido devido a sua simplicidade e pela ampla literatura disponível (WEISBERG, 2014). O objetivo do método é achar $b_0, b_1, \ldots, b_{n+1}$ tal que

$$\sum_{i=1}^{k} \varepsilon_i^2 = \sum_{i=1}^{n+1} (pw_i - b_0 - uw_{i,1} \times b_1 - uw_{i,2} \times b_2 - \ \ldots \ - uw_{i,n} \times b_n - uw_{i,n+1} \times b_{n+1})^2, \quad (4.4)$$

é minimizado. Após obtenção dos coeficientes $b_0, b_1, \ldots, b_{n+1}$, o projetista pode verificar a qualidade do modelo de potência consumida através do coeficiente de determinação R^2 (WEISBERG, 2014):

$$R^2 = 1 - \frac{\sum_{i=1}^{k} \varepsilon_i^2}{\sum_{i=1}^{k} (pw_i - \overline{pw})^2}, \quad (4.5)$$

onde $\overline{pw} = \sum_{i=1}^{k} pw_i / k$. Valores altos de R^2 indicam que o modelo possui boa qualidade.

A subseção anterior mostrou como a potência consumida por uma plataforma de *hardware* pode ser medida, e como diferentes utilizações podem ser geradas nos processadores. Dado que estamos interessados em estimar a potência *média* consumida, é importante que a função $Workload\_Function()$ do código de caracterização (Figura 4.17) contenha trechos de código que são muito executados na aplicação. Apesar de simples, pesquisas na área mostram que modelos lineares como o da Equação (4.2) possuem exatidão relativamente boa para diversos processadores RISC utilizados em sistemas embarcados, como os processadores StrongARM (SINHA; CHANDRAKASAN, 2001), IBM PowerPC 405LP (RUSU et al., 2004) e ARM9 (RETHINAGIRI et al., 2012). Nos experimentos conduzidos por este trabalho, esse modelo se mostrou adequado para representar a potência consumida dos processadores ARM Cortex-M0 e Cortex-M4.

4.4 Considerações finais

Esse capítulo apresentou as atividades de especificação e caracterização do método proposto. Conforme foi visto nesse capítulo, na atividade de especificação, o projetista especifica a aplicação e a plataforma de *hardware* usando os modelos de especificação propostos. Esses modelos não possuem qualquer informação sobre tempo de execução da aplicação, potência consumida e custo monetário. Na atividade de caracterização, essas informações devem ser obtidas e associadas a esses modelos. O capítulo também apresentou o ambiente disponibilizado pelo método proposto para viabilizar as atividades de especificação e caracterização. Este ambiente é composto por uma biblioteca para implementar aplicações em arquiteturas multiprocessadas, e um conjunto de métodos para facilitar o processo de caracterização.

5 Modelos de simulação

Durante a exploração do espaço de projeto, a qualidade de uma alternativa de projeto é avaliada em termos de: (i) probabilidades de violação de *deadlines*, (ii) custo monetário, e (iii) potência consumida. A Seção 4.3 apresentou as expressões adotadas para avaliar o custo e a potência consumida de uma alternativa candidata. O presente capítulo mostra como as probabilidades de violação de *deadlines* são avaliadas.

No método proposto, probabilidades de violação de *deadlines* são avaliadas por simulação estacionária (PAWLIKOWSKI, 1990; BANKS, 1998). Para cada solução candidata explorada, um modelo de simulação é construído e avaliado. A construção dos modelos é feita a partir da combinação de blocos básicos de simulação. Quando combinados, esses blocos básicos são capazes de modelar o processador, o sistema operacional que executa em cada processador, e a aplicação. Os blocos básicos, assim como sua combinação, são especificados através do formalismo *Parallel* DEVS (P-DEVS) (ZEIGLER; PRAEHOFER; KIM, 2000; CHOW; ZEIGLER, 1994), que é usado para garantir que o comportamento concorrente dos sistemas embarcados seja representado de maneira precisa e sem ambiguidades (Seção 2.4). Conforme será visto neste capítulo, ao contrário da grande maioria dos métodos que adotam simulação, os modelos propostos são livres de detalhes funcionais da aplicação e da arquitetura, fazendo com que seja possível avaliar rapidamente uma alternativa de projeto. É importante ressaltar também que esses modelos são automaticamente construídos e, portanto, o projetista não precisa lidar diretamente com eles.

O restante do capítulo está organizado da seguinte forma: a Seção 5.1 apresenta uma visão geral dos blocos básicos de simulação propostos e como eles funcionam. A Seção 5.2 mostra como tempo de execução e comunicação são modelados. A Seção 5.3 fornece as especificações formais dos blocos básicos propostos. A Seção 5.4 detalha como um modelo de simulação é avaliado, e a Seção 5.5 conclui este capítulo.

5.1 Visão geral dos blocos básicos propostos

Quatro tipos de blocos básicos de simulação foram desenvolvidos: (i) *bloco tarefa*, que modela os atores do HSDG que representam tarefas, (ii) *bloco recurso*, que modela o processador e o sistema operacional, (iii) *bloco gerador*, que modela os atores do HSDG que representam os processos do ambiente externo, e (iv) *bloco monitor*, que monitora violações de *deadlines* do sistema. Os modelos construídos a partir da combinação desses blocos básicos são usados para co-simular a aplicação (representada pelo HSDG), e a arquitetura (representada pelo ATG e as informações geradas na etapa de caracterização). Os blocos básicos são modelos atômicos P-DEVS, e a composição desses blocos forma um modelo acoplado P-DEVS (recordar a definição de modelo atômico e modelo acoplado na Seção 2.4). Nesta seção, apresentaremos de maneira informal como os modelos de simulação funcionam e, na Seção 5.3, as especificações formais e

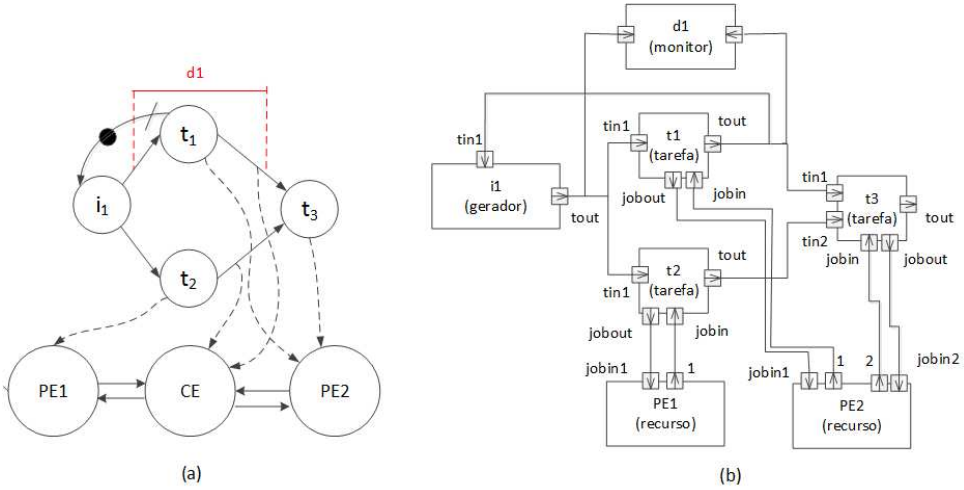


Figura 5.1: Mapeamento de um modelo de simulação.

maiores detalhes serão fornecidos.

Considere a especificação apresentada na Figura 5.1a, que mostra: (i) um HSDG $G_{hsdg} = (A = T = \{ t_1, t_2, t_3 \} \cup I = \{ i_1 \}, C = C_d = \{ (t_1,t_3), (t_2,t_3) \} \cup C_{lim} = \{ (t_1,i_1) \} \cup C_{int} = \{ (i_1,t_1), (i_1,t_2) \}, s_0 = \{ (t_1,t_3,0), (t_2,t_3,0), (t_1,i_1,1), (i_1,t_1,0), (i_1,t_2,0) \})$, (ii) um ATG $G_{atg} = (V = V_{ep} = \{ PE1, PE2 \} \cup V_{ec} = \{ CE \}, E = \{ (PE1,CE), (CE,PE1), (PE2,CE), (CE,PE2) \})$, (iii) um *deadline* $d((i_1,t_1), (t_1,t_3))$, e (iv) e uma relação de associação $MAP : T \cup C_d \rightarrow V$, que é usada para indicar onde as tarefas e canais de comunicação representados no HSDG estão mapeados nos elementos da plataforma representada pelo ATG. A Figura 5.1b mostra a estrutura do modelo P-DEVS acoplado que é gerada a partir da especificação na Figura 5.1a. Na Figura 5.1b, as caixas representam os blocos básicos (os nomes entre parênteses indicam o tipo de bloco básico), os arcos representam conexões entre as portas dos blocos básicos, e os quadrados com seta são portas (a direção da seta indica se é uma porta de saída ou de entrada, e o nome da porta é mostrado ao lado da porta).

No modelo P-DEVS, os blocos gerador e blocos tarefa (blocos básicos t_1, t_2, t_3 e i_1 da Figura 5.1b) são responsáveis por simular a regra de disparo do HSDG. As portas de entrada tin_1, ... , tin_j, ... , tin_n, e a porta de saída *tout* de cada um desses blocos básicos é usada, respectivamente, para modelar o recebimento e o envio de *tokens*. Para cada arco $(c_i, c_j) \in C$ no HSDG, existe uma conexão correspondente entre uma porta *tout* e outra tin_j no modelo P-DEVS. A quantidade de *tokens* nos arcos do HSDG é modelada por contadores internos aos blocos tarefa/gerador, de maneira que cada porta tin_j de um bloco tarefa/gerador está associada a um contador, que indica quantos *tokens* existem atualmente no arco correspondente. Assim, para saber se estão prontos para disparar, estes blocos verificam se cada contador interno de *tokens* é maior que zero.

Como precisamos co-simular a aplicação e a arquitetura, no modelo P-DEVS, os blocos tarefa são associados (mapeados) aos blocos recurso (blocos básicos $PE1$ e $PE2$ da Figura 5.1b). Cada bloco recurso modela o processador e o sistema operacional de tempo-real que executa nele. As conexões entre as portas de um bloco recurso e um bloco tarefa indicam que a tarefa representada pelo bloco tarefa está mapeada no processador representado pelo bloco recurso. O leitor deve notar que o modelo P-DEVS não incorpora os arcos do ATG. Esta informação só é usada para definir se um mapeamento é viável (Seção 6.1). Além disso, apesar de considerar os tempos de comunicação (ver próxima seção), os elementos de comunicação não são explicitamente representados no modelo P-DEVS.

Resumidamente, o modelo P-DEVS funciona da seguinte forma: suponha que o bloco tarefa t_2 fica pronto para disparar (o contador interno de *tokens* associado à sua porta tin_1 passou de zero para um). Neste momento, uma carga de trabalho é submetida ao bloco recurso $PE1$ ao qual o bloco tarefa t_2 está associado (ver portas *jobout* dos blocos tarefa, que são usadas para enviar cargas de trabalho). O bloco tarefa t_2 então espera até que a carga de trabalho submetida termine. A carga de trabalho entra na fila de prioridades do bloco recurso $PE1$, e quando $PE1$ termina de executá-la, o bloco tarefa t_2 é notificado (ver portas *jobin* dos blocos tarefa, que são usadas para receber notificações sobre a finalização de uma carga de trabalho). Depois da notificação, o bloco tarefa t_2 imediatamente dispara, isto é, ele decrementa cada contador interno de *tokens*, e notifica cada um dos blocos tarefa/gerador conectado à sua porta $tout$ que um *token* foi enviado. O bloco tarefa t_3, que recebeu o *token*, por sua vez, irá incrementar o respectivo contador interno de *tokens*. Caso o bloco tarefa t_3 fique pronto para disparar (caso todos os contadores internos de *tokens* associados à suas portas de entrada tin_1 e tin_2 sejam maiores que zero), ele irá enviar uma carga de trabalho para $PE2$. Este processo continua até que o critério de parada da simulação seja atingido. A partir dessa descrição, é possível notar que os blocos tarefas/gerador modelam as dependências de dados existentes na aplicação, e os blocos recurso modelam os aspectos temporais resultantes das computações geradas pelos blocos tarefa.

5.2 Tempos de execução e comunicação

Nos modelos de simulação, o tempo de execução de uma carga de trabalho gerada por um bloco tarefa i para um bloco recurso j é dado pela variável aleatória $Exec_{i,j}$, que é igual ao somatório dos tempos das fases de cópia de mensagens, execução, e escrita de mensagens da tarefa que o bloco tarefa i representa:

$$Exec_{i,j} = \underbrace{\sum_{\{z\,:\,(z,i)\in C_d,\, m((z,i))\neq j\}} \mathbb{E}(COPY_{z,i,j,m((z,i))})}_{\text{fase de cópia de mensagens}} + \underbrace{ET_{i,j}}_{\text{fase de execução}} + \underbrace{\sum_{\{z\,:\,(i,z)\in C_d,\, m((i,z))\neq j\}} \mathbb{E}(COPY_{i,z,j,m((z,i))}) + \sum_{\{z\,:\,(i,z)\in C_d,\, m((i,z))= j\}} \mathbb{E}(COPY^{*}_{i,z,j})}_{\text{fase de escrita das mensagens}} \quad (5.1)$$

onde $\mathbb{E}(COPY_{i,z,j,w})$, $\mathbb{E}(COPY^{*}_{i,z,j})$ e $ET_{i,j}$ foram definidas na Subseção 4.3.2, C_d é o conjunto de arcos que representam canais de comunicação do HSDG, e a função $m : C_d \rightarrow V$ retorna onde um dado canal está mapeado (V é o conjunto dos elementos do ATG).

Os termos correspondentes às fases de cópia de mensagens e escrita de mensagens na Equação (5.1) não consideram os atrasos devido à contenção dos elementos de comunicação. Embora isso seja válido para comunicações intraprocessador, pode não ser válido para comunicações interprocessador. Ou seja, caso mais de um elemento de processamento esteja acessando ao mesmo tempo um elemento de comunicação, então o tempo para leitura/escrita de uma mensagem pode ser maior que o tempo verificado, caso não existisse a disputa pelo recurso. A seguir, serão descritas as condições para que a Equação (5.1), que considera que não há disputa por recursos, seja uma aproximação válida do comportamento real.

Como já mencionado, quando dois ou mais elementos acessam ao mesmo tempo um barramento, este trabalho assume que uma política *round-robin* é adotada. Resultados disponíveis

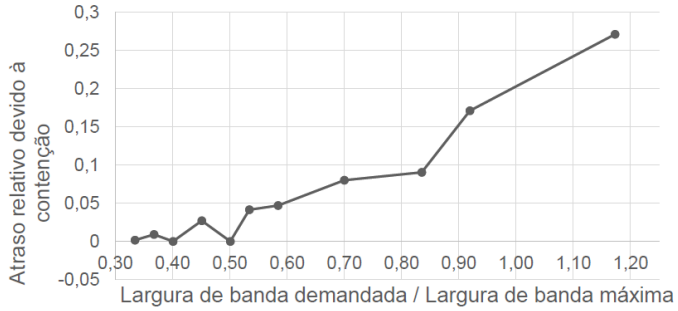


Figura 5.2: Atraso relativo no tempo de execução devido à contenção no barramento.

na literatura, obtidos por simulação de baixo nível (RUGGIERO et al., 2006, 2004), indicam que o tempo de atraso causado pela contenção em barramentos AHB (*Advanced High-performance Bus*) (HOLDINGS, 2006) é relativamente pequeno, caso uma política *round-robin* seja adotada e a largura de banda demandada pelas tarefas seja inferior a 50% da largura de banda máxima suportada pelo barramento. A largura de banda demandada por uma tarefa é calculada como sendo a razão entre quantidade de dados transferida e o tempo de execução da etapa de transferência.

Os resultados obtidos por simulação nos trabalhos da literatura foram confirmados nesta tese através de medições em uma arquitetura real com dois processadores e um barramento AHB conectando os dois processadores. A Figura 5.2 mostra o atraso relativo devido à contenção no barramento, considerando o tempo de execução de dois trechos de código. Cada trecho de código é executado em um processador diferente, e eles foram concebidos especificamente para gerar diferentes níveis de utilização no barramento. A seguinte expressão foi adotada para calcular o atraso relativo:

$$\frac{(Ta_1^{'} + Ta_2^{'}) - (Ta_1 + Ta_2)}{(Ta_1 + Ta_2)},$$

onde Ta_i e $Ta_i^{'}$ representam, respectivamente, o tempo de execução do trecho de código i quando ele é executado isoladamente, e quando ele é executado em paralelo ao outro trecho de código.

Os resultados na Figura 5.2 mostram que, de fato, quando o nível de utilização do barramento é menor ou igual a 50%, o atraso devido à contenção é muito pequeno (atraso médio de 0,8% e atraso máximo de de 2,7%). Dessa forma, a Equação (5.1) representa uma boa aproximação caso a utilização dos barramentos sejam iguais ou menores que 50%. Os algoritmos de exploração propostos levam em consideração essa restrição durante a atividade de mapeamento dos canais de comunicação. A Equação (5.2) é usada pelos algoritmos para estimar qual a largura de banda demandada em um dado barramento b.

$$breq_b = \sum_{\forall p \in P} max\left(\frac{size_{c_1}}{tcopy_{c_1,p}}, \frac{size_{c_2}}{tcopy_{c_2,p}}, \ldots, \frac{size_{c_n}}{tcopy_{c_n,p}}\right) \quad c_1, c_2, \ldots, c_n \in Map(b,p), \qquad (5.2)$$

onde P é o conjunto de processadores alocados, $size_c$ é o tamanho das mensagens do canal de comunicação c, $tcopy_c$ é o tempo médio para o processador p copiar da memória compartilhada para memória local as mensagens do canal de comunicação c, e $c \in Map(b,p)$ indica que o canal c está mapeado no barramento b e a tarefa emissora ou receptora de c está mapeada no processador p.

Neste trabalho, consideramos apenas barramentos AHB (*Advanced High-performance Bus*). Caso o projetista utilize outro tipo de barramento, ele pode aumentar/diminuir o percentual

máximo de utilização suportado pelo barramento. É importante ressaltar que caso a contenção nos barramentos fosse modelada com maior exatidão, um modelo de simulação mais detalhado seria necessário, o que, por outro lado, implicaria em uma análise muito mais lenta das alternativas de projeto. Não obstante, um modelo mais detalhado de barramento é uma possibilidade a ser considerada em trabalhos futuros.

5.3 Blocos básicos propostos

Nesta seção, a especificação formal dos blocos básicos utilizados para modelar os aspectos temporais de um sistema embarcado é apresentada.

5.3.1 Bloco tarefa

A Figura 5.3 exibe a especificação do bloco tarefa, que é o modelo atômico P-DEVS adotado para modelar os atores do HSDG que representam uma tarefa. A estrutura desse bloco básico é apresentada na Figura 5.4. O modelo possui $k+1$ portas de entrada (linha 2 da Figura 5.3), onde k é o número de blocos tarefa/gerador predecessores (recordar a Definição 2.17). A primeira porta ("*jobin*") é utilizada para receber do bloco recurso ao qual o bloco tarefa está associado o sinal indicando o fim da execução de uma carga de trabalho, e as outras portas ("tin_1", ... , "tin_i", ... ,"tin_k") são utilizadas para receber *tokens* dos blocos tarefa/gerador predecessores, ou seja, um evento em uma dessas portas indica que um bloco tarefa/gerador precedente disparou. O modelo possui duas portas de saída (linha 4): a primeira ("*jobout*") é usada para enviar cargas de trabalho, e a segunda ("*tout*") é usada para avisar blocos tarefa/gerador sucessores que um *token* foi gerado. Uma carga de trabalho é representada por uma tupla que possui (linha 3): tempo de execução (*exec_time*), prioridade (*priority*), e um identificador único (*id*). O identificador único é usado para que o bloco recurso possa saber qual bloco tarefa enviou a carga de trabalho e, assim, notificar o bloco tarefa correto quando a carga de trabalho for finalizada.

O estado do modelo é representado por uma tupla (linha 5 da Figura 5.3), que determina: (i) se o bloco tarefa está esperando ou não que o bloco recurso termine de executar a carga de trabalho enviada por ele (*waiting*); (ii) se o bloco recurso terminou ou não de executar a carga de trabalho enviada por ele (*done*); (iii) o estado dos contadores de *tokens* associados às portas de entrada "tin_1", ... , "tin_i", ... , "tin_k" ($channel_1$, ... , $channel_i$, ... , $channel_k$); (iv) se o bloco tarefa está ou não pronto para disparar, isto é, se cada contador de *token* é maior que zero (*ready*); (v) e se o bloco tarefa recebeu ou não um *token* que representa uma mensagem interprocessador (*overhead*). Além de enviar cargas quando estão prontos para disparar, um bloco tarefa envia uma carga de trabalho sempre que ele recebe um token de outro bloco tarefa que não está associado ao mesmo bloco recurso que o seu. Este comportamento representa o recebimento de mensagens interprocessador, em que a tarefa de tratamento de mensagens interprocessador deve ser executada (ver Figura 4.11). A carga de trabalho, neste caso, representa a execução da tarefa de tratamento, e não a tarefa representada pelo bloco tarefa, portanto, quando o bloco recurso terminar de executar esta carga, ele não precisa notificar o bloco tarefa.

O bloco tarefa faz uso das seguintes funções auxiliares: $RandomExecTime()$, que gera um número aleatório obedecendo a distribuição de probabilidade da variável aleatória que representa o tempo de execução da tarefa (Equação 5.1); e $InterProcMsg(p)$, que retorna verdadeiro, caso a porta de entrada p do bloco tarefa seja usada para receber *tokens* que representam mensagens interprocessador, e falso, caso contrário. Além disso, as seguintes constantes são usadas no modelo: *task_priority*, que representa a prioridade das cargas de trabalho geradas pelo bloco tarefa; *max_priority*, que representa a máxima prioridade possível de uma carga

$Task = (X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ tal que $IPorts \in \{$"$jobin$","tin_1", ... ,"tin_i", ... ,"tin_k"$\}$, $X_{jobin} \in \{$ "$done$"$\}$, $X_{tin_i} \in \{$ "$token$"$\}$

$Job = \{exec\_time \in \mathbb{R}^+, priority \in \mathbb{N}, id \in \mathbb{N}\}$

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ tal que $OPorts \in \{$"$jobout$","$tout$"$\}$, $Y_{jobout} \in Job, Y_{tout} \in \{$ "$token$"$\}$

$S = waiting \in \{true, false\}, done \in \{true, false\}, \{channel_1 \in \mathbb{N}, \ldots, channel_i \in \mathbb{N}, \ldots, channel_k \in \mathbb{N}, ready \in \{true, false\}, overhead \in \{true, false\}\}$

$\mathbf{s' = \delta_{ext}(s, e, x^b)}$:
- $s' := s$
- **foreach** x **in** x^b **do**
  - **if** $x.p ==$ "tin_i" **then** // `token` chegou pela porta tin_i
    - $s'.channel_i := s'.channel_i + 1$ // incrementa o contador associado à porta
    - **if** $InterProcMsg(i)$ **then** // se tin_i é uma porta para *tokens* que representam mensagens interprocessador
      - $s'.overhead := true$ // a carga de trabalho que representa o tratamento de mensagens interprocessador deve ser gerada
  - **else if** $x.p ==$ "$jobin$" **then** // uma carga de trabalho foi finalizada
    - $s'.channel_1 := s'.channel_1 - 1, \ldots, s'.channel_k := s'.channel_k - 1$ // decrementa cada contador de *tokens*
    - $s'.done := true$ // um *token* precisa ser enviado para os sucessores
    - $s'.waiting := false$ // o modelo não está aguardando a execução da carga de trabalho
  - **if** $s'.waiting = false \wedge s'.channel_1 > 0 \wedge \ldots \wedge s'.channel_k > 0$ **then** // se o bloco tarefa não está esperando e cada contador de *token* é maior que zero
    - $s'.ready := true$ // uma carga de trabalho precisa ser enviada
    - $s'.waiting := true$ // o bloco tarefa está esperando a execução da carga de trabalho

$\mathbf{y^b = \lambda(s)}$:
- **if** $s.ready$ **then** // cria uma carga de trabalho
  - $y_1.p :=$ "$jobout$", $y_1.v := (RandomExecTime(), task\_priority, task\_id)$
  - $y^b := \{y_1\}$
- **if** $s.overhead$ **then** // cria uma carga de trabalho referente a tarefa de tratamento de mensagens interprocessador
  - $y_2.p :=$ "$jobout$", $y_2.v := (inter\_proc\_overhead, max\_priority, task\_id)$
  - $y^b := y^b \cup \{y_2\}$
- **if** $s.done$ **then** // envia um *token* para os blocos tarefa/gerador sucessores
  - $y_3.p :=$ "$tout$", $y_3.v :=$ "$token$"
  - $y^b := y^b \cup \{y_3\}$

$\sigma = \mathbf{ta(s)}$:
- **if** $s.done \vee s.ready \vee s.overhead$ **then**
  - $\sigma := 0$
- **else**
  - $\sigma := +\infty$

$\mathbf{s' = \delta_{int}(s)}$:
- $s'.done := false$, $s'.ready := false$, $s.overhead := false$

Figura 5.3: Modelo atômico P-DEVS que modela um ator do HSDG que representa uma tarefa (bloco tarefa).

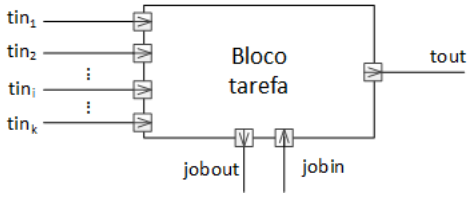


Figura 5.4: Visão estrutural do bloco tarefa.

de trabalho (apenas a carga de trabalho que representa a execução da tarefa de tratamento de mensagens interprocessador possui essa prioridade); *task_id*, que representa o identificador único do bloco tarefa; e *inter_proc_overhead*, que representa o tempo de execução da tarefa que trata mensagens interprocessador (recorde que esse tempo foi informado durante a etapa de caracterização, Subseção 4.3.2). Essas constantes têm seus valores definidos automaticamente pelos algoritmos de exploração durante a inicialização do modelo.

Quando um bloco tarefa recebe um evento de entrada (linha 7 da Figura 5.3), isso significa que duas situações podem ter acontecido: (i) um *token* foi recebido (linha 10), ou (ii) o bloco recurso ao qual o bloco tarefa está associado terminou a execução de uma carga de trabalho enviada por ele (linha 14). No caso da primeira situação, o contador de *tokens* associado à porta de entrada é incrementado em uma unidade (linha 11). Se o *token* recebido representar uma mensagem interprocessador (linha 12), então uma carga de trabalho que representa a execução da tarefa de tratamento de mensagens interprocessador deve ser executada (linha 13). Caso o evento de entrada represente a situação em que o bloco recurso tenha terminado a execução de uma carga de trabalho (linha 14), então cada contador de *tokens* do bloco tarefa é decrementado em uma unidade (linha 15), e o bloco tarefa se prepara para enviar um *token* para os blocos tarefa/gerador sucessores (linhas 16 e 17). Depois que o evento recebido é tratado, é preciso verificar se cada contador de *tokens* é maior que zero, e se o bloco tarefa não está esperando uma outra carga de trabalho terminar de executar (linha 18). Caso o resultado da verificação seja verdadeiro, então isso significa que o bloco tarefa está pronto para disparar, isto é, enviar uma carga de trabalho ao bloco recurso ao qual o bloco tarefa está associado (linhas 19 e 20).

Quando a função de saída do modelo é chamada (linha 22 da Figura 5.3), ela testa as seguintes condições: (i) se o bloco tarefa está pronto para disparar (linha 23), (ii) se uma carga de trabalho representando a tarefa que trata mensagens interprocessador deve ser enviada (linha 26), e (iii) se o bloco recurso ao qual o bloco tarefa está associado terminou de executar uma carga de trabalho enviada por ele (linha 29). Caso a primeira condição seja verdadeira (linha 23), uma nova carga de trabalho é gerada e enviada para a porta "*jobout*" (linhas 24 e 25). O tempo de execução, a prioridade, e o identificador único da carga de trabalho são dados, respectivamente, pela função *RandomExecTime*(), constante *task_priority*, e constante *task_id*. Caso a segunda condição seja verdadeira (linha 26), uma carga de trabalho que representa o a execução da tarefa de tratamento de mensagens interprocessador é gerada (linhas 27 e 28) e também enviada pela porta "*jobout*". Essa tarefa tem prioridade máxima, *max_priority*, e seu tempo de execução é definido pela constante *inter_proc_overhead*. Caso a terceira condição seja verdadeira (linha 29), a porta "*tout*" é usada para enviar um *token* para todos os blocos tarefa/gerador que sucedem o bloco tarefa (linhas 30 e 31).

A função de avanço do tempo testa se o bloco tarefa está pronto para disparar, se uma carga de trabalho representando a tarefa de tratamento de mensagens interprocessador deve ser enviada, ou se o bloco recurso ao qual o bloco tarefa está associado finalizou a execução de uma carga de trabalho (linhas 33 e 34 da Figura 5.3). Caso alguma das condições seja verdadeira, então a função de saída é imediatamente chamada (linha 35). Caso contrário, o bloco tarefa fica

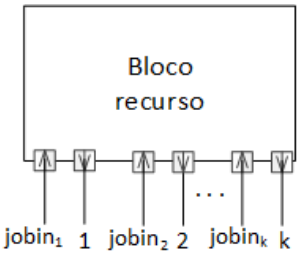


Figura 5.5: Visão estrutural de um bloco recurso.

em estado ocioso, aguardando algum evento externo chegar (linhas 36 e 37).

5.3.2 Bloco recurso

Dois modelos atômicos foram desenvolvidos para representar um processador e o sistema operacional que executa nele: (i) bloco recurso que suporta preempções de tarefas, e (ii) bloco recurso que não suporta preempções de tarefas. A Figura 5.5 apresenta a estrutura desses blocos, e a Figura 5.6 mostra a especificação do bloco recurso que suporta preempções de tarefas. Este bloco básico é usado para modelar um processador com um sistema operacional em que preempções estão autorizadas. Devido às similaridades desse modelo com o modelo em que preempções não são suportadas, o último é apresentado no Apêndice A. O bloco recurso possui uma porta de entrada e uma de saída para cada um dos k blocos tarefa associados ao bloco recurso (linhas 3 e 4 da Figura 5.6): "$jobin_1$", ... , "$jobin_k$" são as portas de entrada, e "$jobout_1$", ... , "$jobout_k$" são as portas de saída. As portas de entrada e saída são usadas, respectivamente, para receber novas cargas de trabalho e notificar os blocos tarefa sobre a finalização de sua respectiva carga de trabalho. O estado do modelo é representado por uma tupla que determina (linha 5): (i) qual carga de trabalho está sendo executada no momento (*current job*), (ii) o estado da fila de prioridades com as cargas de trabalho prontas para executar (*priorityqueue*), (iii) o tempo restante para uma mudança de contexto (*so_overhead*), ou seja, o tempo restante para o sistema operacional chavear de uma tarefa para outra.

O modelo faz uso das seguintes funções auxiliares: $Insert(l,w)$, que insere na fila de prioridades l a carga de trabalho w; $Top(l)$, que retorna a carga de trabalho de maior prioridade na fila l; $Pop(l)$, que retorna e remove da fila de prioridades l a carga de trabalho de maior prioridade; e $max(a,b)$, que retorna o maior argumento entre a e b. As seguintes constantes são usadas no modelo: *context_switch_delay*, que representa o *overhead* de uma mudança de contexto pelo sistema operacional (recordar que esse tempo foi informado durante a etapa de caracterização); e *max_priority*, que representa a prioridade máxima de uma carga de trabalho. Conforme visto na subseção anterior, apenas a carga de trabalho que representa a tarefa de tratamento de interrupções interprocessador pode receber esta prioridade.

Sempre que o bloco recurso recebe um evento de entrada (linha 7), isso significa que novas cargas de trabalho chegaram ao recurso. Quando isso acontece, as novas cargas de trabalho são inseridas na fila de prioridades (linhas 9 e 10 da Figura 5.6). Caso o bloco recurso não esteja ocioso no momento da chegada (linha 11), o tempo restante para finalizar a carga de trabalho em execução, assim como o tempo restante para uma mudança de contexto são atualizados (linhas 11-14). Em seguida, verifica-se se as novas cargas de trabalho provocarão uma mudança de contexto, isto é, se chegou uma carga de trabalho com maior prioridade que a carga sendo atualmente executada (linha 15). Caso se verifique que haverá mudança de contexto, então o tempo para mudança de contexto é reiniciado (linha 16). Finalmente, a carga de trabalho de

$Resource_p = (X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$
$Job = \{exec\_time \in \mathbb{R}^+, priority \in \mathbb{N}, id \in \mathbb{N}\}$
$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ tal que $IPorts \in \{$"$jobin_1$", ... ,"$jobin_i$" , ... ,"$jobin_k$"$\}$, $X_{jobin_i} \in Job$
$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ tal que $OPorts \in \{$"1", ... , i , ... ,"k"$\}$, $Y_i \in \{$ "$done$"$\}$
$S = \{current\_job \in Job \cup \{$"$nil$"$\}, priorityqueue \in \{job \in Job\}*, so\_overhead \in \mathbb{R}^+\}$

$\mathbf{s' = \delta_{ext}(s, e, x^b)}$:
    $s' := s$
    foreach x **in** x^b **do**
        $Insert(s'.priorityqueue, x.v)$ // insere as novas cargas de trabalho
    if $s.current\ job \neq$"nil" **then** // se o bloco recurso não está ocioso
        $s'.current\ job.exec\_time := s'.current\ job.exec\_time - max(e - s'.so\_overhead, 0)$
        // atualiza o tempo restante para terminar a carga de trabalho atual
        $s'.so\_overhead := max(e - s'.so\_overhead, 0)$ // atualiza o tempo restante para finalizar uma mudança de contexto
        $s'.priorityqueue := Insert(s'.priorityqueue, s'.current\ job)$
    if $s.current\ job \neq Top(s'.priorityqueue)$ **then** // verifica se haverá mudança de contexto
        $s'.so\_overhead := context\_switch\_delay$ // acrescenta o tempo para mudança de contexto
    $s'.current\ job := Pop(s'.priorityqueue)$ // seleciona e remove a carga de trabalho com maior prioridade na fila

$\sigma = \mathbf{ta(s)}$:
    if $s.current\ job \neq$"nil" **then**
        $\sigma := s.current\ job.exec\_time + s.so\_overhead$
    else
        $\sigma := +\infty$ // recurso ocioso

$\mathbf{y^b = \lambda(s)}$:
    if $current\ job.id \neq max\_priority$ **then** // verifica se a carga de trabalho finalizada representa a tarefa de tratamento de mensagens interprocessador
        $y.p := current\ job.id,\ y.v :=$"$done$", $y^b := \{y\}$ // avisa o bloco tarefa que a carga de trabalho enviada terminou

$\mathbf{s' = \delta_{int}(s)}$:
    $s'.current\ job :=$"nil", $s'.priorityqueue := s.priorityqueue$
    if $s'.priorityqueue \neq ()$ **then** // se a fila não estiver vazia
        $s'.current\ job := Pop(s'.priorityqueue)$ // inicia a execução da próxima carga de trabalho
        $s'.so\_overhead := context\_switch\_delay$ // acrescenta o tempo para mudança de contexto

$\mathbf{s' = \delta_{con}(s, x^b)}$:
    $s' := \delta_{ext}(\delta_{int}(s), 0, x^b)$ // a carga de trabalho atual precisa ser finalizada primeiro

Figura 5.6: Modelo atômico P-DEVS que representa um processador com sistema operacional em que preempções estão autorizadas (bloco recurso com preempção).

maior prioridade, considerando as cargas antigas e as novas, é colocada para executar (linha 17).

A função de avanço do tempo verifica se o bloco recurso precisa executar uma carga de trabalho (linhas 19 e 20 da Figura 5.6). Caso precise, o tempo de execução é definido como sendo o tempo restante para executar a carga de trabalho mais o tempo restante para mudança de contexto (linha 21). Caso contrário, o bloco recurso fica ocioso, aguardando algum evento externo chegar (linhas 22 e 23).

A função de saída (linha 25 da Figura 5.6) notifica um bloco tarefa que sua respectiva carga de trabalho enviada foi finalizada. Caso a carga de trabalho finalizada tenha sido um serviço de tratamento de mensagens interprocessador, então o bloco tarefa não precisa ser avisado (linha 26). A verificação é feita checando se a carga de trabalho tem prioridade máxima, uma vez que uma carga de trabalho comum não pode assumir essa prioridade. Como o recurso possui uma porta de saída para cada bloco tarefa, a porta correta é selecionada a partir do identificador da carga de trabalho (linha 27).

Quando a função de transição interna é chamada (linha 29 da Figura 5.6), isso significa que o bloco recurso finalizou a execução de uma carga de trabalho. Nesse caso, a função verifica se a fila de cargas de trabalho está vazia (linha 31), caso não esteja, o bloco recurso é configurado para iniciar a execução da próxima carga de trabalho (linhas 32 e 33).

5.3.3 Bloco gerador

A especificação do bloco gerador, que representa os atores do HSDG que modelam processos externos, é mostrada na Figura 5.7. A Figura 5.8 apresenta a estrutura desse bloco. O bloco gerador é bastante similar ao bloco tarefa, no sentido de que ele só pode disparar caso todos os seus contadores de *token* sejam maiores que zero. No entanto, o bloco gerador não é associado a nenhum bloco recurso, uma vez que ele não representa uma tarefa.

O modelo possui k portas de entrada ("tin_1", ... ,"tin_i", ... ,"tin_k"), que são utilizadas para receber *tokens* dos k blocos tarefa que o precedem (linha 2). Ele possui apenas uma porta de saída ("$tout$"), usada para enviar *tokens* aos blocos tarefa que o sucedem (linha 3). O estado do gerador é representado por uma tupla, que define (linha 4): (i) quantos *tokens* existem em cada contador interno de *tokens* ($channel_1$, ... , $channel_i$, ... , $channel_k$), (ii) o tempo restante para um disparo do bloco gerador ($timeleft$), e (iii) se o bloco gerador está pronto para disparar ou não ($ready$). O modelo faz uso da função auxiliar $InterArrivalTime()$, que gera um número aleatório obedecendo a distribuição de probabilidade da variável aleatória IT_i associada ao ator que o bloco gerador representa (recordar a definição desta variável aleatória na Seção 4.3).

Sempre que a função de evento externo é chamada (linha 6), isso significa que algum bloco tarefa predecessor enviou um *token*. Isto faz com que o contador de *tokens* correspondente seja incrementado (linhas 8-10). Caso o bloco gerador já esteja pronto para disparar, o tempo restante para o disparo é atualizado (linhas 11 e 12). Caso o bloco gerador não esteja pronto para disparar e todos os seus contadores de *token* sejam maiores que zero, ele é configurado para um novo disparo (linhas 13-15). A função $InterArrivalTime()$ é usada para definir o tempo restante para o disparo. Caso não haja *tokens* suficientes, o gerador é colocado em modo ocioso para que ele aguarde até que um novo evento aconteça em uma de suas portas de entrada (linhas 16 e 17).

Quando a função de transição interna é chamada (linha 19), isso implica que o bloco gerador acabou de disparar. Assim, cada contador de *token* é decrementado (linha 21). Em seguida, a função verifica se existem *tokens* suficientes para um novo disparo (linha 22). Caso existam, o bloco gerador é preparado para um novo disparo (linha 23), caso contrário, ele é colocado em modo ocioso (linhas 24 e 25).

Generator $= (\mathbf{X}, \mathbf{Y}, \mathbf{S}, \delta_{\mathbf{int}}, \delta_{\mathbf{ext}}, \delta_{\mathbf{con}}, \lambda, \mathbf{ta})$
$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ tal que $IPorts \in \{$"tin_1", ... ,"tin_i", ... ,"tin_k"$\}$, $X_{tin_i} \in \{$ "$token$"$\}$
$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ tal que $OPorts \in \{$"$tout$"$\}$, $Y_{tout} \in \{$ "$token$"$\}$
$S = \{channel_1 \in \mathbb{N}, \ldots, channel_i \in \mathbb{N}, \ldots, channel_k \in \mathbb{N}, timeleft \in \mathbb{R}^+, ready \in \{true, false\}\}$

$\mathbf{s'} = \delta_{\mathbf{ext}}(\mathbf{s}, \mathbf{e}, \mathbf{x^b})$:
  $s' := s$
  foreach x **in** x^b **do**
    if $x.p ==$ "tin_i" **then** // token chegou ao canal tin_i
      $s'.channel_i := s'.channel_i + 1$ // incrementa o contador de *tokens* correspondente
  if $s'.ready$ **then** // se o bloco gerador já estiver pronto para disparar
    $s'.timeleft := s'.timeleft - e$ // atualiza o tempo restante para o disparo
  else if $s'.ready = false \wedge s'.channel_1 > 0 \wedge \ldots \wedge s'.channel_k > 0$ **then** // se o bloco gerador não estiver pronto e todos os seus contadores de *tokens* forem maiores que zero
    $s'.timeleft := InterArrivalTime()$ // inicia o tempo restante para um novo disparo
    $s'.ready := true$ // pronto para disparar
  else
    $s'.timeleft := +\infty$ // deixa o bloco gerador em modo ocioso aguardando um novo evento

$\mathbf{s'} = \delta_{\mathbf{int}}(\mathbf{s})$:
  $s' := s$
  $s'.channel_1 := s'.channel_1 - 1, \ldots, s'.channel_k := s'.channel_k - 1$ // decrementa cada contador de *tokens*
  if $s'.channel_1 > 0 \wedge \ldots \wedge s'.channel_k > 0$ **then** // se todos os contadores são maiores que zero
    $s'.timeleft := InterArrivalTime()$ // inicia o tempo restante para um novo disparo
  else
    $s'.timeleft := +\infty$ // deixa o bloco gerador em modo ocioso aguardando um novo evento

$\mathbf{y^b} = \lambda(\mathbf{s})$:
  $y^b := \{($"out","$token$"$)\}$ // envia um *token* para os blocos tarefa que sucedem o bloco gerador

$\sigma = \mathbf{ta}(\mathbf{s})$:
  $\sigma := s.timeleft$

Figura 5.7: Modelo atômico P-DEVS que modela um ator do HSDG que representa um processo do ambiente externo (bloco gerador).

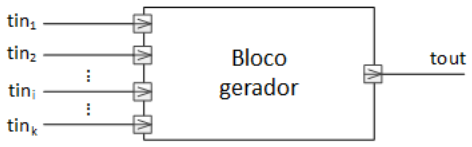


Figura 5.8: Visão estrutural do bloco gerador.

Figura 5.9: Visão estrutural do bloco monitor.

1 **Monitor** $= (\mathbf{X}, \mathbf{Y}, \mathbf{S}, \delta_{\mathbf{int}}, \delta_{\mathbf{ext}}, \delta_{\mathbf{con}}, \lambda, \mathbf{ta})$
2 $X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ tal que $IPorts \in \{$"*begin*","*end*" $\}$, $X_{begin} \in \{$ "*token*"$\}$, $X_{end} \in \{$ "*token*"$\}$
3 $Y = \{\}$
4 $S = \{clock \in \mathbb{R}^+, starttimes \in \{\mathbb{R}^+\}*\}$
5
6 $\mathbf{s}' = \delta_{\mathbf{ext}}(\mathbf{s}, \mathbf{e}, \mathbf{x^b})$:
7 **foreach** x **in** x^b **do**
8 $s'.starttimes := s.starttimes$
9 $s'.clock := s.clock + e$ `// atualiza o clock interno`
10 **if** $x.p == begin$ **then**
11 $Insert(s'.starttimes, s'.clock)$ `// inicia o monitoramento de um token`
12 **else**
13 **if** $DEADLINE < s'.clock - Front(s'.starttimes)$ **then** `// se o deadline foi violado`
14 $Akaroa.Observation(1)$
15 **else**
16 $Akaroa.Observation(0)$
17 $Pop(s'.starttimes)$
18
19 $\sigma = \mathbf{ta}(\mathbf{s})$:
20 $\sigma := +\infty$

Figura 5.10: Modelo atômico P-DEVS usado para verificar violações de *deadlines* (bloco monitor).

5.3.4 Bloco monitor

O bloco monitor é o modelo usado para verificar violações de *deadlines* do sistema. A Figura 5.9 exibe a estrutura desse bloco e a Figura 5.10 apresenta a sua especificação. Um *deadline* $d(c_j, c_l)$ (recordar a definição de *deadline* na Seção 4.3) pode ser interpretado como o tempo máximo que um *token* pode levar para sair do arco c_j até chegar no arco c_l. Caso o *token* demore mais que o *deadline* para sair de c_j até c_l, então o *deadline* foi violado. Dessa forma, assim que o *token* atinge o arco c_j a verificação deve iniciar, e quando ele atingir o arco c_l, a verificação deve terminar. O modelo da Figura 5.10 tem apenas duas portas de entrada, "*start*" e "*end*", que são utilizadas, respectivamente, para marcar os tempos de início e fim de uma verificação (linha 2). O modelo não tem nenhuma porta de saída (linha 3). Ele possui duas variáveis de estado (linha 4): a primeira armazena um *clock* interno (*clock*), e a segunda modela uma fila que armazena os valores do *clock* interno no início das verificações (*starttimes*).

Esse modelo faz uso das seguintes funções auxiliares: (i) $Insert(l,x)$, que insere o valor x no final da fila l; (ii) $Pop(l)$, que remove o primeiro elemento da fila l; $Front(l)$, que retorna o primeiro elemento da fila l, e (iv) $Akaroa.Observation(x)$, que envia a observação x para o *framework* Akaroa (EWING; PAWLIKOWSKI; MCNICKLE, 1999). Esse *framework* é utilizado para realizar análises estatísticas dos resultados da simulação, e será explicado em maiores detalhes na próxima seção. O modelo usa a constante *DEADLINE*, que define qual o

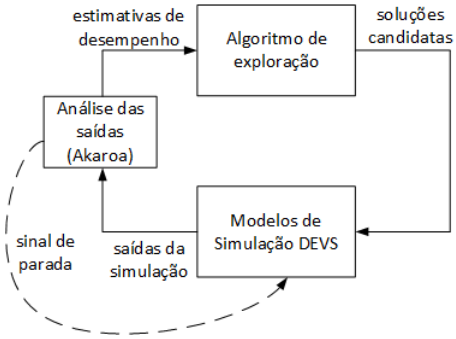


Figura 5.11: Integração entre os modelos de simulação, algoritmos de exploração e o *framework* Akaroa.

tempo máximo entre a saída e chegada do *token* nos arcos do modelo HSDG que estão sendo monitorados.

Sempre que um evento de entrada ocorre, isso indica que um *token* foi enviado ao bloco monitor. Quando um *token* é recebido pelo bloco monitor, isso pode representar o início ou o fim de uma verificação. Depois que o evento ocorre, a função de transição externa atualiza o *clock* interno do modelo (linhas 6-9). Em seguida, ela verifica se o evento indica o início ou o fim de uma verificação (linha 10). Caso o evento indique o início, então o valor atual do *clock* é inserido no final da fila de *clocks*, *starttimes* (linha 11). Caso contrário, uma subtração é feita entre o valor atual do *clock* e o primeiro valor da fila de *clocks* (linha 13). O resultado da subtração indica o tempo que o *token* levou entre a saída e a chegada nos arcos sendo monitorados. Caso esse valor seja maior que a constante *DEADLINE*, então isso significa que um *deadline* foi violado. Caso o *deadline* tenha sido violado, a função envia uma observação com valor 1 para o *framework* Akaroa (linha 14), caso contrário, uma observação com valor 0 é enviada (linha 16). Por fim, caso o evento recebido seja o fim de uma verificação, o primeiro valor da fila de *clocks* é removido (linha 17).

5.4 Avaliação dos modelos

A implementação das especificações e a composição dos blocos básicos são feitas utilizando a biblioteca adevs (NUTARO, 2012). Os modelos são avaliados por meio de simulação estacionária. Como mencionado na Subseção 5.3.4, o *framework* Akaroa (EWING; PAWLIKOWSKI; MCNICKLE, 1999) é usado para analisar as saídas da simulação. A Figura 5.11 exibe com maiores detalhes como o *framework* Akaroa, os modelos de simulação, e os algoritmos de exploração são integrados. Os métodos de análise estatística disponíveis em Akaroa são adotados para determinar quando a simulação deve parar (ver arco *sinal de parada* na Figura 5.11). A seguir, mostramos como probabilidades de violação de *deadlines* são calculadas.

Seja $d(c_k, c_l)$ um *deadline*, e Y_1, Y_2, ... variáveis aleatórias, tal que:

$$Y_j = \begin{cases} 1, & \text{se } \textit{deadline} \text{ foi violado na j-ésima verificação} \\ 0, & \text{caso contrário.} \end{cases} \tag{5.3}$$

Então, a probabilidade θ de se violar o *deadline* $d(c_k, c_l)$ é dada pela média a seguir:

$$\theta = \mathbb{E}[Y_j] \tag{5.4}$$

Devido à natureza da simulação estocástica, probabilidades de violação de *deadlines* não podem ser avaliadas de forma exata: elas só podem ser *estimadas*. Um estimador não enviesado para a probabilidade de violação de *deadline* θ é dado pela Equação (5.5), onde n é o número de amostras (observações) coletadas da saída da simulação (BOLCH et al., 2006):

$$\widehat{\Theta} = \lim_{n\to\infty} \frac{1}{n} \sum_{j=1}^{n} Y_j \tag{5.5}$$

No entanto, na prática, a simulação precisa ser parada depois de um certo número de amostras, n, terem sido coletadas. Esta restrição e a natureza estocástica do modelo de simulação fazem com que o estimador $\widehat{\Theta}$ seja por si só uma variável aleatória, e sua precisão vai depender, além de outros fatores, da sua variância. Portanto, é preciso levar em consideração essa incerteza. A análise estatística realizada pelo *framework* Akaroa estima esta variância (o método conhecido como *Spectral Analysis* (PAWLIKOWSKI, 1990) foi adotado) e também determina quantas amostras devem ser coletadas para que a precisão desejada seja obtida. Akaroa determina a quantidade necessária de amostras por meio de uma abordagem sequencial em que a simulação só é parada quando precisão especificada para as estimativas é satisfeita, considerando um determinado nível de confiança. Uma dificuldade adicional é que a fase transiente da simulação pode deixar as estimativas enviesadas (PAWLIKOWSKI; JEONG; LEE, 2002). Dessa forma, os métodos de Akaroa também foram usados para automaticamente detectar e remover as observações do período transiente da simulação.

5.5 Considerações finais

Esse capítulo apresentou como violações de *deadlines* são estimadas pelo método proposto. Durante a exploração do espaço de projeto, modelos de simulação são automaticamente criados por meio da composição de blocos básicos de simulação. Os blocos básicos, assim como sua composição, são definidas através do formalismo *Parallel* DEVS. Os modelos são avaliados por simulação estacionária e o *framework* Akaroa é adotado para eficientemente analisar as saídas da simulação. Os modelos são livres de detalhes funcionais da arquitetura, o que permite avaliar muitas opções de projeto em um curto espaço de tempo.

6 Algoritmos de exploração

Este capítulo apresenta os dois algoritmos multiobjetivo propostos para explorar o espaço de projeto de sistemas embarcados de tempo-real não críticos. Dada a especificação do problema (arquivo XML), gerada depois das atividades de especificação e caracterização do método proposto, os algoritmos de exploração automaticamente tentam otimizar a alocação, mapeamento e escalonamento do projeto. A otimização é feita considerando simultaneamente as seguintes medidas: probabilidades de violação de *deadlines*, potência consumida e custo monetário. Os algoritmos retornam como saída um subconjunto representativo do conjunto ótimo de Pareto (ou uma aproximação dele), considerando as medidas mencionadas.

Dentre as abordagens para atacar problemas de otimização multiobjetivo envolvendo simulação, os algoritmos genéticos (ou evolucionários) são uma das mais populares (JOINES et al., 2002; DING; BENYOUCEF; XIE, 2006; ESKANDARI; GEIGER, 2009; FU, 2002; GUTJAHR, 2011). Esse sucesso pode ser creditado principalmente a dois fatores (DEB, 2008; COELLO; LAMONT, 2007): (i) a robustez que eles têm para escapar de regiões de mínimos locais e se adaptar às incertezas da simulação, e (ii) sua abordagem baseada em populações, o que permite que eles procurem múltiplas soluções do conjunto ótimo de Pareto em uma única execução.

O restante do capítulo está organizado da seguinte maneira: na Seção 6.1, o problema atacado pelos algoritmos propostos é formalmente definido. A Seção 6.2 apresenta o primeiro algoritmo proposto, denominado de MODSES, e a Seção 6.3 apresenta o segundo algoritmo proposto, chamado de C-MODSES. Por fim, a Seção 6.4 conclui esse capítulo.

6.1 Formulação do problema de otimização

A exploração do espaço de projeto pode ser estabelecida como um problema de otimização. Nesta seção, o problema de otimização tratado pelos algoritmos propostos é formalmente definido.

Considere o conjunto de elementos V e o conjunto de conexões entre esses elementos E do ATG $G_{atg} = (V, E)$, além disso, considere o conjunto de tarefas T e o conjunto de canais de comunicação C_d do HSDG $G_{hsdg} = (A, C, s_0)$, tal que $T \subseteq A$, $C_d \subseteq C$. Então, cada vértice $v \in V$ do ATG representa um elemento potencialmente alocável. Seja uma **alocação** o vetor $\Gamma = (q_v)$, $q_v \in \{0,1\}$, $v \in V$, tal que

$$q_v = \begin{cases} 1, & \text{se o elemento representado por } v \text{ está alocado,} \\ 0, & \text{caso contrário.} \end{cases}$$

Além disso, seja um **mapeamento** a matriz $\Psi = (h_{v,\tau})$, $h_{v,\tau} \in \{0,1\}$, $v \in V$, $\tau \in T \cup C_d$, tal que

$$h_{v,\tau} = \begin{cases} 1, & \text{se a tarefa ou canal de comunicação representado por } \tau \text{ está mapeada} \\ & \text{no elemento representado por } v. \\ 0, & \text{caso contrário.} \end{cases}$$

Seja uma **atribuição de prioridades** o vetor $\Omega = (o_t)$, $o_t \in \mathbb{N}$, $t \in T$, tal que o_t é a prioridade da tarefa representada por t. Por último, seja uma **atribuição de frequência** o valor $\Lambda \in \mathbb{R}^+$, tal que Λ é a frequência de operação da arquitetura. Então, o problema de otimização tratado pelos algoritmos propostos pode ser formulado como

$$\begin{aligned} &\text{minimize} \quad F(x) = (f_1(x), f_2(x), f_3(x)) \\ &\text{sujeito a} \quad x \in S, \end{aligned} \tag{6.1}$$

onde S representa o conjunto de soluções viáveis, e as funções $f_i : S \to \mathbb{R}^+$, $i = 1, \ldots, 3$ representam, respectivamente, os seguintes objetivos a serem otimizados: custo monetário, potência consumida, e probabilidades de violação de *deadlines*. A Seção 4.3 detalhou como custo monetário e potência consumida são calculados, e o Capítulo 5 apresentou como probabilidades de violação de *deadlines* são estimadas. Caso exista mais de um *deadline* a ser atendido no sistema, então $f_3(x)$ é definido como o somatório das probabilidades de violação de *deadlines*, isto é, $f_3(x) = \theta_1(x) + \ldots + \theta_k(x)$, onde k é o número de *deadlines* do sistema e θ_i foi definida na Equação (5.4).

Uma solução $x = (\Gamma, \Psi, \Omega, \Lambda)$ é viável se as seguintes restrições são satisfeitas:

- Toda tarefa ou canal de comunicação representado por τ só pode ser mapeado no elemento representado por v, caso o último esteja alocado,

$$h_{v,\tau} \leq q_v, \ \ \forall v \in V, \forall \tau \in T \cup C_d. \tag{6.2}$$

- Toda tarefa ou canal de comunicação representado por τ deve estar mapeado em exatamente um elemento,

$$\sum_{v \in V} h_{v,\tau} = 1, \ \ \forall \tau \in T \cup C_d. \tag{6.3}$$

- Se duas tarefas representadas por t_i e t_j estão mapeadas no mesmo elemento, então o canal de comunicação entre as duas tarefas ($c = (t_i, t_j)$) também deve ser mapeado no mesmo elemento,

$$h_{v,t_i} + h_{v,t_j} \leq 1 + h_{v,c}, \ \ \forall c = (t_i, t_j) \in C_d, \forall v \in V \tag{6.4}$$

- Se as tarefas representadas por t_i e t_j estão mapeadas em elementos diferentes, então o canal de comunicação entre as duas tarefas ($c = (t_i, t_j)$) deve ser mapeado em um elemento de comunicação que esteja conectado aos processadores das duas tarefas,

$$h_{v_y,t_i} + h_{v_w,t_j} \leq 1 + \sum_{v_\kappa \in N_{y,w}} h_{v_\kappa,c}, \ \ \forall c = (t_i,t_j) \in C_d, \forall v_y, v_w \in V, v_y \neq v_w \tag{6.5}$$

onde $N_{y,w} = \{v_z \mid (y,z),(z,w) \in E\}$.

- Uma tarefa t só pode ser mapeada no elemento v, caso, durante a etapa de caracterização, as informações de que a variável aleatória $Exec_{t,v}$ depende tenham sido fornecidas. $Exec_{t,v}$ representa o tempo de execução de t em v (recorde a definição desta variável aleatória na Equação (5.1)),

$$h_{v,t} \leq \chi(t,v), \ \ \forall v \in V, \forall t \in T, \tag{6.6}$$

onde $\chi(t,v) \in \{0,1\}$, tal que $\chi(t,v) = 1$, caso as informações de que $Exec_{t,v}$ depende tenham sido fornecidas, e $\chi(t,v) = 0$, caso contrário. Por exemplo, caso a distribuição de probabilidade da variável aleatória $ET_{t,v}$, que representa o tempo da fase de execução da tarefa t no elemento v, não tenha sido fornecida pelo projetista, então t não pode ser mapeada em v. Essa restrição pode ser usada pelo projetista para garantir, por exemplo, que uma tarefa não seja mapeada em um elemento de comunicação.

A formulação apresentada consiste na versão genérica do problema. No entanto, dependendo da especificidade do problema em mãos, algumas variações são permitidas pelo método proposto: (i) ao invés de otimizar todas as funções objetivo apresentadas na Equação (6.1), o projetista pode escolher otimizar qualquer subconjunto dessas funções; (ii) é possível definir valores máximos aceitáveis para cada função objetivo, neste caso, uma solução só será viável caso os valores de suas funções objetivos sejam menores que os especificados; (iii) é possível definir restrições de mapeamento que forçam que as tarefas (ou canais de comunicação) de um dado grupo de tarefas (ou grupo de canais de comunicação) seja mapeado em um mesmo elemento; (iv) é possível também adicionar restrições na atribuição de prioridades para garantir que todas as tarefas de um dado grupo de tarefas tenham a mesma prioridade, embora isso não seja desejável no caso geral, uma vez que aumenta o nível de não determinismo durante a execução do sistema. Dessa forma, caso a atribuição de prioridades não seja explicitamente restringida, os algoritmos propostos evitam que duas ou mais tarefas tenham a mesma prioridade.

É muito provável que o problema formulado acima seja intratável, uma vez que ele é composto por problemas NP-difíceis[1] (ver a complexidade dos problemas clássicos de mapeamento em múltiplos processadores (GAREY; JOHNSON, 1979) e de atribuição de prioridades a tarefas (LEUNG, 1989)). Dessa forma, dois algoritmos baseados na teoria dos algoritmos genéticos foram propostos para resolvê-lo.

6.2 Algoritmo MODSES

Essa seção apresenta o primeiro algoritmo proposto: MODSES (***M**ulti-**O**bjective **D**esign **S**pace **E**xploration of Soft Real-Time Embedded **S**ystems*) (NOGUEIRA et al., 2013). MODSES contém todos os elementos básicos dos algoritmos genéticos (Subseção 2.2.3). Ele também conta

[1] Um problema é NP-difícil se, e somente se, qualquer problema NP-completo pode ser reduzido a ele em tempo polinomial (LEISERSON et al., 2001). Portanto, podemos dizer que um problema NP-difícil é no mínimo tão difícil quanto um problema NP-completo.

com heurísticas de inicialização, que são fundamentais para o seu sucesso e foram desenvolvidas para rapidamente convergir para soluções de boa qualidade sem que muitas avaliações de soluções sejam necessárias. MODSES se baseia no mecanismo de seleção proposto pelo algoritmo SPGA (ESKANDARI; GEIGER, 2009) para lidar com as incertezas nas estimativas geradas pelas simulações. As próximas subseções descrevem os blocos principais de MODSES e, por último, uma visão geral de como esses blocos são integrados é apresentada.

6.2.1 Codificação

Em MODSES, uma solução é codificada por um valor $freq \in \{1,...,w\}$, que representa a frequência de operação da arquitetura (w é o número de frequências que podem ser selecionadas), e por três *strings*: *string* de alocação, *string* de mapeamento e *string* de prioridades. Considere o conjunto V do ATG $G_{atg} = (V,E)$, que representa os elementos da plataforma. Considere também os conjuntos T e C_d do HSDG $G_{hsdg} = (A,C)$ representando, respectivamente, o conjunto de tarefas e de canais de comunicação da aplicação, tal que $T \subseteq A$, $C_d \subseteq C$. Então, o *string* de alocação, a, é um *string* binário com $|V|$ bits, tal $a[i] = 0$, se o elemento i está alocado, e $a[i] = 1$, caso contrário. Os *strings* de mapeamento, m, e prioridade, p, são ambos *strings* de inteiros com tamanhos iguais a, respectivamente, $|T| + |C_d|$ e $|T|$. $m[i] = w$ indica que a tarefa ou canal de comunicação representada pelo índice i está mapeada no elemento $w \in \{1, \ldots, |V|\}$, e $p[i] = j$ indica que a tarefa i possui prioridade $j \in \{1, \ldots, |T|\}$. A Figura 6.1a mostra um exemplo de cada um desses *strings*. Em particular, essa figura exibe como o mapeamento da Figura 6.2 é codificado. Existe uma relação entre os valores no *string* de mapeamento e os índices do *string* de alocação. O exemplo de *string* de mapeamento indica que o canal de comunicação c_{12} está mapeado em *SharedBus*, uma vez que o valor 2 corresponde ao índice de *SharedBus* no *string* de alocação. O *string* de prioridades na Figura 6.1a indica que t_1 e t_2 possuem prioridades iguais a 1 e 2, respectivamente.

6.2.2 Operadores de recombinação

Diferentes operadores de mutação e *crossover* foram desenvolvidos para os elementos que codificam uma solução. O operador de mutação do *string* de prioridades seleciona dois índices e inverte suas prioridades (Figura 2.9b). O operador de mutação do código que representa a frequência de operação da arquitetura *freq* incrementa ou decrementa em uma unidade o valor de *freq*. O operador de mutação do *string* de mapeamento move uma tarefa (ou canal de comunicação) selecionada aleatoriamente para outro elemento capaz de recebê-la. Antes de aplicar esse operador, MODSES estima a utilização média de cada processador: processadores que estão sobrecarregados (utilização maior que 1) não são considerados para receber a tarefa a ser movida. A equação abaixo mostra como a utilização de cada processador p é estimada durante a aplicação do operador de mutação:

$$ut_p = \sum_{t \in \{Map(p)\}} \mathbb{E}(Exec_{t,p}) / \mathbb{E}(A_{i,t}), \tag{6.7}$$

onde $Map(p)$ é o conjunto de tarefas mapeadas no processador p, $\mathbb{E}(Exec_{t,p})$ é o tempo médio de execução da tarefa t no processador p (recordar a Equação (5.1)), e $\mathbb{E}(A_{i,t})$ representa o intervalo médio de tempo entre as chegadas de dados para a tarefa t. Este intervalo é aproximado pelo valor médio da variável aleatória IT_i (Seção 4.3), que está associada ao ator i do componente fracamente conectado ao qual o ator que representa t pertence. De maneira similar, antes de aplicar este operador em um canal de dados, MODSES usa a Equação (5.2) para verificar a banda

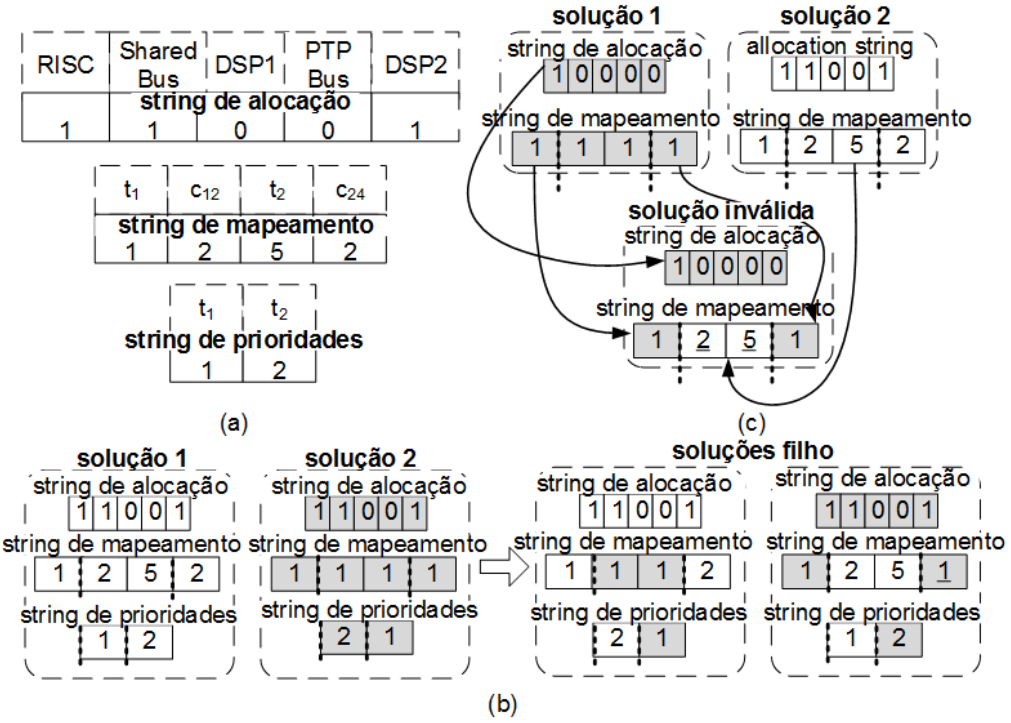


Figura 6.1: (a) Exemplo de *string* de alocação, mapeamento e prioridade. (b) Exemplo de aplicação do *crossover* de dois pontos e *crossover* parcialmente mapeado. (c) Solução inválida gerada por *crossover*.

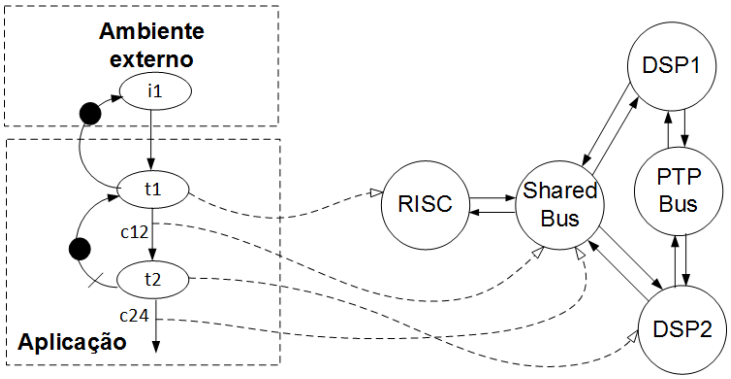


Figura 6.2: Mapeamento de um HSDG em um ATG.

demandada em cada barramento e garantir que o canal a ser movido não faça com que a banda máxima suportada pelo modelo seja violada.

O operador de *crossover* de $freq$ copia para os filhos as frequências de operação dos pais. Dadas duas soluções x_1 e x_2, *crossover* de dois pontos e *crossover* parcialmente mapeado são aplicados, respectivamente, nos *strings* de mapeamento e prioridade de x_1 e x_2 (relembre como esses operadores de *crossover* funcionam na Subseção 2.2.3). A Figura 6.1b mostra um exemplo de aplicação desses operadores. Note que a aplicação dos operadores mutação e *crossover* pode levar à geração de soluções inviáveis (recorde a definição de uma solução viável na Seção 6.1). Quando isso acontece, uma heurística de reparo é adotada para tentar corrigir canais de comunicação mapeados incorretamente. Na Figura 6.1b, a segunda solução gerada possui um mapeamento inviável, pois as restrições (6.4) ou (6.5) serão violadas. No entanto, se o canal de comunicação c_{24} for movido para *SharedBus* (caso a tarefa receptora de c_{24} esteja mapeada em *RISC*), ou movido para *DSP*2 (caso a tarefa receptora também esteja em *DSP*2), então o novo mapeamento será viável. As soluções que o algoritmo não consegue reparar são excluídas.

O Algoritmo 2 mostra a heurística de reparo. Ele recebe como argumento a lista de canais de comunicação que estão mapeados incorretamente, e o mapeamento atual, que não inclui o mapeamento dos canais de comunicação recebidos como argumento. O objetivo do algoritmo é retornar um mapeamento válido. Ele inicia reordenando aleatoriamente a lista de canais de comunicação mapeados incorretamente (linha 1). Os passos a seguir são executados para cada canal da lista reordenada (linha 2). Primeiro, o algoritmo verifica se o receptor e emissor do canal estão mapeados no mesmo elemento (linhas 3-5). Caso estejam, o canal de comunicação também é mapeado nesse mesmo elemento (linhas 6 e 7). Caso contrário, o algoritmo usa a função $BindingList(m,c)$ para receber uma lista com todos os elementos em que o canal de comunicação $c = (t_i, t_j)$ pode ser mapeado, considerando o mapeamento atual m. Mais especificamente, essa função retorna todo barramento w para o qual: (i) a utilização máxima suportada não será violada, e (ii) as médias $\mathbb{E}(COPY_{i,z1,j,w})$ e $\mathbb{E}(COPY_{j,z2,i,w})$ estão definidas (restrição (6.6)), onde $z1$ e $z2$ são os processadores em que as tarefas t_i e t_j estão mapeadas, respectivamente. Em seguida, o algoritmo tenta mapear o canal de comunicação em um dos elementos retornados (linhas 10-14). A função $ValidMapping(m,t,r)$ testa se, considerando o mapeamento atual m, é valido mapear a tarefa t no elemento r. Essa função retorna verdadeiro, caso as restrições (6.2) e (6.5) sejam satisfeitas, e falso, caso contrário. Caso não seja válido mapear o canal em nenhum elemento disponível, o algoritmo retorna um mapeamento vazio (linhas 15 e 16), indicando que não foi possível reparar a solução.

Uma outra fonte de mapeamentos inválidos está relacionada apenas ao operador de *crossover* de dois pontos e é exemplificada na Figura 6.1c. Note que c_{12} e t_2 da solução gerada estão mapeados, respectivamente, em *SharedBus* e *DSP*2. Porém, apenas *RISC* está alocado para esta solução (ver seu *string* de alocação), o que viola a restrição (6.2). Para impedir esse comportamento, MODSES adota a abordagem de EMOGAC (DICK, 2002), que consiste em definir o conceito de *clusters* de soluções. Um *cluster* de soluções agrupa soluções que possuem o mesmo *string* de alocação, e os operadores de *crossover* só podem ser aplicados nas soluções em um mesmo *cluster*. Nos experimentos, foi observado que, além de reduzir a geração de soluções com mapeamentos inviáveis, os *clusters* também ajudam o algoritmo a escapar dos mínimos locais.

Um *cluster* é representado pelo *string* de alocação de suas soluções. O operador de mutação do *string* de alocação aleatoriamente seleciona uma posição e inverte o bit nessa posição (Figura 2.9a). Diferentemente de EMOGAC, não foram definidos operadores de *crossover* para o *string* de alocação, uma vez que isso deixa o algoritmo mais complexo e, além disso, como pequenas mudanças no string de alocação podem resultar em grandes mudanças nas

Input : A lista de canais a serem mapeados, *channels*; o mapeamento atual, *m*.
Output : Um novo mapeamento, *m*, caso um mapeamento válido para os canais tenha sido encontrado, ou um mapeamento vazio, caso contrário.

```
1  RandReoder(channels)                          // reordena aleatoriamente a lista de canais
2  foreach canal c in channels do
3      sender := sender_t(c)                     // recebe a tarefa emissora de c
4      receiver := receiver_t(c)                 // recebe a tarefa receptora de c
       // se a tarefa emissora e receptora estiverem mapeadas no mesmo recurso
5      if get_resource(m, sender) == get_resource(m, receiver) then
6          proc := get_resource(m, receiver)
7          m := m ∪ {(c, proc)}                  // mapeia c no mesmo recurso, proc, que as tarefas
8      else
9          resources := BindingList(m, c)        // recebe a lista de recursos capazes de receberem c
10         RandReoder(resources)                 // reordena aleatoriamente a lista de recursos
11         foreach resource r in resources do
12             if ValidMapping(m, c, r) then     // se o mapeamento de c em r não gera um mapeamento
               inválido
13                 m := m ∪ {(c, r)}             // mapeia c em r
14                 break
15         if get_resource(m, c) == NULL then    // se o canal c não estiver mapeado em nenhum
           lugar
16             return {}                         // a heurística não achou um mapeamento válido
17 return m                                      // a heurística achou o mapeamento válido
```

Algoritmo 2: Algoritmo de mapeamento dos canais de comunicação.

características da solução, dificilmente as boas características dos pais seriam combinadas e repassadas para as soluções-filho.

6.2.3 Aptidão de uma solução e de um *cluster* de soluções

Como mencionado na subseção anterior, operações de *crossover* só devem ocorrer entre soluções de um mesmo *cluster*. Devido a essa característica, diferentemente de um algoritmo genético comum, MODSES gerencia dois tipos de população: a população de soluções dentro de um *cluster* e a população de *clusters*. Dessa forma, cada *cluster* e cada solução dentro de um *cluster* recebe um valor de aptidão. O valor de aptidão de um cluster (solução) é usado pelos mecanismos de seleção para escolher que *clusters* (soluções) serão recombinados e passarão para a geração seguinte. Essa subseção explica como a aptidão de cada cluster e cada solução é calculada.

6.2.3.1 Aptidão de uma solução

Após avaliar uma solução, para cada função objetivo f_i definida na Equação (6.1), MODSES armazena o valor resultante da avaliação (a estimativa da função objetivo), a variância da função objetivo (denotada por $\widehat{\sigma}_i^2$), o comprimento do intervalo de confiança da estimativa e um valor booleano indicando se a avaliação resultou ou não em uma violação de restrição. MODSES trata todas as funções objetivo da mesma forma, no entanto, apenas a função objetivo f_3 (probabilidades de violação de *deadlines*) é avaliada por simulação estocástica. Assim, apenas ela possui variância e intervalo de confiança. Dessa forma, as outras funções objetivo são tratadas como um caso particular de estimativa cuja variância e intervalo de confiança é zero. Como explicado na Subseção 2.2.5.2, esse vetor de estimativas precisa ser convertido para um escalar,

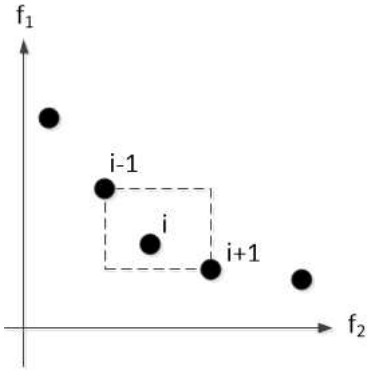


Figura 6.3: Cálculo do operador *crowding distance*.

que define a aptidão da solução correspondente. Para converter o vetor de estimativas, primeiro MODSES insere as soluções de todos os *clusters* em um *pool* de soluções. Em seguida, ele classifica estas soluções em dois grupos: grupo 1, soluções que não são dominadas por nenhuma outra solução no *pool*, e grupo 2, soluções não atendem ao critério de participação do grupo 1. Depois de classificar as soluções do *pool*, dois métodos diferentes são usados para definir a aptidão das soluções em cada grupo.

Aptidão das soluções do grupo 1: A aptidão de uma solução no primeiro grupo é igual ao *crowding distance* da solução (DEB et al., 2002). O *crowding distance* é uma estimativa da densidade de soluções na vizinhança de cada solução. Essa estimativa é feita calculando o perímetro do cuboide que é formado pelos vértices das soluções vizinhas mais próximas. Na Figura 6.3, o *crowding distance* da i-ésima solução é igual à média dos tamanhos dos lados do cuboide (mostrado como uma caixa com linhas tracejadas). Um valor alto de *crowding distance* indica que a solução não possui muitas soluções próximas a ela. Dessa forma, ao utilizar o *crowding distance* para definir a aptidão de uma solução, MODSES procura preservar a diversidade das soluções não dominadas.

O Algoritmo 3 mostra o procedimento usado para calcular o *crowding distance* de uma lista de soluções. O algoritmo inicia calculando o número de soluções na lista de soluções e o número de objetivos considerado (linhas 1 e 2). Em seguida, o valor de *crowding distance* de todas as soluções é zerado (linha 3 e 4). Depois, para cada objetivo o, os seguintes passos são executados (linha 5). Primeiro, a lista de soluções é ordenada de acordo com o objetivo o (linha 6). Baseado nessa ordenação, o algoritmo obtém os valores máximos e mínimos do objetivo o (linhas 7 e 8). Em seguida, o algoritmo atribui valor máximo de *crowding distance* para as soluções que possuem os valores máximo e mínimo de o (linhas 9 e 10). Todas as outras soluções intermediárias recebem um valor de *crowding distance* que é igual à diferença absoluta normalizada dos valores do objetivo o de duas soluções adjacentes (linhas 11 e 12). Esse cálculo é realizado para as outras funções objetivo. O valor final do *crowding distance* é igual à soma das distâncias individuais para cada objetivo.

Aptidão das soluções dominadas: Ao contrário do procedimento de atribuição de aptidão para soluções do primeiro grupo, o procedimento de atribuição de aptidão para soluções do segundo grupo considera as estimativas de variância das funções objetivo. O cálculo da aptidão das soluções do segundo grupo é feito em três passos. O primeiro passo é estimar qual a probabilidade de que uma solução a domine outra solução b (denotado por $Pr(a \prec b)$, $a \neq b$):

Input : Uma lista de soluções, *sols*, em que cada solução possui um vetor de objetivos, *objs*, e um valor de *crowding distance*, *distance*.
Output : Uma lista de soluções com o valor de *crowding distance*, *distance*, atualizado.

$l := SizeOf(sols)$ // número de soluções em *sols*
$m := SizeOf(sols[0].objs)$ // número de objetivos
foreach *solution s in sols* **do**
  $s.distance := 0$ // inicia as distâncias
foreach *objetivo o* **in** $\{1, \ldots, m\}$ **do** // para cada objetivo do problema
  $sols := Sort(sols, o)$ // ordena as soluções usando o objetivo *o* como critério
  $omin := sols[1].objs[o]$ // valor mínimo do objetivo *o*
  $omax := sols[l].objs[o]$ // valor máximo do objetivo *o*
  $sols[1].distance := +\infty$ // atribui distância máxima para a solução com valor mínimo
  $sols[l].distance := +\infty$ // atribui distância máxima para a solução com valor máximo
  for $i := 2$ **in** $l-1$ **do**
    $sols[i].distance := sols[i].distance + (sols[i+1].objs[o] - sols[i-1].objs[o])/(omax - omin)$
return *sols*

Algoritmo 3: Algoritmo para calcular o *crowding distance*.

$$Pr(a \prec b) = \begin{cases} 0, \text{ se } \forall i \;\; \widehat{\sigma}_i^2(a) = 0 \wedge \widehat{\sigma}_i^2(b) = 0 \wedge f_i(a) = f_i(b) \\ 0, \text{ se } \exists i \;\; \widehat{\sigma}_i^2(a) = 0 \wedge \widehat{\sigma}_i^2(b) = 0 \wedge f_i(a) > f_i(b) \\ \prod_{i=1}^{k} Pr(f_i(a) \leq f_i(b)), \qquad \text{caso contrário.} \end{cases} \tag{6.8}$$

Quando f_i tem variância diferente de zero, a probabilidade $Pr(f_i(a) \leq f_i(b))$ na Equação (6.8) pode ser calculada da seguinte forma: primeiro, assuma que o ruído de f_i pode ser aproximado pela distribuição Normal $N(f_i, \widehat{\sigma}_i^2)$, o que é razoável devido ao tratamento estatístico da saída da simulação dado pelo *framework* Akaroa. Então, dado que a soma de duas variáveis aleatórias normalmente distribuídas é também uma variável aleatória normalmente distribuída (com média igual à soma das duas médias, e variância igual à soma das duas variâncias), a probabilidade $Pr(f_i(a) \leq f_i(b)) = Pr(f_i(a) - f_i(b) \leq 0)$ pode ser aproximada por $Pr(X \leq 0)$, onde $X \sim N(f_i(a) - f_i(b), \widehat{\sigma}_i{}^2(a) + \widehat{\sigma}_i{}^2(b))$.

Baseado na Equação (6.8), o segundo passo do cálculo da aptidão das soluções do segundo grupo é atribuir um valor denominado de força, $S(x_i)$, para cada solução do primeiro e segundo grupo. Esse valor indica o somatório das probabilidades de que uma solução x_i domina as outras soluções:

$$S(x_i) = \sum_j Pr(x_i \prec x_j), \forall x_j, i \neq j. \tag{6.9}$$

Finalmente, o valor de aptidão, $F(x_i)$, atribuído a uma solução no segundo grupo x_i é igual à força das soluções que ela domina menos a força das soluções que a dominam:

$$F(x_i) = \sum_{x_i \prec x_j} S(x_j) - \sum_{x_j \prec x_i} S(x_j), \forall x_j, i \neq j \tag{6.10}$$

6.2.3.2 Aptidão de um *cluster* de soluções

A aptidão de um *cluster* é definida como sendo o valor de aptidão máximo entre as soluções que fazem parte do *cluster* e pertencem ao grupo 1 de soluções. Caso o *cluster* não tenha soluções que participem do primeiro grupo, ele recebe a menor aptidão possível.

6.2.4 Inicialização

Na inicialização do algoritmo, um *string* de alocação para cada *cluster* é aleatoriamente gerado. Um operador de reparo pode ser aplicado nesse *string* para garantir que todas as tarefas e canais de comunicação possam ser mapeados em pelo menos um elemento. Dado o *string* de alocação de um *cluster*, MODSES gera soluções para popular o *cluster*. O código que define a frequência de operação da plataforma, assim como o *string* de prioridades dessas soluções são aleatoriamente gerados. O *string* de mapeamento de cada solução é gerado em duas fases. Para acelerar a convergência do algoritmo, na primeira fase, o algoritmo guloso randomizado, Algoritmo 4, é usado para mapear as tarefas de processamento nos elementos de processamento alocados. Este algoritmo inicia recebendo os componentes fracamente conectados do HSDG (linha 1). Em seguida, o mapeamento é iniciado (linha 2). Depois, para cada partição p, os seguintes passos são executados (linha 3). Primeiro, o algoritmo gera a lista de tarefas não mapeadas de p e aleatoriamente seleciona uma tarefa, t, dessa lista (linhas 4-6). Depois, o algoritmo seleciona o processador que pode executar t mais rapidamente (linha 7). Esse processador também não deve ter fator de utilização maior que 1. Com probabilidade dada pela constante P, o algoritmo ignora o processador escolhido anteriormente e escolhe aleatoriamente outro processador capaz de executar t. Esta escolha aleatória também é feita caso todos os processadores capazes de executar t já estejam sobrecarregados (linhas 8 e 9). Por fim, o algoritmo tenta mapear todas as tarefas da partição no processador escolhido pelos passos descritos anteriores (linhas 10-13). Caso alguma tarefa da partição não possa ser mapeada no processador escolhido, o algoritmo volta a gerar a lista de tarefas não mapeadas da partição e inicia o processo de escolha do processador para executar essas tarefas (linhas 6-9). Utilizando esse algoritmo, a probabilidade de que tarefas que trocam mensagens sejam mapeadas no mesmo elemento aumenta, o que reduz o *overhead* de comunicação.

Input : A lista de elementos de processamento alocados, *procs*; o HSDG, G.
Output : Um mapeamento para as tarefas de processamento.

```
1  partitions := Partition(G)                          // recebe os componentes do HSDG
2  mapping := {}                                       // inicia o mapeamento
3  foreach partição p in partitions do
4      not_mapped_tasks := GetTasks(p)      // recebe a lista de tarefas de p que não foram
       mapeadas
5      while not_mapped_tasks ≠ {} do
6          t := SelectRandomTask(not_mapped_tasks)     // seleciona uma tarefa aleatoriamente
           // seleciona o processador mais rápido capaz de executar t; o processador
              selecionado também precisa ter um fator de utilização que não seja maior
              que 1
7          proc := SelectFasterProc(t)
8          if proc == NULL ∨ Uniform(0,1) ≤ P then          // seleciona outro proc. com
           probabilidade P
9              proc := SelectRandomProc(t)       // seleciona aleatoriamente um proc. capaz de
               executar t
10         foreach task t in not_mapped_tasks do
11             if Capable(t, proc) then    // se o processador proc é capaz de executar a tarefa t
12                 mapping := mapping ∪ {(t, proc)}        // mapeia a tarefa t no processador proc
13                 not_mapped_tasks := not_mapped_tasks \ {t}      // remove t da lista de tarefas
                   não mapeadas
```

Algoritmo 4: Algoritmo que mapeia as tarefas nos elementos de processamento.

Depois que as tarefas são mapeadas nos elementos de processamento, na segunda fase de

inicialização do *string* de mapeamento, o Algoritmo 2 é usado para encontrar um mapeamento válido para os canais de comunicação.

6.2.5 Seleção e reprodução

Após a atribuição dos valores de aptidão, quando duas soluções precisam ser comparadas de forma a determinar a melhor delas, três cenários podem ocorrer: no primeiro cenário, as soluções estão em diferentes grupos. Nesse caso, a solução no primeiro grupo é escolhida. A segunda situação ocorre se as duas soluções estão no mesmo grupo, mas possuem diferentes valores de aptidão. Nesse caso, a solução com maior aptidão é escolhida. Na última situação, as soluções estão no mesmo grupo e possuem a mesma aptidão. Assim, uma delas é escolhida aleatoriamente com igual probabilidade. Quando dois *clusters* precisam ser comparados para escolher o melhor deles, o *cluster* com maior aptidão é selecionado.

MODSES adota a abordagem de seleção por torneio para escolher as soluções e *clusters* que irão passar pelos operadores de *crossover* e mutação para produzir a nova geração da população (Subseção 2.2.3.3). Quando um *cluster* é selecionado para mutação, MODSES executa os seguintes passos. Inicialmente, ele gera um novo *string* de alocação para o *cluster*. Para gerar esse novo *string*, primeiro MODSES seleciona a solução de maior aptidão no *cluster*. O novo *string* de alocação do *cluster* é baseado no *string* de mapeamento da solução selecionada: apenas os elementos dessa solução em que algum canal de comunicação ou tarefa está mapeada estarão alocados no novo *string*. A ideia desse processo de geração do novo *string* de alocação é intensificar a busca em alocações promissoras. Depois de gerar o novo *string*, MODSES aplica o operador de mutação nesse *string*. Em seguida, metade das soluções do *cluster* são excluídas e trocadas por novas soluções. As soluções excluídas são aquelas de menor aptidão. As novas soluções são geradas de acordo com o algoritmo da Seção 6.2.4. As soluções que não foram excluídas são atualizadas para que elas compartilhem o novo *string* de alocação do *cluster*.

6.2.6 Visão geral do algoritmo

O Algoritmo 5 mostra o pseudocódigo de MODSES. O laço mais interno do algoritmo (linhas 11-16) trata da evolução das soluções dentro dos *clusters*, enquanto que o laço mais externo lida com a evolução da população de *clusters* (linhas 6-22). O algoritmo começa inicializando algumas variáveis (linha 1). A variável t representa o número da geração atual da população, *no_improv* é o número de vezes em que não houve melhora entre a geração anterior e a atual, e A é o conjunto com as melhores soluções encontradas até o momento. Depois, a primeira população de *clusters* de soluções é criada utilizando o algoritmo detalhado na Subseção 6.2.4 (linha 2). Em seguida, as soluções de cada *cluster* são avaliadas (linhas 3 e 4). Baseado nos resultados da avaliação, MODSES atribui um valor de aptidão para cada solução e para cada *cluster* (linha 5). A Subseção 6.2.3 explicou como funciona o processo de atribuição de aptidão. Depois da atribuição dos valores de aptidão, MODSES entra no laço principal (linha 6). A condição de parada desse laço define o critério de parada do algoritmo, que é: (i) quando o número máximo de gerações for atingido, ou (ii) quando nenhuma melhora for observada por um dado número vezes em gerações consecutivas.

No laço principal (linhas 7-23), primeiro, o algoritmo aumenta o número da geração atual (linha 7). Em seguida, ele seleciona alguns *clusters* para sofrerem mutação (linhas 8 e 9). Os processos de seleção e mutação de *clusters* foram detalhados na Subseção 6.2.5. O algoritmo adota uma abordagem elitista para garantir que bons clusters sobrevivam entre as gerações. Nessa abordagem, todos os *clusters* da população anterior, CP_{t-1}, são copiados para a população atual,

```
1  t := 0, no_improv := 0, A := {}
2  CP_t = {C_1, C_2, ..., C_n}                          // cria a população inicial de clusters
3  foreach cluster C_i in CP_t do
4      Evaluate(C_i.pop)                                // avalia as soluções no cluster C_i
5  Rank(CP_t)                                           // faz o ranking dos clusters e soluções
6  while t ≤ MaxGeneration ∧ no_improv ≤ MaxNoImprovement do
7      t := t + 1
8      CP'_t := SelectC(CP_{t-1})                       // seleciona os clusters para mutação
9      CO_t := MutateC(CP'_t)                           // realiza mutação nos clusters
10     CP_t := CP_{t-1} ∪ CO_t
11     foreach cluster C_i in CP_t do
12         P' := Select(C_i.pop)                        // seleciona as soluções para crossover
13         O := Crossover(P')                           // realiza crossover nas soluções selecionadas
14         O := Mutation(O)                             // realiza mutação nas soluções
15         Evaluate(O)                                  // avalia as novas soluções
16         C_i.pop := C_i.pop ∪ O
17     UpdateBestSolutions(CP_t, A)                     // atualiza o conjunto com as melhores soluções
       encontradas até o momento
18     if NoImprovement(CP_t, CP_{t-1}) then            // se não houve melhora
19         no_improv:=no_improv + 1
20     else
21         no_improv:=0
22     Rank(CP_t)                                       // faz o ranking dos clusters e soluções
23     Terminate(CP_t)                                  // remove soluções/clusters inferiores
24 return A
```

Algoritmo 5: Algoritmo MODSES.

CP_t. A população atual CP_t é, portanto, composta por *clusters* da população anterior mais os *clusters* que foram gerados por meio da operação de mutação (linha 10).

No laço mais interno, para cada *cluster*, o algoritmo executa as seguintes operações (linha 11). Primeiro, ele seleciona as soluções que sofrerão *crossover* e mutação (linha 12). Depois, *crossover* e mutação são aplicadas a essas soluções (linhas 13 e 14). Em seguida, as novas soluções geradas são avaliadas (linha 15). No final do laço, a nova população do *cluster* é formada por soluções da geração anterior do *cluster* mais as novas soluções (linha 16).

Depois que o algoritmo termina o laço interno, ele atualiza (linha 17) o conjunto A (recorde que esse conjunto mantém as melhores soluções encontradas até o momento). A atualização é feita removendo desse conjunto todas as soluções que são dominadas por alguma solução da geração atual. Depois, as novas soluções da geração atual são adicionadas ao conjunto. Uma solução só pode entrar nesse conjunto caso atenda dois critérios: (i) ela não pode ser dominada por outra solução da geração atual e nem por outra solução já presente no conjunto, e (ii) a solução não pode violar nenhuma restrição. Depois que o conjunto A é atualizado, MODSES verifica se houve melhora na geração atual. Caso não tenha havido, a variável *no_improv* é incrementada, caso contrário, ela é zerada (linhas 18-21). Para manter a população em um tamanho fixo, no final de cada geração, a população é ordenada e os *clusters* e soluções inferiores são excluídos (linha 22 e 23). Por fim, quando o critério de parada do algoritmo é atingido, ele retorna as soluções presentes no conjunto A (linha 24).

6.3 Algoritmo C-MODSES

Em MODSES, a quase totalidade do tempo de busca é gasta com simulações. Esta seção apresenta o algoritmo C-MODSES (***C****onstrained*-MODSES), que é uma versão modificada de MODSES e que tem como objetivo reduzir o tempo gasto com simulações. C-MODSES utiliza uma abordagem baseada em restrições para acelerar o tempo de avaliação de uma solução. Considere um projeto em que um dos requisitos é garantir que a seguinte restrição não seja violada,

$$\theta \leq K, \tag{6.11}$$

onde θ é a probabilidade de violação de um *deadline*, e K é um limite superior neste valor. Então, para verificar se a restrição é violada ou não, MODSES avalia cada solução candidata utilizando simulação estacionária e para a simulação apenas quando uma estimativa precisa de θ é obtida. Porém, note que, muitas vezes, uma estimativa precisa não é necessária: é preciso saber apenas se θ está acima ou abaixo de K. Seria um grande desperdício de recursos computacionais obter uma estimativa precisa de θ apenas para depois saber que ela está muito longe de K. Ao contrário de MODSES, C-MODSES para a simulação assim que ele verifica que é possível decidir se a restrição será violada ou não. Nas próximas subseções, o algoritmo C-MODSES é detalhado, destacando-se as diferenças entre ele e MODSES.

6.3.1 Método de avaliação

Em MODSES, a função objetivo f_3 na Equação (6.1) é definida como sendo o somatório das probabilidades de violação de *deadlines*. Em *C-MODSES* ela é definida da seguinte forma

$$f_3 = \sum_{i=1}^{k} (\theta_i - D_i) \times v_i, \tag{6.12}$$

onde k é o número de *deadlines*; θ_i é a probabilidade de violação do *deadline* i; D_i é a probabilidade de violação máxima tolerável, e $v_i \in \{0,1\}$, tal que $v_i = 1$, se $D_i > \theta_i$, e $v_i = 0$, caso contrário. Note que a função f_3 penaliza as soluções que violam restrições temporais.

Como já mencionado, para estimar θ_i, MODSES utiliza Akaroa para determinar quando a simulação deve parar. Akaroa, por sua vez, adota uma abordagem sequencial em que a verificação do critério de parada é feita sempre depois que um determinado número de amostras é coletado. Os momentos em que a verificação é feita são chamados de *checkpoints*. Sempre que um *checkpoint* é atingido, Akaroa verifica se a metade do comprimento do intervalo de confiança para a estimativa (denotado por Δ) é menor que a precisão absoluta especificada (denotada por γ). Caso $\Delta < \gamma$, então a simulação para. Caso contrário, a simulação continua e essa checagem só será feita novamente no próximo *checkpoint*. Como C-MODSES apenas deseja saber se a probabilidade de violação θ_i está ou não acima da probabilidade máxima aceitável D_i, uma estimativa precisa de θ_i não é necessária na maioria dos casos. Dessa forma, a cada *checkpoint*, C-MODSES adota o Algoritmo 6 para determinar se a simulação deve parar ou continuar. A ideia do algoritmo é: sempre que em um *checkpoint* o intervalo de confiança não incluir D_i, então a simulação pode parar. Se ele inclui D_i, então procede-se normalmente e a simulação só para quando a metade do comprimento do intervalo de confiança Δ for menor que a precisão absoluta γ. É importante ressaltar que à medida que mais amostras da simulação são obtidas, Δ tende a diminuir.

A Figura 6.4 mostra três situações que podem ocorrer em um dado *checkpoint*. Na

Input : Precisão absoluta, γ; limite superior para a probabilidade de violação do *deadline*, D_i; estimativa atual de θ_i, $\widehat{\theta}_i$; metade do comprimento do intervalo de confiança para θ_i, Δ.
Output : 0, se a simulação deve continuar; 1, se a simulação deve parar e $\widehat{\theta}_i \leq D_i$; 2, se a simulação deve parar e $\widehat{\theta}_i > D_i$.

```
if (D_i ≤ (θ̂_i − Δ)) ∨ (D_i ≥ (θ̂_i + Δ)) ∨ (Δ < γ) then
    if θ̂_i ≤ D_i then
        return 1
    else
        return 2
else
    return 0
```

Algoritmo 6: Algoritmo para o critério de parada das simulações conduzidas por C-MODSES.

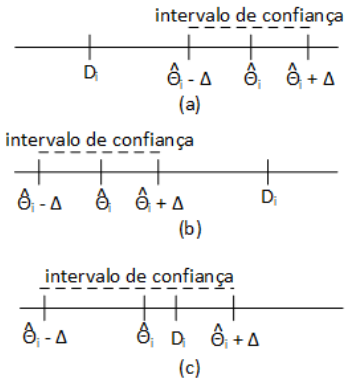


Figura 6.4: Exemplos de situações que podem ocorrer em um dado *checkpoint*.

primeira situação (Figura 6.4a), o intervalo de confiança gerado por Akaroa não inclui D_i, e como D_i é maior que a estimativa atual $\widehat{\theta}_i$, o algoritmo conclui que a simulação pode parar e que a restrição é violada. De maneira similar, na segunda situação (Figura 6.4b), como o intervalo de confiança não inclui D_i e D_i é menor que a estimativa atual $\widehat{\theta}_i$, o algoritmo conclui que a simulação pode parar e a restrição não é violada. Na última situação (Figura 6.4c), o intervalo de confiança inclui D_i. Assim, caso o a metade do comprimento do intervalo de confiança Δ seja maior que a precisão absoluta γ, a simulação deve continuar, caso contrário, a simulação para e o algoritmo conclui que a restrição é violada, já que $D_i > \widehat{\theta}_i$.

De acordo com a Equação (6.12), além de checar $D_i \leq \theta_i$, C-MODSES precisa também estimar θ_i quando $D_i > \theta_i$. Caso $D_i > \theta_i$ seja verdadeiro, o algoritmo usa a estimativa atual de θ_i, $\widehat{\theta}_i$. Note, no entanto, que como estamos terminando a simulação antecipadamente, $\widehat{\theta}_i$ pode não ser confiável e isso pode fazer com que o algoritmo tome decisões erradas. A próxima seção mostra como C-MODSES trata esse problema.

6.3.2 Visão geral do algoritmo

Exceto pelo novo método de avaliação das soluções e a reformulação da função objetivo, o algoritmo C-MODSES tem uma estrutura igual à do algoritmo MODSES (Algoritmo 5). No entanto, como os resultados obtidos por simulação ao adotar o critério de parada descrito no Algoritmo 6 podem, algumas vezes, ser diferentes dos resultados obtidos caso a simulação fosse parada apenas quando ela alcançasse a precisão de fato especificada, isso pode fazer com que o algoritmo tome decisões erradas durante a busca.

As soluções que mais influenciam as decisões tomadas pelo algoritmo durante a busca

são aquelas que possuem maior aptidão, uma vez que a exploração tende a seguir na direção das melhores soluções encontradas. Dessa forma, para diminuir a chance de decisões erradas serem tomadas é preciso garantir que as soluções de maior aptidão tenham resultados confiáveis. No Algoritmo 5, as soluções de maior aptidão são aquelas presentes no conjunto A. Assim, C-MODSES adota a seguinte abordagem antes de atualizar o conjunto A. Primeiro, ele verifica se a solução a ser adicionada é dominada por outra solução em A. Se ela não for dominada, então C-MODSES verifica se ela foi simulada até que a precisão especificada fosse de fato atingida ou se o Algoritmo 6 fez com que a simulação fosse parada antecipadamente. Caso tenha sido parada antecipadamente, C-MODSES simula a solução novamente até que a precisão especificada seja atingida (ou seja, o Algoritmo 6 não é usado). Como os resultados na nova simulação podem ser diferentes dos resultados anteriores, C-MODSES verifica novamente se a solução é dominada antes de finalmente armazená-la no conjunto A. Utilizando essa abordagem, garante-se que a maior parte dos recursos computacionais sejam gastos apenas nas soluções que indicam ter boa qualidade (ou seja, soluções que não violam restrições e soluções que não são dominadas por nenhuma outra solução). Além disso, como o algoritmo retorna apenas as soluções presentes em A, garante-se também que todas as soluções retornadas ao projetista são confiáveis.

6.4 Considerações finais

Esse capítulo apresentou os algoritmos propostos para exploração do espaço de projeto de sistemas embarcados de tempo-real não críticos. Dois algoritmos foram propostos: MODSES e C-MODSES. Os dois algoritmos possuem o mesmo objetivo: dada a especificação gerada pelas atividades de especificação e caracterização, encontrar um subconjunto representativo do conjunto ótimo de Pareto (ou uma aproximação desse conjunto), considerando as seguintes medidas de projeto: custo monetário, potência consumida e probabilidades de violação de *deadlines*. A implementação atual dos algoritmos possui aproximadamente 1000 linhas de código C++. Na implementação, a biblioteca OpenMP foi usada para paralelizar as simulações conduzidas pelos algoritmos.

Em MODSES, a quase totalidade do tempo de busca é gasto com simulações. Caso o projetista possa estabelecer quais são os valores máximos de probabilidade de violação de *deadlines* aceitáveis, então, ao invés de MODSES, ele pode utilizar C-MODSES, que é uma alternativa mais rápida. C-MODSES trata violações de *deadlines* como restrições a serem cumpridas e não como um objetivo a ser otimizado. Esse algoritmo acelera o tempo de busca evitando desperdiçar recursos computacionais com a simulação de soluções de baixa qualidade (soluções dominadas) e que possuem grandes chances de violar restrições.

7 Resultados experimentais

Este capítulo apresenta os experimentos conduzidos para demonstrar a aplicação do método de exploração proposto. Nesse sentido, faz parte dos objetivos deste capítulo: (i) analisar a eficiência dos algoritmos de exploração desenvolvidos, (ii) verificar a viabilidade do fluxo de atividades proposto (Figura 1.2), e (iii) validar os modelos de avaliação desenvolvidos para representar as características do sistema. A seção a seguir apresenta o conjunto de experimentos realizados para verificar a eficiência dos algoritmos de exploração. Na Seção 7.2, analisamos a viabilidade do fluxo de atividades proposto, assim como a exatidão dos modelos de avaliação, através da exploração do espaço de projeto de um sistema embarcado real (*SHOUTcast player*). Por último, a Seção 7.3 conclui esse capítulo.

7.1 Eficiência de MODSES e C-MODSES

Para verificar a eficiência de MODSES e C-MODSES, esses algoritmos foram aplicados em problemas baseados em vários *benchmarks* disponíveis na literatura. As próximas subseções apresentam os resultados produzidos pelos algoritmos propostos e uma comparação entre eles e os seguintes algoritmos:

- **Random**: Esse algoritmo foi especificamente concebido para que os algoritmos propostos pudessem ser comparados a uma busca aleatória. Em Random, soluções com diferentes alocações, mapeamentos e atribuições de prioridades são geradas aleatoriamente.

- **SPGA** (ESKANDARI; GEIGER, 2009; ESKANDARI; GEIGER; BIRD, 2007): SPGA é um exemplo representativo do estado da arte dos algoritmos genéticos multiobjetivo. Nos resultados disponíveis na literatura, SPGA apresentou resultados promissores quando comparado a NSGA-II (ESKANDARI; GEIGER, 2009). A escolha específica por SPGA na comparação se deve ao fato de que os algoritmos propostos se baseiam nos mecanismos de seleção de SPGA. SPGA foi implementado de acordo com a descrição disponível em (ESKANDARI; GEIGER; BIRD, 2007). Como o *string* de alocação de uma solução pode ser deduzido do *string* de mapeamento, a codificação adotada em SPGA considera apenas *strings* de prioridades e mapeamento. Assim, SPGA também não utiliza o conceito de *clusters*.

- **EMOGAC** (DICK, 2002): EMOGAC é uma ferramenta de otimização multiobjetivo para sistemas embarcados de tempo-real críticos. Essa ferramenta mostrou ótimo desempenho em diversos problemas da literatura (DICK, 2002). Como EMOGAC foi desenvolvida focando em sistemas críticos, as decisões de projeto tomadas por essa ferramenta são baseadas em cenários de pior caso. O objetivo da comparação com

EMOGAC é verificar a eficiência dos algoritmos propostos em relação aos algoritmos que focam no pior caso. EMOGAC foi gentilmente cedida por seus criadores para que pudesse ser usada nessa comparação.

Os resultados apresentados nesta seção foram obtidos em um computador com a seguinte configuração: Intel i7 2.3 GHz com 8 GB de memória e sistema operacional Linux.

7.1.1 Descrição dos *benchmarks*

Os problemas utilizados nos experimentos foram baseados em três *benchmarks* diferentes:

- Os problemas de Hou e Wayne (HOU; WOLF, 1996) (denominados *Hou 12*, *Hou 13*, *Hou 34*, *Hou 12 Clustered*, *Hou 13 Clustered* e *Hou 34 Clustered*).
- O codec de vídeo com períodos e *deadlines* iguais a 22 descrito em (BLICKLE, 1997) (denominado *Video codec*).
- Os problemas do *benchmark* E3S (DICK, 2014, 2002) (denominados *Consumer*, *Auto-Indust*, *Office-Automation*, *Networking* e *Telecom*). Este *benchmark* é composto por dados obtidos por medições em processadores reais (ex.: AMD ElanSC520, Analog Devices 21065L e Motorola MPC555) considerando aplicações do *Embedded Microprocessor Benchmark Consortium* (EMBC).

Nesses *benchmarks*, as aplicações são definidas como um conjunto de grafos acíclicos direcionados (DAGs - *Directed Acyclic Graphs*). Os vértices dos DAGs representam as tarefas da aplicação e os arcos representam comunicações entre as tarefas. Cada DAG é anotado com seus respectivos *deadlines* e seu respectivo período de ativação, e cada arco é anotado com a quantidade de dados que precisa ser transferida de uma tarefa para outra. Assim, o DAG pode ser facilmente convertido para um HSDG, que é o modelo de aplicação adotado por este trabalho. No HSDG, os atores que representam processos do ambiente externo são usados para modelar os períodos de ativação dos DAGs.

Para avaliar estes *benchmarks*, assumimos o modelo de comunicação adotado por EMOGAC. De acordo com este modelo, comunicações entre tarefas mapeadas no mesmo processador têm tempos insignificantes e, por isso, tais tempos não são representados. O tempo de atraso devido ao envio e recebimento de mensagens entre tarefas mapeadas em processadores diferentes é modelado por tarefas de comunicação (MANOLACHE; ELES; PENG, 2008; TEICH, 2012), que devem ser mapeadas nos elementos de comunicação da plataforma. Prioridades devem ser atribuídas às tarefas de comunicação para indicar quais mensagens possuem maior prioridade para serem transferidas no elemento de comunicação.

Os *benchmarks* E3S e de Hou e Wayne informam apenas que tipos de elementos de processamento e comunicação podem ser usados na arquitetura. Baseado na descrição desses elementos, para os problemas de Hou e Wayne e os problemas de E3S foram definidos ATGs com, respectivamente, três e duas instâncias de cada elemento especificado. A Tabela 7.1 exibe algumas informações sobre a dimensão dos problemas considerados.

Os *benchmarks* adotados foram originalmente desenvolvidos considerando que os tempos de execução das tarefas assim como os intervalos (períodos) de ativação dos DAGs são constantes. Em particular, estes *benchmarks* assumem que o tempo de execução das tarefas são sempre iguais ao WCET. Dessa forma, as seguintes suposições foram feitas com relação a esses tempos:

Problema	# processadores	# elementos de comunicação	# tarefas	# arcos de comunicação	# deadlines
Hou 1&2	9	3	20	29	3
Hou 1&3	9	3	20	29	5
Hou 3&4	9	3	20	29	5
Hou 1&2 Clustered	9	3	8	7	3
Hou 1&3 Clustered	9	3	7	5	4
Hou 3&4 Clustered	9	3	6	4	4
Video codec	9	4	19	24	2
Consumer	34	12	12	12	3
Auto-Indust	34	12	24	21	4
Office-Automation	34	12	5	5	2
Networking	34	12	13	9	2
Telecom	34	12	13	9	9

Tabela 7.1: Informações sobre a dimensão dos *benchmarks*.

- **Problemas de Hou e Wayne**: Os tempos de execução das tarefas de processamento são uniformemente distribuídos no intervalo [0,9 $\times$ WCET, WCET] e os intervalos de ativação dos DAGs são uniformemente distribuídos no intervalo [intervalo de ativação, 1,4 $\times$ intervalo de ativação].
- **Codec de vídeo**: Os tempos de execução das tarefas possuem distribuição de probabilidade exponencial truncada com média WCET/10 e que exclui os valores que excedem o WCET; os intervalos de ativação originais foram adotados.
- **E3S *benchmark***: Os tempos de execução são uniformemente distribuídos no intervalo [0,45 $\times$ WCET, WCET]; os intervalos originais de ativação foram adotados.

Por último, consideramos que preempção de tarefas não é suportado.

7.1.2 MODSES: resultados

Os algoritmos MODSES, Random, EMOGAC e SPGA foram configurados para, simultaneamente, otimizar custo monetário e probabilidades de violação de *deadlines*. Para cada problema, os algoritmos foram executados 10 vezes com sementes aleatórias variando de 1 a 10. As seguintes medidas de qualidade foram adotadas para comparar os algoritmos: (i) diferença de cobertura de dois conjuntos (medida D), e (ii) cobertura de dois conjuntos (medida C). Recorde como essas duas medidas são definidas na Subseção 2.2.5.3. Os parâmetros de todos os algoritmos foram os mesmos em todos os experimentos. Random só possui um parâmetro: a quantidade soluções visitadas, que foi configurada para ser igual à quantidade de soluções visitadas por MODSES. Os parâmetros usados em EMOGAC foram os parâmetros sugeridos pelos criadores dessa ferramenta. Os parâmetros usados em SPGA foram os mesmos parâmetros usados em MODSES. O *framework* Akaroa foi configurado para parar as simulações sempre que,

com 95% de confiança, a metade do comprimento dos intervalos de confiança para as estimativas fossem menores que 0,015.

As Figuras 7.1, 7.2 e 7.3 mostram uma comparação entre MODSES e os outros algoritmos, considerando as medidas C e D. Essas figuras apresentam as médias com intervalo de 95% de confiança para cada medida. Os intervalos de confiança para essas medidas foram gerados usando o método *bootstrap* com 1000 amostras de *bootstrap*.

A comparação entre MODSES e SPGA (Figura 7.1) mostra que MODSES é superior em todos os problemas. Em particular, SPGA não foi capaz de encontrar nenhuma solução viável para os problemas *Auto-Indust*, *Networking*, e *Telecom*, e por isso os resultados para esses problemas não são apresentados. Dado que os mecanismos de seleção de MODSES são baseados em SPGA, esses resultados revelam a importância das heurísticas construtivas de inicialização e o conceito dos *clusters* de MODSES.

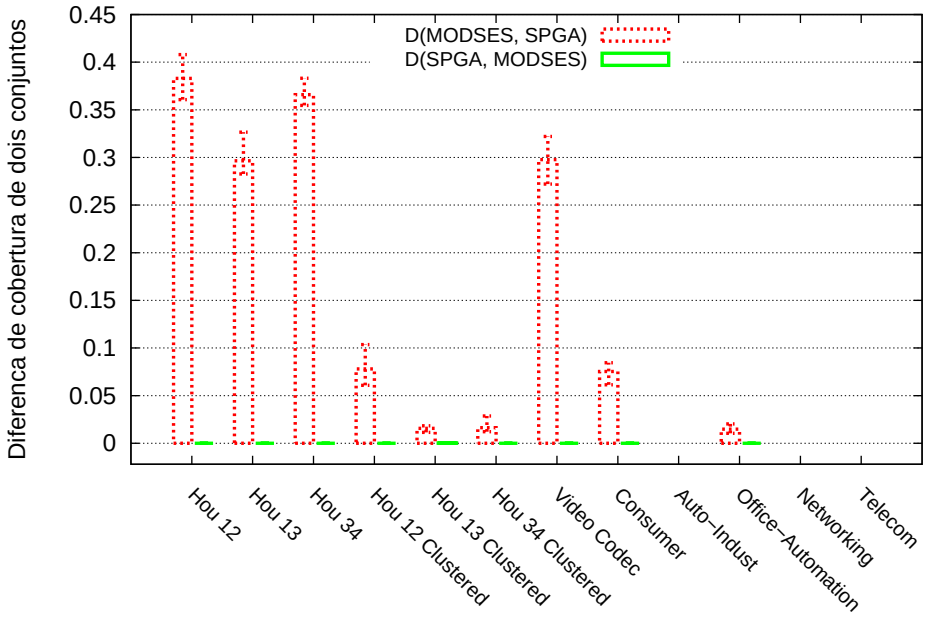
Em todos os problemas considerados, o algoritmo Random é inferior a MODSES (Figura 7.2). O principal motivo disso é o grande tamanho do espaço de soluções dos problemas, o que faz com que buscas aleatórias não tenham bom desempenho. Assim como SPGA, Random também não foi capaz de encontrar nenhuma solução viável para os problemas *Auto-Indust*, *Networking* e *Telecom*.

A comparação de MODSES com EMOGAC (Figura 7.3) mostra que, para os problemas *Networking* e *Office-Automation*, os dois algoritmos apresentaram desempenho igual. Além disso, MODSES se mostrou superior a EMOGAC nos problemas *Telecom*, *Auto-Indust*, *Consumer*, *Hou 34 Clustered*, *Hou 13 Clustered*, *Hou 12 Clustered* e *Hou 34*. Nos problemas *Hou 12* e *Hou 13*, os dois apresentaram desempenho similar para a medida C, porém, quando a medida D é considerada, MODSES é muito superior nos dois problemas. Não foi possível aplicar EMOGAC no problema *Video Codec*, pois EMOGAC não suporta as restrições impostas por esse problema.

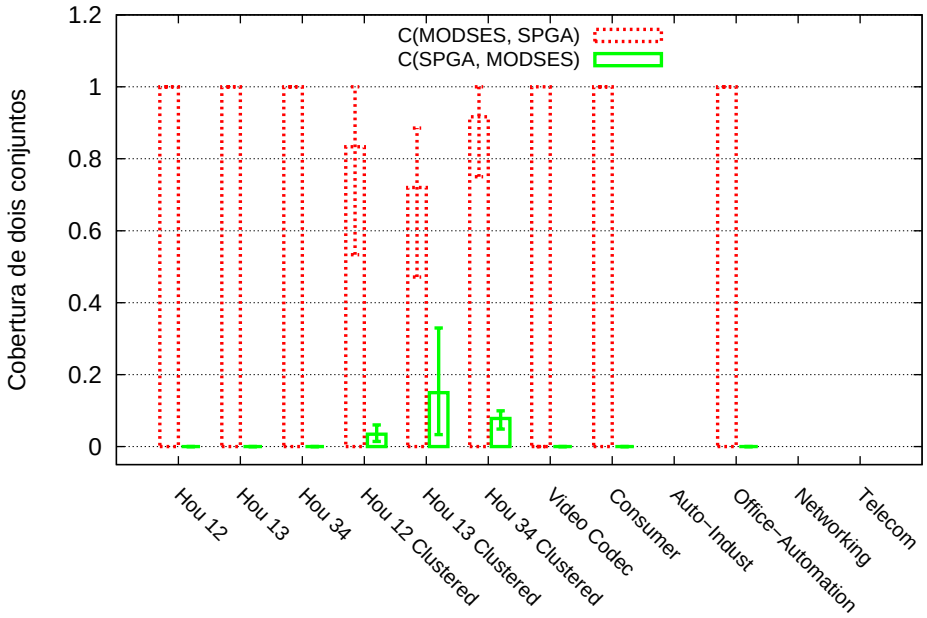
A Tabela 7.2 mostra o custo monetário e o somatório das probabilidades de violações de *deadlines* de algumas das melhores soluções achadas por EMOGAC e MODSES (colunas cinco e seis). Ela também mostra o tempo médio de execução dos algoritmos (coluna três) e o número médio de soluções visitadas durante a busca (coluna quatro). Os resultados da tabela mostram que, apesar de visitar muito mais soluções, EMOGAC é mais rápida que MODSES. EMOGAC consegue avaliar mais rapidamente uma solução candidata porque essa ferramenta assume que os tempos de execução das tarefas, assim como os intervalos de ativação dos DAGs são constantes. Isso permite que essa ferramenta construa um modelo analítico que é bastante simples de ser avaliado.

Resultados como os apresentados para o problema *Auto-Indust*, em que MODSES conseguiu encontrar uma solução com custo monetário 50% inferior a encontrada por EMOGAC (Tabela 7.2), mostram que MODSES é mais indicado que EMOGAC para o desenvolvimento de sistemas embarcados de tempo-real não críticos. A principal razão para a superioridade de MODSES nos resultados se deve ao fato de que EMOGAC sempre considera os piores cenários de execução do sistema. Portanto, probabilidades de violação de *deadlines* muitas vezes são superestimadas, fazendo com que EMOGAC fique preso em regiões com soluções de baixa qualidade. É preciso ressaltar, no entanto, que EMOGAC consegue determinar limites superiores para as probabilidades de violação de *deadlines*. Dessa forma, essa ferramenta é mais apropriada que MODSES para desenvolver sistemas embarcados de tempo-real críticos, os quais requerem maior previsibilidade.

Potência, custo e violações de *deadline*: A Tabela 7.3 mostra alguns resultados de MODSES para os problemas do *benchmark* E3S, considerando a otimização simultânea da potência consumida, custo monetário e violações de *deadlines*. As informações necessárias para avaliar a potência consumida de uma solução candidata (ver Equação (4.2)) são fornecidas pelo

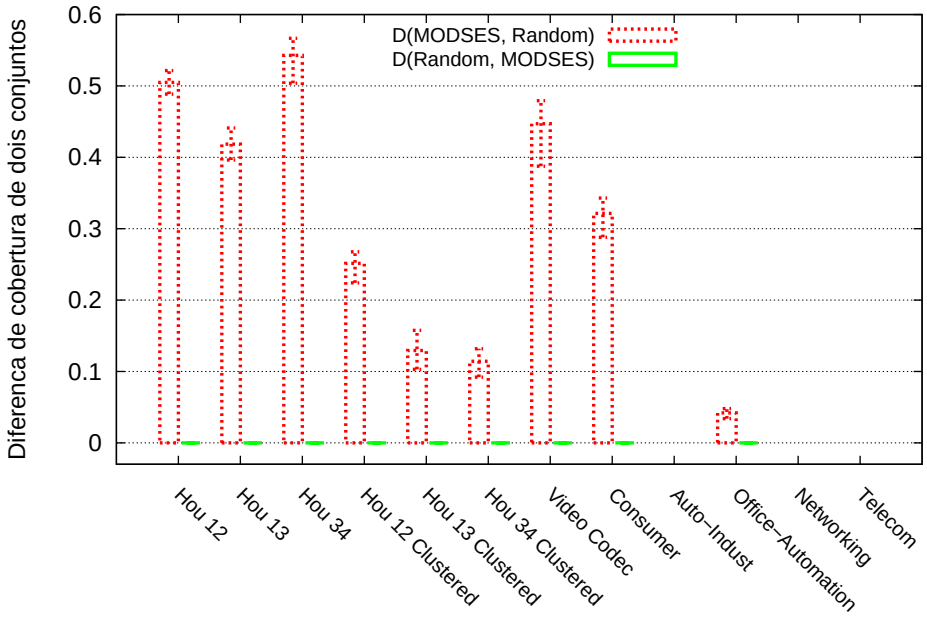


(a)

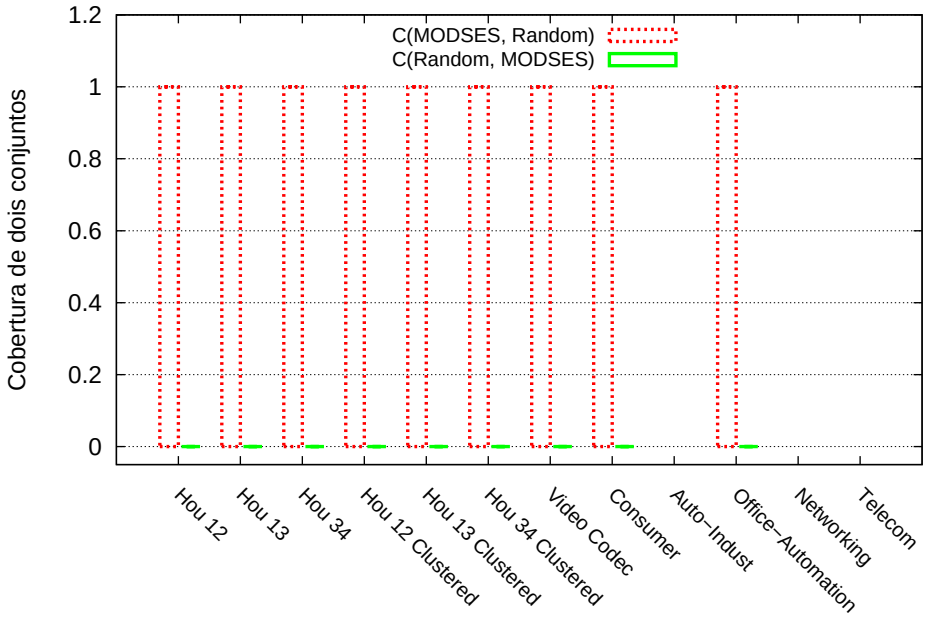


(b)

Figura 7.1: Comparação entre os algoritmos MODSES e SPGA usando as medidas D (Figura 7.1a) e C (Figura 7.1b).

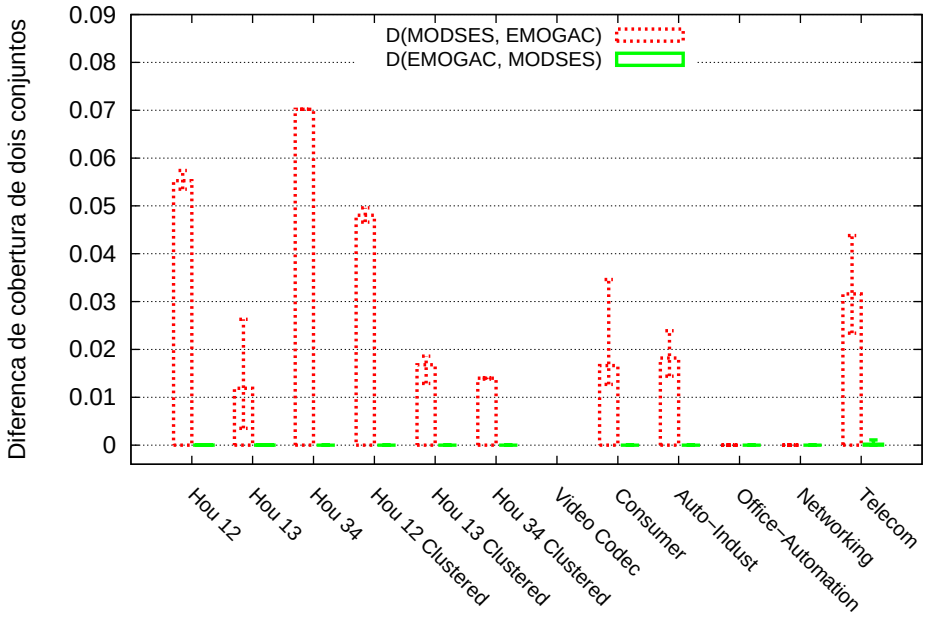


(a)

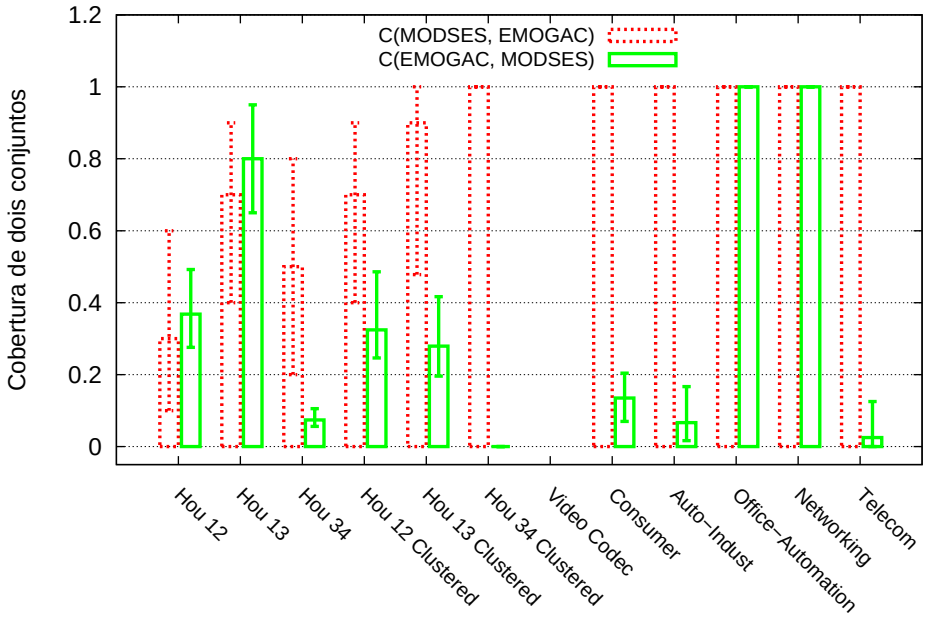


(b)

Figura 7.2: Comparação entre os algoritmos MODSES e Random usando as medidas D (Figura 7.2a) e C (Figura 7.2b).



(a)



(b)

Figura 7.3: Comparação entre os algoritmos MODSES e EMOGAC usando as medidas D (Figura 7.3a) e C (Figura 7.3b).

Problema	Algoritmo	Tempo médio (s)	# Soluções exploradas	Resultados	
				Custo ($)	Prob. violação
Hou 1&2	MODSES	745	2215	140 100	0 ± 0 $0{,}0168 \pm 0{,}0025$
	EMOGAC	41	102982	140	0
Hou 1&3	MODSES	465	2508	170 140	0 ± 0 $0{,}0052 \pm 0{,}004$
	EMOGAC	223	165583	170	0
Hou 3&4	MODSES	1238	1885	100	0 ± 0
	EMOGAC	1162	154990	140	0
Hou 1&2 Clustered	MODSES	347	2151	140 100	0 ± 0 $0{,}0265 \pm 0{,}00327$
	EMOGAC	25	69298	140	0
Hou 1&3 Clustered	MODSES	247	2810	170 140	0 ± 0 $0{,}0405 \pm 0{,}0093$
	EMOGAC	65	82950	170	0
Hou 3&4 Clustered	MODSES	424	2037	170 140 100	0 ± 0 $0{,}065 \pm 0{,}0026$ $0{,}068 \pm 0{,}0033$
	EMOGAC	94	74914	170	0
Consumer	MODSES	453	2412	101,12 65	0 ± 0 $0{,}0196 \pm 0{,}003$
	EMOGAC	72	65457	130	0
Auto-Indust	MODSES	1131	2336	75 45	0 ± 0 $0{,}00002 \pm 0$
	EMOGAC	124	102931	90	0
Office-Automation	MODSES	225	1397	65	0 ± 0
	EMOGAC	33	17781	65	0
Networking	MODSES	256	1536	52,1	0 ± 0
	EMOGAC	302	188084	52,1	0
Telecom	MODSES	2316	2811	176,2 147,32	0 ± 0 $0{,}0016 \pm 0{,}008$
	EMOGAC	118	158183	205,82	0

Tabela 7.2: Tempo médio de execução para cada problema e alguns dos melhores resultados gerados por MODSES e EMOGAC.

Problema	Tempo de execução (s)	Resultados		
		Custo ($)	Prob. de violação	Potência (mW)
Consumer	702	130	0 ± 0	214
		101,12	0 ± 0	260
		65	$0{,}02 \pm 0{,}004$	195
Auto-Indust	1036	90	0 ± 0	92
		84,62	$0{,}000009 \pm 0$	91
		81	$0{,}00002 \pm 0$	106
		45	$0{,}00006 \pm 0$	82
Office-Automation	217	65	0 ± 0	45
Networking	254	52,1	0 ± 0	48
Telecom	1669	222,4	$0 \pm 0{,}0001$	163
		210,2	$0{,}0006 \pm 0{,}0003$	1159
		210,2	$0{,}008751 \pm 0{,}005$	1065
		135,2	$0{,}01 \pm 0{,}0002$	197
		111,2	$0{,}05 \pm 0{,}004$	148

Tabela 7.3: Resultados gerados por MODSES considerando a otimização simultânea da potência consumida, custo monetário e probabilidades de violação de *deadlines* para os exemplos baseados no *benchmark* E3S.

benchmark E3S. Para cada exemplo, MODSES foi executado três vezes.

Os resultados demonstram a importância da abordagem multiobjetivo para analisar os *trade-offs* entre as diferentes decisões de projeto: existe uma grande variação entre os custos e o potências consumidas das soluções encontradas por MODSES. Por exemplo, dentre as diferentes soluções para o problema *Telecom*, os custos variaram de $ 111,2 até $ 222, a potência consumida variou de 148 mW até 1159 mW, e o somatório das probabilidades de violação de *deadlines* de 0 até 0,05.

7.1.3 C-MODSES: resultados

No conjunto de experimentos apresentados a seguir, C-MODSES é comparado a MODSES, EMOGAC e Random. Os algoritmos foram configurados para otimizar apenas custo monetário. Probabilidades de violação de *deadlines* foram modeladas como restrições a serem cumpridas, e não como um objetivo a ser otimizado. Assumiu-se que a probabilidade máxima de violação para todos os *deadlines* é de 0,05. Como MODSES trata violações de *deadlines* como um objetivo a ser otimizado, a função objetivo de MODSES que representa a minimização de violações de *deadlines* foi igualada à função objetivo adotada por C-MODSES (descrita na Equação (6.12)). Para cada problema, os algoritmos foram executados cinco vezes com sementes aleatórias variando de 1 a 5. Os parâmetros usados nos algoritmos MODSES, Random e EMOGAC foram os mesmos da Subseção 7.1.2. C-MODSES usou os mesmos parâmetros de MODSES. O *framework* Akaroa foi configurado para parar as simulações sempre que, com 95% de confiança, a metade do comprimento dos intervalos de confiança para as estimativas fossem menores ou iguais que 0,01.

A Tabela 7.4 apresenta o melhor valor e o valor médio dos custos das soluções encontradas pelos algoritmos. Essa tabela mostra que, exceto para os problemas *Networking* e *Office-Automation*, MODSES e C-MODSES foram capazes de achar soluções de melhor qualidade que EMOGAC para todos os outros problemas. A principal razão para isso foi dada

Problema	C-MODSES Melhor/Médio ($)	MODSES Melhor/Médio ($)	EMOGAC Melhor/Médio ($)	Random Melhor/Médio ($)
Hou 1&2	100/100	100/100	140/140	300/388
Hou 1&3	140/152	140/164	170/170	350/400
Hou 3&4	100/100	100/100	140/140	350/401
Hou 1&2 Clustered	100/100	100/100	140/140	210/255
Hou 1&3 Clustered	140/146	140/146	170/170	190/215
Hou 3&4 Clustered	100/100	100/100	170/170	190/220
Video codec	250/250	250/250	-	530/690
Consumer	65/65	65/65	130/130	958,07/1313,43
Auto-Indust	45/45	45/45	90/116,43	-
Office-Automation	65/65	65/65	65/65	153,12/229,434
Networking	52,1/52,1	52,1/52,1	52,1/52,1	-
Telecom	142,32/172,8	123,7/169,5	205,82/261,201	-

Tabela 7.4: Melhor valor e valor médio das soluções encontradas pelos algoritmos.

na subseção anterior: ao contrário de MODSES e C-MODSES, EMOGAC considera sempre os piores cenários de execução do sistema. Os resultados indicam ainda que a estratégia de C-MODSES, que evita desperdiçar recursos computacionais em soluções de baixa qualidade, não afetou os resultados das buscas, uma vez que MODSES e C-MODSES apresentaram desempenho muito similar.

A Figura 7.4 apresenta o tempo médio de execução dos algoritmos, em escala logarítmica. Ela mostra que C-MODSES é em média 10 vezes mais rápido que MODSES, o que comprova a eficiência da estratégia de C-MODSES para acelerar a busca. Dessa forma, caso probabilidades de violação de *deadlines* possam ser estabelecidas como restrições a serem cumpridas, e não como objetivos a serem otimizados, o algoritmo C-MODSES é mais indicado que MODSES. O leitor deve notar que os tempos de execução de MODSES para o primeiro conjunto de experimentos (apresentados na Tabela 7.2) são diferentes dos tempos de execução do segundo conjunto de experimentos (apresentados na Figura 7.4). Esta diferença ocorreu porque, como mencionado, o critério de parada para as simulações, assim como a formulação da função objetivo nos dois conjuntos de experimentos foram diferentes.

7.2 *SHOUTcast player*

O objetivo desta seção é validar a aplicação do fluxo de atividades do método proposto (Figura 1.2) em um estudo de caso real: um *SHOUTcast player* portátil. A partir deste estudo de caso, a viabilidade dos modelos de especificação, assim como a exatidão dos modelos de avaliação serão analisadas. A Figura 7.5 apresenta o esquema geral do *SHOUTcast player* (SMITH, 2007). O dispositivo recebe da internet (rádio *SHOUTcast*) um *stream* de MP3. Esse *stream* precisa ser decodificado e enviado para a interface de áudio. O usuário pode interagir com o sistema por meio de um *display* LCD e botões de pressão. O *SHOUTcast player* foi

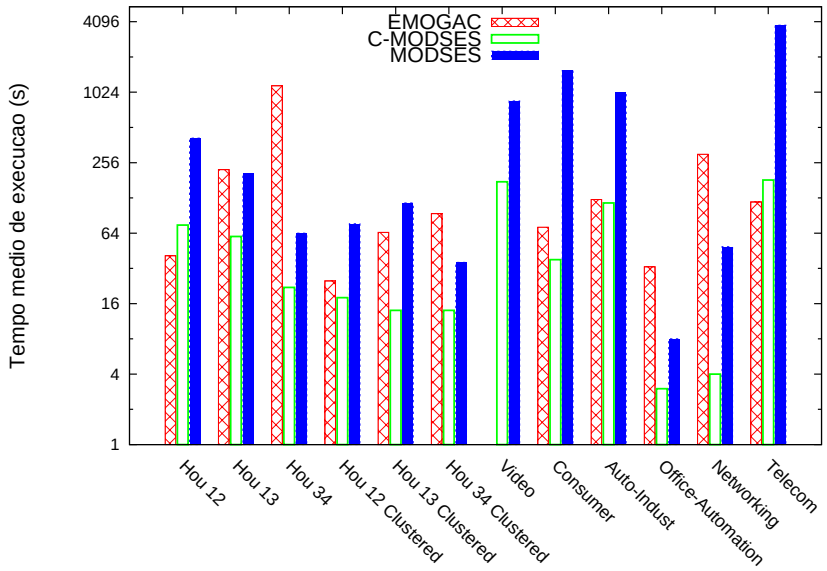


Figura 7.4: Comparação do tempo de execução de MODSES, C-MODSES e EMOGAC (escala logarítmica).

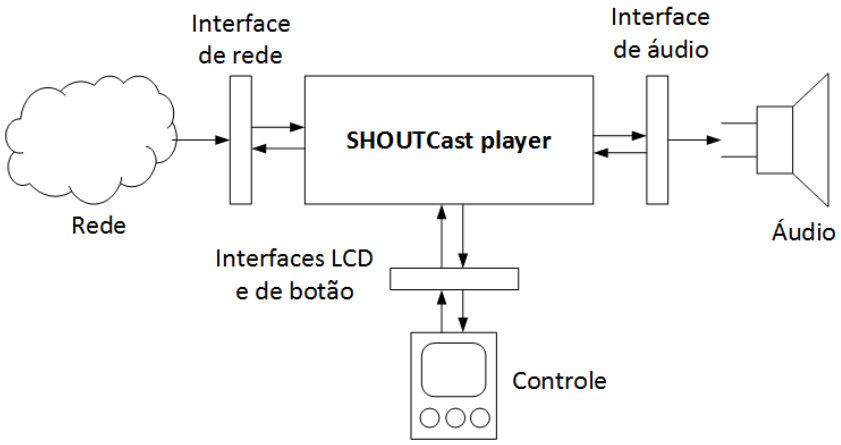


Figura 7.5: Esquema geral de um *SHOUTcast player* portátil.

escolhido como um exemplo representativo para demonstrar o método proposto porque esse sistema embarcado possui as seguintes características:

- **Tempo-real:** O sistema possui *deadlines* não críticos que precisam ser atendidos: (i) cada quadro de MP3 recebido deve ser decodificado no tempo correto, (ii) o sistema deve responder rapidamente aos comandos enviados pelo usuário, como aumentar ou baixar o volume do áudio.
- **Baixa potência consumida:** Dado que ele é um dispositivo portátil, deve consumir o mínimo possível de energia.
- **Complexidade:** É um sistema relativamente complexo, no sentido de que sua especificação funcional (protocolo de internet, decodificador de MP3, etc) é composta por milhares de linhas de código. Ademais, como o sistema pode interagir com o mundo externo de diversas formas, muitas atividades no sistema podem ocorrer em paralelo.

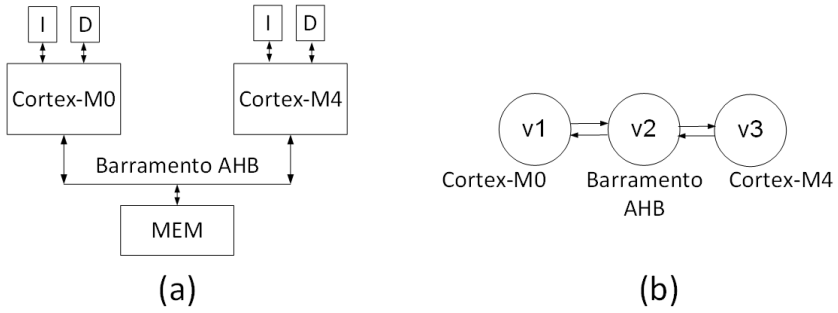


Figura 7.6: Especificação da plataforma de *hardware* LPC 4357.

As próximas subseções descrevem a aplicação de cada uma das atividade do fluxo de atividades proposto. O objetivo será explorar o espaço de projeto do *SHOUTcast player*, considerando uma plataforma heterogênea real com dois processadores. Assumimos que a plataforma de *hardware* é fixa e, portanto, o custo monetário da arquitetura também é fixo. Dessa forma, o objetivo da exploração será minimizar simultaneamente as probabilidades de violação de *deadlines* e a potência consumida.

7.2.1 Especificação

A primeira atividade do fluxo de atividades é a especificação da plataforma e da aplicação por meio dos modelos de especificação propostos (Seção 4.1).

Especificação da plataforma: Para implementar o *SHOUTcast player*, a plataforma de *hardware* NXP LPC4357 (SEMICONDUCTORS, 2014) foi adotada. Esta plataforma é voltada para dispositivos em que baixo consumo de energia e custo monetário são requisitos primordiais. Essa plataforma de *hardware* possui dois processadores heterogêneos: ARM Cortex-M0 (YIU, 2011) e ARM Cortex-M4 (YIU, 2013). O processador Cortex-M0 tem uma arquitetura mais simples e consome menos energia em comparação ao Cortex-M4. Porém, o processador Cortex-M4 tem melhor desempenho (YIU, 2013).

A Figura 7.6a apresenta uma visão simplificada da arquitetura do LPC 4357. Os processadores se comunicam por meio do barramento AHB. As mensagens ficam armazenadas na memória compartilhada associada ao barramento AHB (ver caixa *MEM*). Além disso, cada processador possui sua memória de dados e instrução (ver caixas *I* e *D*). A Figura 7.6b apresenta o ATG definido para representar esta plataforma. O ATG é composto por três vertices: v_1, v_2 e v_3. Os vertices v_1 e v_3 representam os processadores, e o vértice v_2 representa o barramento AHB responsável pela comunicação entre os dois processadores.

Especificação da aplicação: A especificação funcional da aplicação é composta pelos seguintes módulos: (i) protocolo de internet, (ii) decodificador dos *streams* de MP3, e (iii) interface gráfica/botões. A implementação do protocolo de internet e do decodificador de MP3 se baseou, respectivamente, nas seguintes implementações de código aberto: lwIP (MANSLEY, 2014) e Helix MP3 (INC, 2014). Para implementar a interface gráfica, a biblioteca SWIM (SEMICONDUCTORS, 2011) foi adotada. A Figura 7.7 apresenta o protótipo de interface gráfica desenvolvido sendo exibido no *display* LCD do dispositivo real, e a Figura 7.7 apresenta a placa MCB 4357, que foi adotada para implementar o protótipo do *SHOUTcast player*.

A Figura 7.9 apresenta o HSDG do *SHOUTcast player*, definido a partir de sua especificação funcional. Esse HSDG é formado por três componentes: (i) decodificador de MP3 (Figura 7.9a), (ii) tratamento de comandos do usuário (Figura 7.9b), e (iii) conexão com a internet (Figura 7.9c). A biblioteca proposta para dar suporte ao desenvolvimento de aplicações multiprocessadas

Figura 7.7: Protótipo de interface gráfica desenvolvido para o *SHOUTcast player* sendo exibida no *display* LCD do dispositivo.

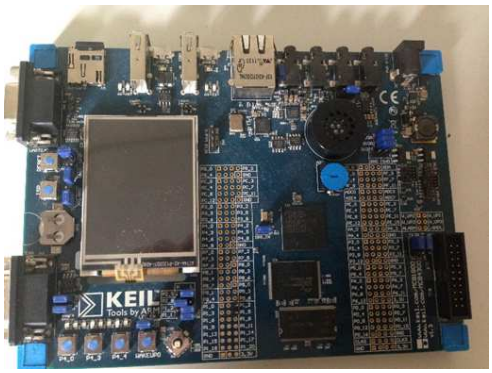


Figura 7.8: Placa de desenvolvimento MCB4357, que foi adotada para implementar o protótipo do *SHOUTcast player*.

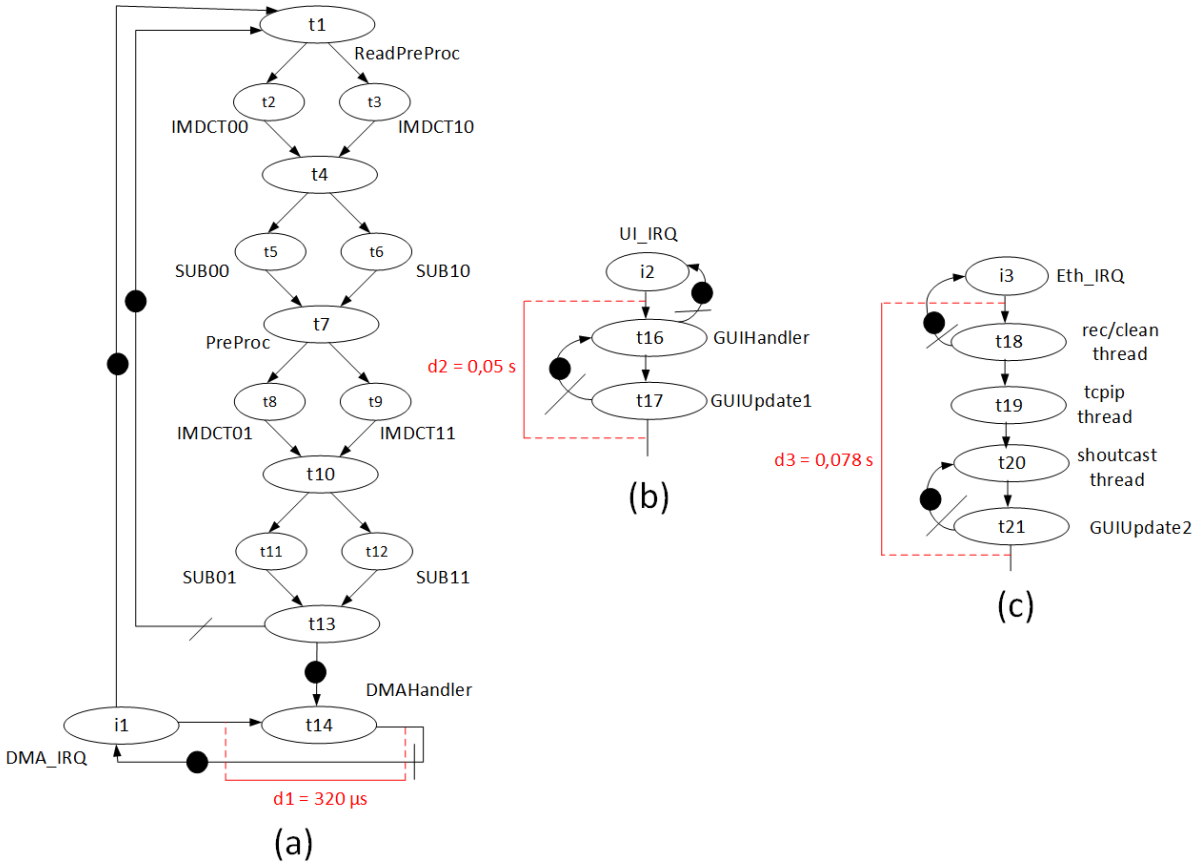


Figura 7.9: HSDG do *SHOUTcast player*, que é formado pelos seguintes componentes decodificador de MP3 (Figura 7.9a), tratamento de comandos do usuário (Figura 7.9b), e conexão com a internet (Figura 7.9c).

foi usada para implementar a especificação funcional do *SHOUTcast player* (Seção 4.2).

Antes de definir o HSDG do componente de decodificação de MP3 (Figura 7.9a), a especificação sequencial original do decodificador Helix MP3 foi reestruturada para que ela tivesse sua concorrência exposta, conforme descrito na Figura 4.6. A Figura 7.10a apresenta, de maneira simplificada, a especificação sequencial original do decodificador e como essa especificação foi mapeada na especificação concorrente. Como a decodificação *stereo* dos canais de áudio esquerdo e direito pode ser feita independentemente, a concorrência foi explorada nesse sentido. As tarefas representadas pelos atores t_1, t_2, ... , t_{13} representam o fluxo de atividades que deve ser executado para decodificar um quadro de MP3. A estrutura do HSDG indica que, por exemplo, as tarefas representadas pelos atores t_2 e t_3 podem executar em paralelo, e a tarefa representada pelo ator t_4 só pode executar depois que as tarefas representadas por t_2 e t_3 terminarem suas execuções. A interface entre o decodificador e o dispositivo de áudio é feita por meio de um controlador de DMA (*Direct memory access*). Sempre que o decodificador termina a decodificação de um quadro, o quadro decodificado é armazenado em um *buffer*. O controlador de DMA é responsável por copiar o quadro decodificado e enviá-lo para o dispositivo de áudio. Quando o controlador de DMA termina o envio, uma interrupção é gerada e o controlador de DMA deve ser reprogramado para fazer um novo envio. Na Figura 7.9a, o ator t_{14} representa a atividade de reprogramação do DMA, e o ator i_1 representa a geração de interrupções pelo controlador de DMA. Caso não exista no canal representado pelo arco (t_{13}, t_{14}) um quadro pronto para ser enviado depois da interrupção, o controlador de DMA fica impedido de fazer o envio, e

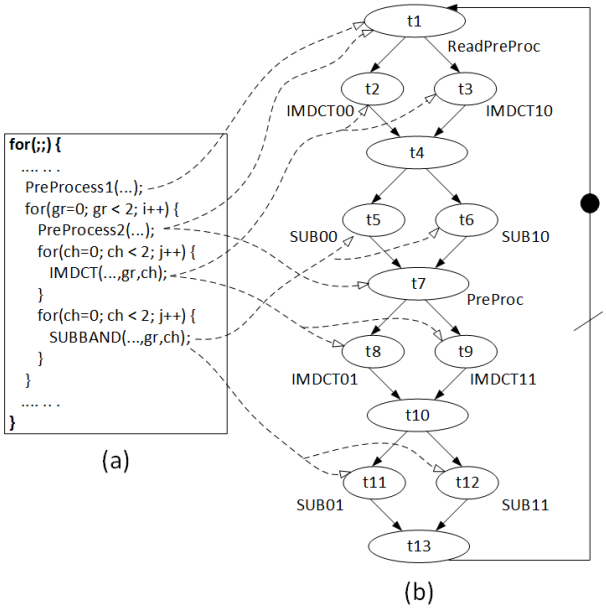


Figura 7.10: (a) Especificação sequencial do decodificador Helix MP3. (b) HSDG da especificação concorrente do decodificador Helix MP3.

o áudio para de tocar. Dessa forma, o *deadline* d_1 da Figura 7.9a corresponde ao intervalo de tempo máximo aceitável entre a interrupção gerada pelo controlador de DMA e a reprogramação do controlador de DMA.

O ator i_2 do componente de tratamento de comandos do usuário (Figura 7.9b) representa as interrupções geradas quando o usuário aperta algum botão da interface. O ator t_{16} representa a tarefa que faz o tratamento dessa interrupção, e o ator t_{17} representa a tarefa que atualiza o áudio e a interface gráfica depois do comando recebido. O *deadline* d_2 define o intervalo de tempo máximo aceitável entre a inserção de um novo comando pelo o usuário, e a atualização do sistema/interface. Pesquisas sobre usabilidade de interfaces indicam que tempos de resposta abaixo de 0,1 s são imperceptíveis ao usuário (NIELSEN, 1994). Dessa forma, consideramos que o *deadline* d_2 é igual a 0,05 s.

O ator i_3 do componente de conexão com a internet (Figura 7.9c) representa as interrupções geradas quando um novo pacote de dados é recebido. O ator t_{18} corresponde à tarefa que faz o tratamento dessa interrupção, e os atores t_{19} e t_{20} representam as tarefas de tratamento do pacote de dados recebido. Por último, o ator t_{21} representa a tarefa de atualização da interface gráfica que ocorre após o recebimento de um pacote.

7.2.2 Caracterização

A segunda atividade do fluxo proposto é a atividade caracterização (Seção 4.3), cujo objetivo é determinar informações sobre tempo de execução das tarefas, o custo monetário e a potência consumida pelos elementos da plataforma de *hardware*. Como o objetivo é minimizar as probabilidades de violação de *deadlines* e a potência consumida, é preciso definir: (i) os *deadlines* da aplicação, (ii) as distribuições de probabilidade dos intervalos de tempo entre a geração das interrupções no sistema (recordar a definição das variáveis aleatórias IT_i na Subseção 4.3.2), (iii) o tamanho das mensagens de cada canal de comunicação, (iv) as distribuições de probabilidade dos tempos de execução das tarefas, (v) o *overhead* do sistema operacional, e (vi) a potência consumida dos elementos *hardware*.

***Deadlines* e distribuições de probabilidade dos intervalos de tempo entre a geração das interrupções:** Os *deadlines* da aplicação foram definidos na subseção anterior (ver d_1, d_2, e d_3 na Figura 7.9). Para que o áudio não pare de tocar, o controlador de DMA deve ser reprogramado a cada 36,1 ms. Assim, este intervalo de tempo foi associado ao ator i_1, que representa as interrupções geradas pelo controlador de DMA. Assumiu-se que, no pior caso, o intervalo de tempo entre as interações do usuário com a interface varia uniformemente entre 3 e 19 ms. Dessa forma, essa distribuição de probabilidade foi usada para descrever o intervalo entre a geração das interrupções representadas pelo ator i_2. Para definir a distribuição de probabilidade do intervalo de tempo entre as chegadas de pacotes de dados pela internet (representado pelo ator i_3), um *trace* com intervalos reais de tempo de chegada de pacotes foi usado. Para gerar este *trace*, consideramos um intervalo de meia hora de medição. A partir do *trace*, os algoritmos de exploração automaticamente constroem a distribuição de probabilidade empírica (ROBINSON, 2004), como descrito na Seção 4.3.

Distribuições de probabilidade dos tempos de execução das tarefas: Assim como a distribuição de probabilidade dos intervalos de chegada de pacote, as distribuições de probabilidade relacionadas a tempos de execução das tarefas foram definidas por meio de *traces* reais de execução do sistema (ver Figuras 4.14 e 4.15). Os *traces* foram gerados considerando entradas representativas da aplicação. Em particular, para obter os *traces* das tarefas que decodificam um *stream* de MP3, uma rádio *SHOUTcast* foi escolhida e por um intervalo de tempo correspondente a meia hora, os dados enviados por essa rádio foram arquivados para, posteriormente, serem usados na caracterização. Os algoritmos automaticamente definem distribuições de probabilidade empírica para os tempos de execução a partir dos *traces* fornecidos.

Tamanho das mensagens: É necessário também determinar o tamanho (em *bytes*) de cada mensagem nos canais de comunicação. Esta informação, junto com os tempos das fases de cópia de mensagens e escrita de mensagens, é utilizada para determinar o nível de utilização do barramento (Seção 5.2).

***Overhead* do sistema operacional:** O *overhead* do sistema operacional compreende o tempo médio para mudança de contexto do sistema operacional FreeRTOS, que é o sistema operacional adotado neste trabalho, e o tempo médio para executar a tarefa que trata o recebimento de mensagens interprocessador. Esses tempos foram obtidos por meio de medição no sistema real, como descrito na Subseção 4.3.2.

Potência consumida: A Equação (7.1) foi adotada para determinar a potência média consumida pelos processadores da plataforma LPC 4357 (recordar a Equação (4.2)):

$$f_2(\mu_1, \mu_2, \mathfrak{f}) = b_0 + \mu_1 \times b_1 + \mu_2 \times b_2 + \mathfrak{f} \times b_3, \tag{7.1}$$

onde μ_1 e μ_2 denotam, respectivamente, a utilização dos processadores Cortex-M0 e Cortex-M4, $\mathfrak{f}$ denota a frequência de operação da plataforma, e b_0, b_1, b_2 e b_3 foram os coeficientes determinados durante a atividade de caracterização, usando regressão linear múltipla.

A Figura 7.11 apresenta os dados obtidos por medição na plataforma, usando o esquema de medição apresentado na Seção 4.3.1. Cada ponto nessa figura representa a potência consumida pela plataforma para um dada combinação de frequência de operação, utilização do processador Cortex-M0, e utilização do processador Cortex-M4. A LPC 4357 suporta frequências de operação que vão de 120 até 204 MHz. A linha 1 da Figura 7.11 mostra a potência consumida pela plataforma para diferentes frequências de operação quando ambos os processadores estão em estado ocioso (sem executar nenhuma tarefa). A linha 2 exibe a potência consumida pela plataforma quando o processador Cortex-M4 está sendo 100% utilizado e o outro processador não executa nada. A linha 3 exibe a situação inversa, em que o processador Cortex-M0 está sendo 100% utilizado e o processador Cortex-M4 não executa nada. Finalmente,

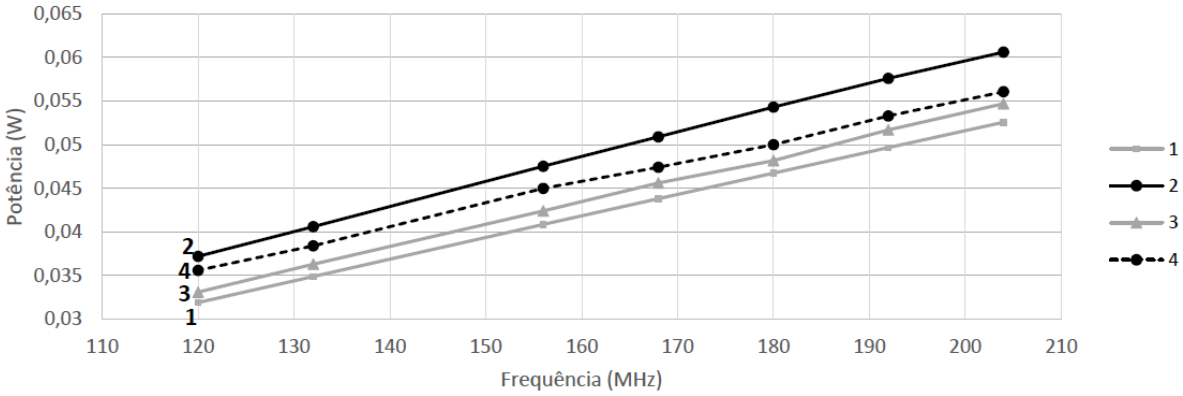


Figura 7.11: Potência consumida pela plataforma LPC 4357, considerando diferentes utilizações dos processadores e frequências de operação.

a linha 4 representa a potência consumida pela plataforma quando ambos os processadores estão executando alguma coisa. Para gerar essas curvas, códigos de caracterização para estimular diferentes níveis de utilização nos processadores foram criados, como descrito na Seção 4.3.1. A função $Workload\_Function()$ do código de caracterização (Figura 4.17) foi especificamente desenvolvida para que fizesse várias chamadas às funções do decodificador de MP3, que é a parte do código mais executada da aplicação.

Na Equação (7.1), a potência consumida é dada em watts e ƒ em megahertz. Os valores obtidos para b_0, b_1, b_2 e b_3, utilizando regressão linear múltipla foram, respectivamente, -0,000192; 0,004; 0,0049; e 0,000263. O valor da medida estatística R^2 foi igual a 96%, o que mostra a boa aderência da Equação (7.1) aos dados da Figura 7.11.

7.2.3 Alocação, mapeamento e atribuição de prioridades

Nesta atividade, o espaço de projeto é explorado usando um dos dois algoritmos propostos. O algoritmo escolhido otimizará a alocação, mapeamento e atribuição de prioridades e frequência do sistema (Seção 6.1).

Como as probabilidades de violação de *deadlines* devem ser minimizadas, o algoritmo MODSES foi escolhido para explorar o espaço de projeto. Caso elas pudessem ser tratadas como restrições a serem cumpridas, então o algoritmo C-MODSES seria uma escolha melhor, pois ele executa mais rapidamente.

7.2.4 Avaliação

A avaliação é conduzida automaticamente pelos algoritmos de exploração. No presente estudo de caso, cada solução candidata produzida por MODSES é automaticamente avaliada em termos de potência consumida e probabilidades de violação de *deadlines*.

Para analisar a exatidão dos modelos de simulação propostos, alguns experimentos foram conduzidos para comparar os valores estimados através dos modelos de simulação P-DEVS e os valores reais obtidos por medição com o sistema em funcionamento. Nesses experimentos, as seguintes medidas de desempenho foram comparadas: o tempo médio de decodificação de um quadro de MP3, e a probabilidade de violação do *deadline* d_1. O tempo médio de decodificação corresponde ao tempo médio necessário para executar o fluxo de atividades representado pelos atores $t_1, t_2, \ldots, t_{13}$. Cinco medições foram feitas para cada medida de desempenho analisada, e cada medição compreendeu um intervalo de cinco minutos de funcionamento do sistema real.

Diferentes cenários de mapeamento/atribuição de prioridades foram testados. Para verificar o comportamento do sistema em função da interação do usuário com a interface, em todos os cenários testados, o intervalo de tempo médio em que as interrupções representadas pelo ator i_2 são geradas foi variada de 11 até 120 ms. Uma tarefa extra com tempo de execução desprezível foi adicionada à especificação para gerar estas interrupções. Os experimentos foram divididos em dois conjuntos. No primeiro conjunto de experimentos, os sistemas operacionais que executam em cada processador foram configurados para não permitir preempção de tarefas e, no segundo conjunto, os sistemas operacionais foram configurados para permitir preempção.

A Figura 7.12 apresenta os resultados do primeiro conjunto de experimentos, em que três cenários diferentes de mapeamento/atribuição de prioridades são analisados. No primeiro cenário (Figura 7.12a), as tarefas representadas pelos atores t_3, t_6, t_{18}, t_{19} e t_{20} foram mapeadas no processador Cortex-M0, e as outras tarefas no processador Cortex-M4. Para evitar ao máximo violações do *deadline* d_1, a tarefa representada pelo ator t_{14} recebeu a maior prioridade dentre as tarefas da aplicação. Nós escolhemos analisar este cenário de porque ele representa a situação em que comunicações interprocessador ocorrem frequentemente. Assim, além de avaliar a capacidade dos modelos em representar os processadores e os sistemas operacionais, avaliamos também a exatidão do modelo de comunicação. Os resultados mostram que a probabilidade de violação do *deadline* d_1 é alta no primeiro cenário. Com o intuito de diminuir essa probabilidade, no segundo cenário (Figura 7.12b), as prioridades das tarefas de decodificação (Figura 7.9a) foram aumentadas. Os resultados mostram que, embora o tempo médio de decodificação de um quadro tenha diminuído, a probabilidade de violação permanece alta. No terceiro cenário (Figura 7.12c), a tarefa t_{14} foi movida para o processador Cortex-M0. Os resultados desse cenário mostram que o tempo médio de decodificação praticamente não se altera, porém, a probabilidade de violação reduziu significativamente.

No segundo conjunto de experimentos (Figura 7.13), os mesmos cenários descritos acima foram analisados. No entanto, agora os sistemas operacionais dos processadores foram configurados para permitir preempção. A Figura 7.13 mostra que, para todos os cenários, as probabilidades de violação *deadline* são iguais a zero e o modelo estimou corretamente que elas seriam iguais a zero. Como a tarefa t_{14} tem a maior prioridade dentre as tarefas da aplicação, sempre que ela fica pronta para executar, o sistema operacional interrompe (preempção) a tarefa executando no momento para t_{14} execute. Isto faz com que, para os cenários avaliados, o *deadline* d_1 nunca seja violado quando preempções estão autorizadas.

Experimentos foram feitos também para avaliar a exatidão do modelo de potência consumida, apresentado na Equação (7.1). Dois conjuntos de experimentos foram feitos. A Figura 7.14a apresenta os resultados do primeiro conjunto de experimentos, em que analisamos a potência consumida do mapeamento/atribuição de prioridades correspondente ao cenário 1 das Figuras 7.12 e 7.13. Como não verificamos diferenças na potência consumida entre o caso em que preempções são permitidas e o caso em que elas não são permitidas, apenas a Figura 7.14a é apresentada. A Figura 7.14b mostra os resultados do segundo conjunto de experimentos, em que consideramos o mesmo cenário, porém, fixamos a taxa média de geração das interrupções representas por i_2 e variamos a frequência de operação da plataforma.

Como pode ser observado nos resultados apresentados nessa seção, os erros nas estimativas geradas pelos modelos são muito pequenos. Para todos os experimentos realizados, este erro foi sempre menor que 5%. Dessa forma, pode-se concluir que não há evidências para refutar a hipótese de que os modelos desenvolvidos representam uma boa aproximação do sistema real.

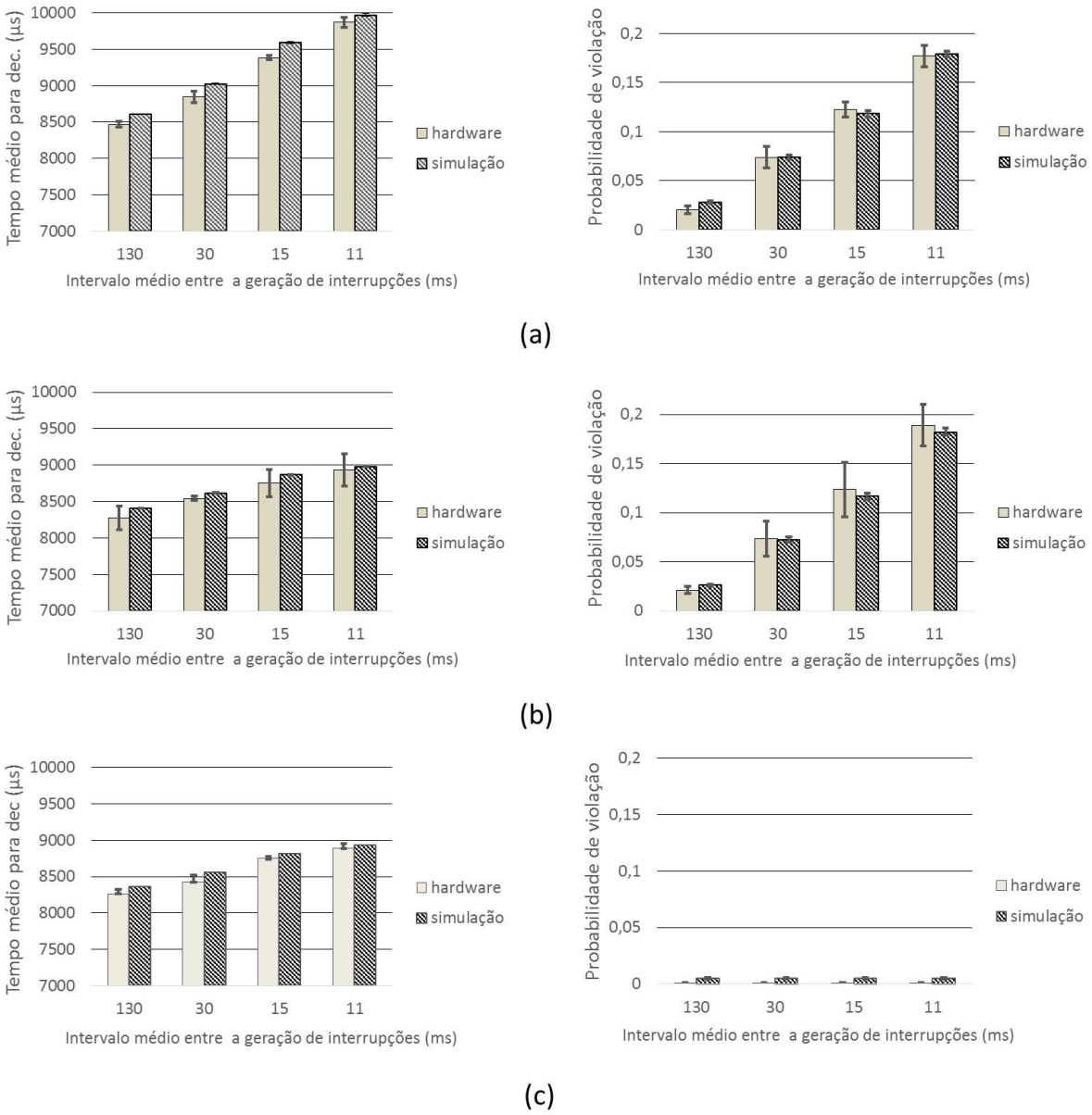


Figura 7.12: Tempo médio de decodificação de um quadro, e probabilidade de violação do *deadline* de decodificação, considerando que tarefas não podem sofrer preempção: (a) Cenário 1, (b) Cenário 2, e (c) Cenário 3.

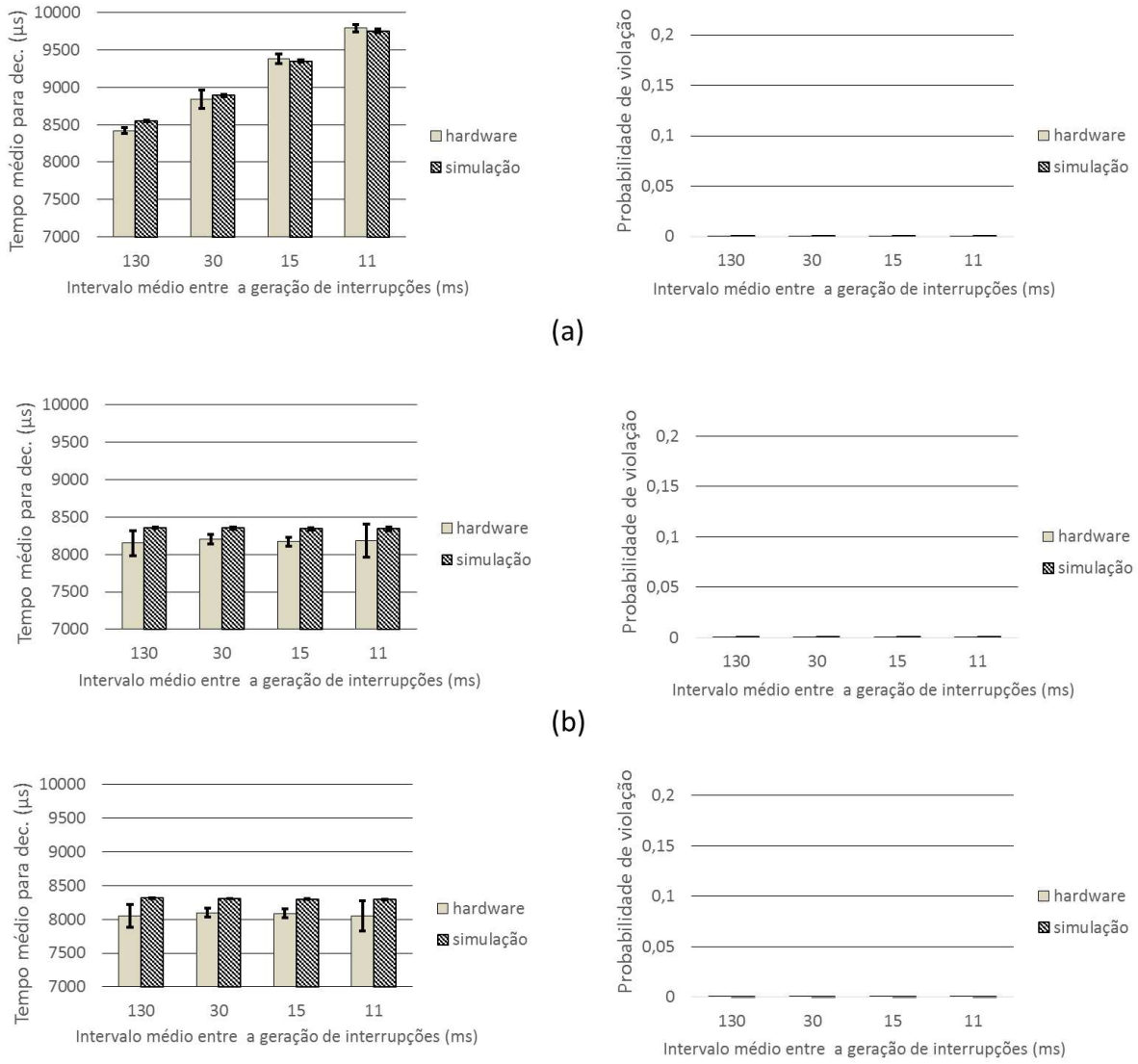


Figura 7.13: Tempo médio de decodificação de um quadro, considerando que tarefas podem sofrer preempção: (a) Cenário 1, (b) Cenário 2, (c) Cenário 3.

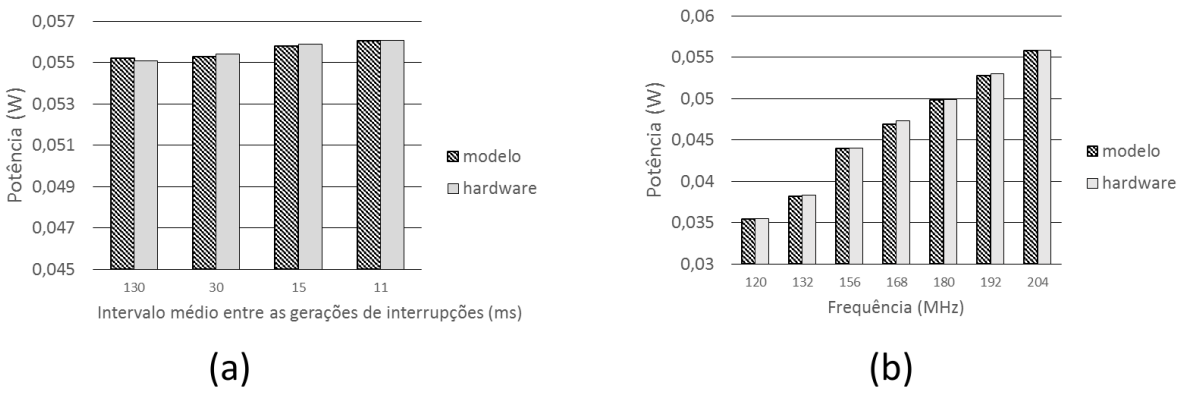


Figura 7.14: Comparação entre potência consumida estimada pelo modelo e a medida no *hardware*.

	Potência (mW)	Prob. de violação	# Processadores	# Barramentos	Frequência (MHz)
Sem preempção	34,807	0	2	1	120
	34,805	0,02	2	1	120
Com preempção	34,793	0	1	0	120

Tabela 7.5: Melhores soluções encontradas durante a exploração do espaço de projeto do *SHOUTcast player*.

7.2.5 Implementação

A última atividade do fluxo proposto consiste em selecionar uma das soluções geradas pelos algoritmos propostos e implementá-la. Neste estudo de caso, inicialmente, MODSES foi executado considerando que preempções de tarefas não podem ocorrer e, depois, considerando que elas podem ocorrer. A Tabela 7.5 apresenta as melhores soluções encontradas para os dois casos. Essa tabela mostra que, para ambos os casos, MODSES encontra soluções em que a probabilidade de violação de *deadlines* é zero. A frequência de operação da plataforma nas soluções encontradas é a menor possível (120 MHz). Em particular, é possível notar a importância de uma abordagem multiobjetivo observando as soluções encontradas para o caso em que preempções não são suportadas. Conhecendo os *trade-offs* das soluções, dificilmente um projetista escolheria a solução que viola 2% *deadlines* no lugar da solução em que essa probabilidade é zero, uma vez que a diferença de potência consumida entre as duas soluções é quase insignificante. Os resultados demonstram o potencial do método proposto para atacar problemas reais e complexos, pois: (i) foi possível especificar e caracterizar o sistema usando os modelos e métodos propostos, (ii) soluções de boa qualidade foram encontradas, isto é, soluções com a menor frequência de operação possível e sem violar *deadlines*, e (iii) os modelos de avaliação demonstraram boa exatidão (erro máximo menor que 5%).

7.2.6 Cenários

O método proposto permite que diversos cenários de exploração sejam rapidamente avaliados, bastando mudar as especificações da plataforma ou da aplicação. Isso pode ser feito alterando as linhas correspondentes do arquivo XML que MODSES usa como entrada. A seguir, serão mostrados diferentes exemplos de cenários de exploração que podem ser analisados. Para esses cenários, consideramos que preempção de tarefas é permitido.

Cenário 1: A interface gráfica apresentada na Figura 7.7 é apenas um protótipo. Para a versão final do produto, possivelmente uma interface gráfica mais elaborada e que demande mais recursos computacionais deverá ser desenvolvida. Nesse sentido, seria interessante verificar qual a potência consumida nas soluções encontradas por MODSES, caso os tempos de execução das tarefas que gerenciam a interface gráfica aumentem. Verificamos as situações em que esses tempos aumentam 50%, 100% e 200%. A Figura 7.15 apresenta os resultados dessa análise. Para cada situação de aumento, o valor de potência consumida mostrado na figura corresponde à menor potência consumida dentre as soluções que violam menos que 1% dos *deadlines*. Comparando os valores desse gráfico com o valor de potência consumida da última solução apresentada na Tabela 7.5, é possível notar que aumentos de 50%, 100% e 200% resultam em aumentos de 2%, 5% e 40% na potência consumida pelas soluções encontradas por MODSES.

Cenário 2: Embora a plataforma atual não permita frequências menores que 120 MHz, seria interessante analisar até quanto a potência consumida poderia ser diminuída caso frequências menores que 120 MHz fossem possíveis. Dessa forma, a especificação foi modificada para

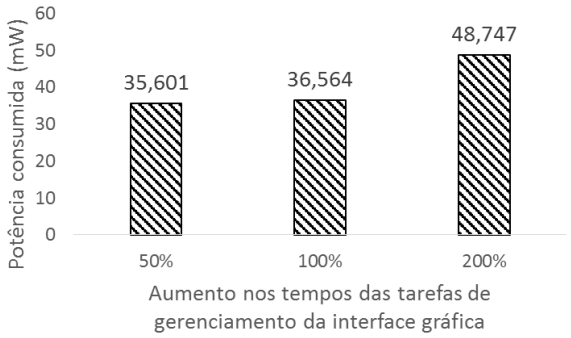


Figura 7.15: Potência consumida em função dos aumentos nos tempos de execução das tarefas que gerenciam a interface gráfica (Cenário de exploração 1).

Potência (mW)	Prob. de violação	# Processadores	# Barramentos	Frequência (MHz)
32,105	0	2	1	108
26,926	0,0002	2	1	84
24,584	0,0331	2	1	72

Tabela 7.6: Algumas das melhores soluções encontradas para o *SHOUTcast player*, caso a frequência de operação da plataforma possa ir até 60 MHz (Cenário de exploração 2).

que 5 novas possibilidades de frequência pudessem ser selecionadas. Com essa modificação, consideramos que a frequência pode ir de 60 até 204 MHz e que a Equação (7.1) também é valida para as novas frequências. A Tabela 7.6 apresenta algumas das melhores soluções encontradas. Essa tabela mostra que MODSES foi capaz de encontrar uma solução que adota frequência de 108 MHz e não viola nenhum *deadline*. Caso o projetista esteja disposto a aceitar violações de até 3,3%, então ele pode escolher a solução que adota frequência de 72 MHz. Os resultados indicam que uma plataforma com a mesma configuração de processadores e que suporte frequência menores que 120 MHz pode implementar mais eficientemente o *SHOUTcast player*.

Cenário 3: Adicionando novos elementos de *hardware* na especificação, é possível analisar, por exemplo, qual seria o impacto no desempenho da plataforma caso um processador mais rápido fosse incluído. No último cenário analisado, a especificação foi modificada para que uma plataforma composta por dois processadores Cortex-M4 fosse adotada, ao invés da plataforma original com um processador Cortex-M4 e um processador Cortex-M0. Para analisar esse cenário, consideramos que $b_1 = b_2$ na Equação (7.1), já que os dois processadores são iguais. Além disso, consideramos também que a frequência pode ir até 60 Mhz, como no cenário anterior. A Tabela 7.7 apresenta algumas das melhores soluções encontradas por MODSES. Os resultados mostram que, para essa nova configuração, MODSES é capaz de encontrar soluções em que a arquitetura adota uma frequência de 60 MHz sem que nenhum *deadline* seja violado.

Potência (mW)	Prob. de violação	# Processadores	# Barramentos	Frequência (MHz)
22,438	0,0003	2	1	60
22,453	0,0001	2	1	60
22,755	0	2	1	60

Tabela 7.7: Algumas das melhores soluções encontradas para o *SHOUTcast player* caso a frequência de operação da arquitetura possa ir até 60 MHz e a plataforma seja composta por dois processadores Cortex-M4 (Cenário de exploração 3).

Para explorar o espaço de projeto do *SHOUTcast player*, MODSES demorou em média 10 minutos para cada execução. Dessa forma, em menos de duas horas todos os cenários acima foram analisados (incluindo o tempo para modificar o XML de entrada de MODSES), o que mostra a eficiência do método proposto.

7.3 Considerações finais

Este capítulo apresentou o conjunto de experimentos concebidos para analisar a viabilidade e eficiência do método proposto. Inicialmente, os algoritmos de exploração desenvolvidos foram aplicados a problemas baseados em *benchmarks* disponíveis na literatura. As soluções produzidas pelos algoritmos propostos foram comparadas às soluções produzidas por algoritmos similares. Os resultados da comparação indicam que: (i) os algoritmos propostos conseguem achar soluções de melhor qualidade que algoritmos baseados em cenários de pior caso, dessa forma, eles são mais indicados para o desenvolvimento de sistemas embarcados de tempo-real não críticos; (ii) para o tipo de problema tratado pelo método proposto, os algoritmos desenvolvidos são superiores a um algoritmo representativo do estado da arte dos algoritmos genéticos multiobjetivo, SPGA; e (iii) dado que os modelos de simulação propostos são livres de detalhes funcionais, os algoritmos propostos são capazes de achar soluções de boa qualidade em um curto espaço de tempo. No final deste capítulo, o fluxo de atividades do método proposto foi aplicado no desenvolvimento de um sistema embarcado real: um *SHOUTcast player*. Esse estudo de caso demonstrou que: (i) os modelos de avaliação propostos possuem boa exatidão (erro máximo $<$ 5%); e (ii) o método proposto é adequado para explorar o espaço de projeto de sistemas que possuem especificações funcionais relativamente complexas.

8 Conclusão

Para atender aos requisitos de um novo projeto de sistema embarcado, frequentemente, os projetistas usam o conhecimento adquirido em projetos anteriores para restringir o espaço de projeto e selecionar a melhor solução a partir de um conjunto pequeno de opções. Porém, devido ao aumento da complexidade dos sistemas embarcados, essa abordagem *ad-hoc* tem se tornado muito propensa a erros e lenta. Dessa forma, atualmente é essencial dispor de métodos e ferramentas para avaliar as possíveis alternativas de projeto e determinar a arquitetura que melhor atende aos objetivos do projeto.

Este trabalho apresentou um método para exploração do espaço de projeto de sistemas embarcados de tempo-real não crítico. O principal objetivo do método proposto é prover meios para que o projetista possa implementar uma aplicação em uma arquitetura com processadores programáveis e heterogêneos, considerando as seguintes restrições (conflitantes) de projeto: probabilidades de violação de *deadlines*, potência consumida e custo monetário.

O método proposto contempla um conjunto de atividades e um ambiente integrado, composto por modelos, biblioteca de funções, mecanismos de caracterização e algoritmos de exploração, para que o projetista possa eficientemente explorar o espaço de projeto. A primeira atividade do método proposto consiste em definir a plataforma de *hardware* e a aplicação usando os modelos de especificação propostos. Durante essa atividade, a biblioteca de funções concebida por este trabalho é usada para garantir que o código funcional da aplicação obedeça a semântica dos modelos de especificação. Na segunda atividade do método proposto, o projetista deve fornecer informações sobre o tempo de execução das tarefas da aplicação, e o custo monetário/potência consumida dos elementos de *hardware* da plataforma. Tais informações são capturadas usando os mecanismos de caracterização propostos por este trabalho. Na terceira atividade do método, um dos algoritmos de exploração desenvolvidos nesta tese (MODSES ou C-MODSES) é usado para explorar o espaço de projeto. Esses algoritmos executam automaticamente três subatividades: alocação da arquitetura, mapeamento das tarefas e escalonamento (atribuição de prioridades às tarefas). Quando o critério de parada dos algoritmos propostos é atingido, eles retornam um subconjunto representativo do conjunto ótimo de Pareto (ou uma aproximação dele). As soluções candidatas geradas pelos algoritmos são avaliadas em termos de custo monetário, potência consumida e probabilidade de violação de *deadlines*. Para estimar as probabilidades de violação de *deadlines* das soluções candidatas, modelos formais DEVS (*Discrete Event System Specification*) são construídos e analisados usando simulação estacionária. Os modelos DEVS representam simultaneamente a aplicação, os processadores, e os sistema operacionais de tempo-real que executam em cada processador. Técnicas do estado da arte são usadas para a analisar os resultados da simulação e determinar quando ela deve parar.

Diversos experimentos foram conduzidos para demonstrar a viabilidade do método proposto. Em particular, a exploração do espaço de projeto considerando uma plataforma de *hardware* real e uma aplicação real (*SHOUTcast player*) foi demonstrada. Os resultados mostram a boa exatidão dos modelos de desempenho desenvolvidos (erro máximo de 5%, em comparação

a medições em um sistema real), e a eficiência do método proposto em encontrar soluções de boa qualidade para especificações que os métodos existentes têm dificuldade em explorar.

Apesar de na literatura existirem muitos métodos para exploração do espaço de projeto de sistemas embarcados de tempo-real não críticos, os métodos existentes possuem muitas limitações em relação aos modelos adotados, das quais as principais são:

- Grande parte dos métodos considera que os tempos de execução das tarefas são constantes (normalmente iguais ao WCET da tarefa). Apesar de um modelo com tempos constantes facilitar a exploração do espaço de projeto, ele é pouco realista e pode levar a arquiteturas ineficientes. Variações nos tempos de execução podem ser causadas por diversos fatores, tais como os dados de entrada e aspectos da arquitetura (ex.: *pipeline*).

- Ao invés de adotar tempos constantes, uma parte dos métodos existentes usa distribuições de probabilidade para representar a variabilidade no tempo de execução dos sistemas embarcados. No entanto, a maioria deles limita os tipos de sistemas que podem ser representados. Por exemplo, grande parte deles considera sistemas com apenas um processador ou distribuições de probabilidade específicas (ex.: distribuição exponencial) para os tempos de execução.

- Uma outra classe de métodos adota modelos executáveis para representar a aplicação e/ou arquitetura. Modelos executáveis são uma alternativa natural para capturar a variabilidade no tempo de execução dos sistemas embarcados. A grande desvantagem dos modelos executáveis é que o tempo de avaliação (simulação) é em geral fortemente dependente da complexidade da especificação funcional da aplicação/arquitetura, o que muitas vezes limita a quantidade de soluções que podem ser exploradas em um espaço razoável de tempo.

- Outra grande limitação frequentemente encontrada nos métodos da literatura é que as estimativas geradas pelos modelos propostos não são comparadas a medições em uma arquitetura real (ou um simulador de baixo nível). Dessa maneira, fica difícil analisar a exatidão desses modelos.

Os itens a seguir destacam as principais contribuições do trabalho proposto e como as restrições dos trabalhos atuais são atacadas por ele.

- **Modelos de especificação:** O trabalho desenvolveu novos modelos de especificação para capturar os aspectos comportamentais e as restrições de projeto. Diferentemente dos métodos que assumem tempos constantes, os modelos de especificação concebidos adotam distribuições de probabilidade para capturar a variabilidade dos tempos de execução dos sistemas embarcados. Para demonstrar a viabilidade destes modelos, este trabalho mostrou como uma aplicação (*SHOUTcast player*) e arquitetura reais foram especificadas.

- **Mapeamento e modelos de avaliação:** Este trabalho concebeu um procedimento que automaticamente mapeia as especificações da aplicação e plataforma de *hardware* em modelos de simulação formal DEVS. Os modelos são livres de detalhes funcionais, permitindo que o método proposto evite os longos tempos de avaliação dos métodos que adotam modelos executáveis. Além disso, os modelos desenvolvidos não possuem as limitações dos modelos analíticos existentes (ex.: tempos de

execução com distribuição de probabilidade exponencial). O trabalho também propôs modelos para estimar a potência consumida de uma arquitetura. Diferentemente de muitos trabalhos na literatura, as estimativas produzidas pelos modelos propostos foram comparadas a medições em um sistema real. Os experimentos conduzidos mostram a boa exatidão dos modelos de avaliação desenvolvidos (erro máximo menor que 5%).

- **Algoritmos de exploração:** Dois algoritmos genéticos multiobjetivo foram desenvolvidos para exploração do espaço de soluções: MODSES e C-MODSES. A partir de *benchmarks* disponíveis na literatura, MODSES foi comparado a três outros algoritmos: (i) EMOGAC, que é um algoritmo representativo da classe de algoritmos que assumem tempos constantes, (ii) SPGA, que é um algoritmo representativo do estado da arte dos algoritmos genéticos multiobjetivo, e (iii) um algoritmo que implementa uma busca aleatória. Os resultados mostram que, para sistemas embarcados de tempo-real não críticos, MODSES é superior a EMOGAC. MODSES foi capaz de achar soluções com custo monetário até 50% inferior que as encontradas por EMOGAC. MODSES também foi superior a SPGA e a busca aleatória. O algoritmo C-MODSES é uma versão modificada de MODSES, em que violações de *deadlines* são tratadas como restrições a serem cumpridas, e não como um objetivo a ser otimizado. O algoritmo C-MODSES acelera o tempo de busca evitando desperdiçar recursos computacionais com soluções que possuem grandes chances de violar restrições. Os resultados mostram que C-MODSES é até 10 vezes mais rápido que MODSES.
- **Métodos de caracterização:** Em uma das atividades do fluxo de atividades do método proposto, algumas informações sobre o tempo de execução das tarefas e potência consumida da plataforma de *hardware* precisam ser obtidas. Este trabalho propôs métodos baseados em medição para obter essas informações. Em particular, mostramos como a potência consumida de uma plataforma de *hardware* multiprocessada (LPC 4357) foi caracterizada usando a ferramenta AMALGHMA, da qual o autor desta tese é co-criador. No melhor do nosso conhecimento, nenhum outro trabalho havia caracterizado esta plataforma de *hardware*.
- **Biblioteca de funções:** Uma biblioteca foi desenvolvida para dar suporte ao desenvolvimento de aplicações que executam em arquiteturas heterogêneas. A biblioteca fornece um conjunto de funções para gerenciar a comunicação entre tarefas mapeadas no mesmo processador ou entre tarefas mapeadas em processadores diferentes, permitindo que o projetista possa abstrair detalhes de implementação de baixo nível, como semáforos e gerenciamento de interrupções. Os resultados mostram que a biblioteca desenvolvida é adequada para especificar aplicações que seguem um modelo de computação baseado em HSDGs.

8.1 Trabalhos futuros

A seguir, são listadas algumas possibilidades de trabalhos futuros:

- Trabalhos futuros poderiam considerar outras medidas de projeto (ex.: espaço em memória, confiabilidade, tamanho do chip), além das que são consideradas atualmente. Como os algoritmos de exploração propostos são multiobjetivo, nenhuma modificação precisaria ser feita neles, bastando definir e incluir os modelos de avaliação apropriados para as novas medidas.

- O método proposto atualmente só permite a especificação de *deadlines* não críticos. Uma extensão natural seria permitir também a especificação de *deadlines* críticos. Nesse caso, modelos analíticos devem ser desenvolvidos e usados em conjunto aos modelos de simulação atuais.

- Uma maneira direta de facilitar a execução do fluxo de atividades do método proposto seria automatizar mais passos desse fluxo. Nesse sentido, os passos de caracterização dos tempos de execução das tarefas e a definição (extração) do HSDG da aplicação são bons candidatos a serem automatizados.

- Nos últimos anos, as NoC (*Network-on-Chip*) tem despontado como uma resposta viável aos problemas que ocorrem nas arquiteturas de comunicação tradicionais quando o número de componentes no chip cresce (MARCULESCU et al., 2009). Dessa forma, o trabalho poderia ser estendido para considerar arquiteturas mais complexas de comunicação, como as NoC.

- O modelo adotado para especificar a aplicação é mais indicado para representar aplicações que possuem comportamento focado em dados. Extensões ao modelo de aplicação atual podem ser consideradas para que aplicações com comportamento focado em controle também possam ser modeladas.

- Trabalhos futuros poderiam estudar a integração do método proposto com métodos de exploração que atuam em níveis mais baixos de abstração, como os métodos que tentam achar a melhor arquitetura de *cache* para uma dada aplicação.

- Por último, o ambiente proposto por esta tese considera que a aplicação será implementada apenas em *software*, que executa em processadores programáveis. Futuramente, iremos estender o ambiente para permitir que partes da aplicação também possam ser implementadas em *hardware* (ex.: FPGA). Nesse caso, mecanismos para geração de código HDL (*Hardware Description Language*) a partir de uma especificação funcional, assim como um procedimento para geração da interface *hardware/software* devem ser integrados ao ambiente proposto.

Referências

ABENI, L.; MANICA, N.; PALOPOLI, L. Efficient and robust probabilistic guarantees for real-time tasks. **Journal of Systems and Software**, [S.l.], v.85, n.5, p.1147–1156, 2012.

APRIL, J. et al. Practical introduction to simulation optimization. In: WINTER SIMULATION CONFERENCE, 2003. **Proceedings...** [S.l.: s.n.], 2003. v.1, p.71–78.

AUSTIN, T.; LARSON, E.; ERNST, D. SimpleScalar: an infrastructure for computer system modeling. **Computer**, [S.l.], v.35, n.2, p.59–67, 2002.

BANKS, J. **Handbook of simulation**. [S.l.]: Wiley, 1998.

BARRY, R. **Using the FreeRTOS real time kernel**: a practical guide. [S.l.]: Real Time Engineers, 2010.

BENINI, L. et al. Mparm: exploring the multi-processor soc design space with systemc. **Journal of VLSI signal processing systems for signal, image and video technology**, [S.l.], v.41, n.2, p.169–182, 2005.

BLAKE, G.; DRESLINSKI, R.; MUDGE, T. A survey of multicore processors. **IEEE Signal Processing Magazine**, [S.l.], v.26, n.6, p.26–37, 2009.

BLICKLE, T. **Theory of evolutionary algorithms and application to system synthesis**. [S.l.]: Hochschulverlag, 1997. v.17.

BOLCH, G. et al. **Queueing networks and Markov chains**: modeling and performance evaluation with computer science applications. [S.l.]: John Wiley & Sons, 2006.

BROOKS, D.; TIWARI, V.; MARTONOSI, M. Wattch: a framework for architectural-level power analysis and optimizations. **ACM SIGARCH Computer Architecture News**, [S.l.], v.28, n.2, p.83–94, 2000.

CHOW, A.; ZEIGLER, B. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In: CONFERENCE ON WINTER SIMULATION, 26. **Proceedings...** [S.l.: s.n.], 1994. p.716–722.

CHU, P. **RTL hardware design using VHDL**: coding for efficiency, portability, and scalability. [S.l.]: Wiley, 2006.

COELLO, C. A. C.; LAMONT, G. B. **Evolutionary algorithms for solving multi-objective problems**. [S.l.]: Springer, 2007. v.5.

DEB, K. **Multi-objective optimization using evolutionary algorithms**. [S.l.]: John Wiley & Sons, 2008.

DEB, K. et al. A fast and elitist multiobjective genetic algorithm: nsga-ii. **IEEE Transactions on Evolutionary Computation**, [S.l.], v.6, n.2, p.182–197, 2002.

DENSMORE, D.; PASSERONE, R.; SANGIOVANNI-VINCENTELLI, A. A platform-based taxonomy for ESL design. **IEEE Design and Test of Computers**, [S.l.], v.23, n.5, p.359–374, 2006.

DICK, R. **Embedded system synthesis benchmarks suites (E3S)**. 2014.

DICK, R. P. **Multiobjective synthesis of low-power real-time distributed embedded systems**. 2002. Tese (Doutorado em Ciência da Computação) — Princeton University.

DING, H.; BENYOUCEF, L.; XIE, X. A simulation-based multi-objective genetic algorithm approach for networked enterprises optimization. **Engineering Applications of Artificial Intelligence**, [S.l.], v.19, n.6, p.609–623, 2006.

DÖMER, R. et al. System-on-chip environment: a specc-based framework for heterogeneous mpsoc design. **EURASIP Journal on Embedded Systems**, [S.l.], v.2008, 2008.

DORIGO, M.; GAMBARDELLA, L. Ant colony system: a cooperative learning approach to the traveling salesman problem. **IEEE Transactions on Evolutionary Computation**, [S.l.], v.1, n.1, p.53–66, 1997.

EIBEN, A. E.; SMITH, J. E. **Introduction to evolutionary computing**. [S.l.]: Springer, 2003.

ERBAS, C.; CERAV-ERBAS, S.; PIMENTEL, A. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. **IEEE Transactions on Evolutionary Computation**, [S.l.], v.10, n.3, p.358–374, 2006.

ESKANDARI, H.; GEIGER, C. Evolutionary multiobjective optimization in noisy problem environments. **Journal of Heuristics**, [S.l.], v.15, n.6, p.559–595, 2009.

ESKANDARI, H.; GEIGER, C. D.; BIRD, R. Handling uncertainty in evolutionary multiobjective optimization: spga. In: IEEE CONGRESS ON EVOLUTIONARY COMPUTATION. **Anais...** [S.l.: s.n.], 2007. p.4130–4137.

EWING, G.; PAWLIKOWSKI, K.; MCNICKLE, D. Akaroa-2: exploiting network computing by distributing stochastic simulation. In: EUROPEAN SIMULATION MULTI-CONFERENCE, 13. **Proceedings...** SCSI Press, 1999.

FEO, T. A.; RESENDE, M. G. C.; SMITH, S. H. A greedy randomized adaptive search procedure for maximum independent set. **Operations Research**, [S.l.], v.42, n.5, p.860–878, 1994.

FERRARI, A.; SANGIOVANNI-VINCENTELLI, A. System design: traditional concepts and new paradigms. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN (ICCD). **Anais...** [S.l.: s.n.], 1999.

FU, M. C. Optimization for simulation: theory vs. practice. **INFORMS Journal on Computing**, [S.l.], v.14, n.3, p.192–215, 2002.

FU, M. C.; GLOVER, F. W.; APRIL, J. Simulation optimization: a review, new developments, and applications. In: WINTER SIMULATION CONFERENCE. **Proceedings...** IEEE, 2005.

GAJSKI, D. D. et al. **Embedded System Design**: modeling, synthesis and verification. [S.l.]: Springer, 2009.

GARDNER, M.; LIU, J. Analyzing stochastic fixed-priority real-time systems. In: INTERNATIONAL CONFERENCE ON TOOLS AND ALGORITHMS FOR CONSTRUCTION AND ANALYSIS OF SYSTEMS, 5. **Proceedings...** Springer, 1999. p.44–58.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability**: a guide to the theory of np-completeness. [S.l.]: WH Freeman and Company, 1979.

GAUTAMA, H.; GEMUND, A. van. Static performance prediction of data-dependent programs. In: INTERNATIONAL WORKSHOP ON SOFTWARE AND PERFORMANCE, 2. **Proceedings...** ACM, 2000. v.2000, p.216–226.

GEILEN, M.; BASTEN, T.; STUIJK, S. Minimising buffer requirements of synchronous dataflow graphs with model checking. In: DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.: s.n.], 2005. p.819–824.

GEN, M.; CHENG, R. **Genetic algorithms and engineering optimization**. [S.l.]: John Wiley & Sons, 2000. v.7.

GERSTLAUER, A. et al. Electronic system-level synthesis methodologies. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.28, n.10, p.1517–1530, 2009.

GHAMARIAN, A. et al. Liveness and boundedness of synchronous data flow graphs. In: FORMAL METHODS IN COMPUTER AIDED DESIGN. **Anais...** [S.l.: s.n.], 2006. p.68–75.

GILLES, K. The semantics of a simple language for parallel programming. In: IN INFORMATION PROCESSING: PROCEEDINGS OF THE IFIP CONGRESS. **Anais...** [S.l.: s.n.], 1974. v.74, p.471–475.

GONZÁLEZ, M. et al. Mast: modeling and analysis suite for real time applications. In: EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS, 13. **Anais...** [S.l.: s.n.], 2001. p.125–134.

GRIES, M. Methods for evaluating and covering the design space during early design development. **Integration, the VLSI Journal**, [S.l.], v.38, n.2, p.131–183, 2004.

GUTJAHR, W. J. Recent trends in metaheuristics for stochastic combinatorial optimization. **Central European Journal of Computer Science**, [S.l.], v.1, n.1, p.58–66, 2011.

HA, S. et al. PeaCE: a hardware-software codesign environment for multimedia embedded systems. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, [S.l.], v.12, n.3, p.24, 2007.

HENIA, R. et al. System level performance analysis–the SymTA/S approach. **Computers and Digital Techniques**, [S.l.], v.152, n.2, p.148–166, 2005.

HERRERA, F.; SANDER, I. Combining Analytical and Simulation-Based Design Space Exploration for Efficient Time-Critical and Mixed-Criticality Systems. In: **Languages, Design Methods, and Tools for Electronic System Design**. [S.l.]: Springer, 2015. p.167–188.

HILLIER, F. S. **Introduction to Operations Research**. [S.l.]: McGraw-Hill, 1990.

HOLDINGS, A. **AMBA 3 AHB-Lite Protocol**. 2006.

HOLLAND, J. **Adaptation in natural and artificial systems**: an introductory analysis with applications to biology, control, and artificial intelligence. [S.l.]: University of Michigan Press, 1975.

HOU, J.; WOLF, W. Process partitioning for distributed embedded systems. In: INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CO-DESIGN, 4. **Proceedings...** IEEE, 1996. p.70.

HUA, S.; QU, G.; BHATTACHARYYA, S. Probabilistic design of multimedia embedded systems. **ACM Transactions on Embedded Computing Systems (TECS)**, [S.l.], v.6, n.3, p.15, 2007.

INC, R. **The Helix MP3 Decoder**. 2014.

JIA, Z. et al. A two-phase design space exploration strategy for system-level real-time application mapping onto MPSoC. **Microprocessors and Microsystems**, [S.l.], v.38, n.1, p.9–21, 2014.

JOINES, J. et al. Supply chain multi-objective simulation optimization. In: WINTER SIMULATION CONFERENCE. **Proceedings...** [S.l.: s.n.], 2002. v.2, p.1306–1314.

KANGAS, T. et al. UML-based multiprocessor SoC design framework. **ACM Transactions on Embedded Computing Systems (TECS)**, [S.l.], v.5, n.2, p.281–320, 2006.

KEINERT, J. et al. SystemCoDesigner - an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, [S.l.], v.14, n.1, p.1, 2009.

KEMPF, T.; ASCHEID, G.; LEUPERS, R. **Multiprocessor Systems on Chip**. [S.l.]: Springer, 2011.

KIM, K.; LEE, C. A safe stochastic analysis with relaxed limitations on the periodic task model. **IEEE Transactions on Computers**, [S.l.], v.58, n.5, p.634–647, 2009.

KOPETZ, H. **Real-time systems**: design principles for distributed embedded applications. [S.l.]: Springer, 2011.

LAVAGNO, L.; PASSERONE, C. **Embedded Systems Handbook**. [S.l.]: Taylor & Francis, 2006.

LEE, E. A.; MESSERSCHMITT, D. G. Synchronous data flow. In: IEEE. **Proceedings...** IEEE, 1987. v.75, p.1235–1245.

LEISERSON, C. E. et al. **Introduction to algorithms**. [S.l.]: The MIT press, 2001.

LEUNG, J. A new algorithm for scheduling periodic, real-time tasks. **Algorithmica**, [S.l.], v.4, n.1-4, p.209–219, 1989.

LINDEMANN, C. Performance modelling with deterministic and stochastic Petri nets. **ACM SIGMETRICS Performance Evaluation Review**, [S.l.], v.26, n.2, p.3, 1998.

LIU, J. W. S. **Real-Time Systems**. [S.l.]: Prentice Hall, 2000.

MANOLACHE, S.; ELES, P.; PENG, Z. Schedulability analysis of multiprocessor real-time applications with stochastic task execution times. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** ACM, 2002. p.699–706.

MANOLACHE, S.; ELES, P.; PENG, Z. Schedulability analysis of applications with stochastic task execution times. **ACM Transactions on Embedded Computing Systems (TECS)**, [S.l.], v.3, n.4, p.706–735, 2004.

MANOLACHE, S.; ELES, P.; PENG, Z. Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints. **ACM Transactions on Embedded Computing Systems (TECS)**, [S.l.], v.7, n.2, 2008.

MANSLEY, K. **lwIP A Lightweight TCP/IP stack**. 2014.

MANTEGAZZA, P.; DOZIO, E.; PAPACHARALAMBOUS, S. RTAI: real time application interface. **Linux Journal**, [S.l.], v.2000, n.72es, p.10, 2000.

MARCULESCU, R. et al. Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.28, n.1, p.3–21, 2009.

MARCULESCU, R.; PEDRAM, M.; HENKEL, J. Distributed multimedia system design: a holistic perspective. In: DESIGN, AUTOMATION AND TEST IN EUROPE. **Proceedings...** [S.l.: s.n.], 2004. p.21342.

MARWEDEL, P. **Embedded systems design - embedded systems foundations of cyber-physical systems**. [S.l.]: Springer, 2011.

MATHWORKS. **Matlab**. 2014.

MOLLOY, K. Performance analysis using stochastic Petri nets. **IEEE Transactions on Computers**, [S.l.], v.100, n.9, p.913–917, 1982.

MOYER, B. **Real World Multicore Embedded Systems**. [S.l.]: Newnes, 2013.

MUPPALA, J.; WOOLET, S.; TRIVEDI, K. Real-time systems performance in the presence of failures. **Computer**, [S.l.], v.24, n.5, p.37–47, 1991.

MURATA, T. Petri nets: properties, analysis and applications. **Proceedings of the IEEE**, [S.l.], v.77, n.4, p.541–580, 1989.

NIELSEN, J. **Usability engineering**. [S.l.]: Elsevier, 1994.

NIKOLOV, H. et al. Daedalus: toward composable multimedia mp-soc design. In: DESIGN AUTOMATION CONFERENCE, 45. **Proceedings...** [S.l.: s.n.], 2008. p.574–579.

NOGUEIRA, B. et al. A Formal Model for Performance and Energy Evaluation of Embedded Systems. **EURASIP Journal on Embedded Systems**, [S.l.], v.2011, 2010.

NOGUEIRA, B. et al. ALUPAS: avaliação de desempenho e consumo de energia de softwares para sistemas embarcados. **Revista de Informática Teórica e Aplicada**, [S.l.], v.16, n.1, p.25–44, 2010.

NOGUEIRA, B. et al. A simulation optimization approach for design space exploration of soft real-time embedded systems. In: IEEE CONGRESS ON EVOLUTIONARY COMPUTATION (CEC). **Anais...** [S.l.: s.n.], 2013. p.2773–2780.

NUTARO, J. ADEVS 2.6. **http://www.ornl.gov/~1qn/adevs/**, [S.l.], 2012.

OPTTEK SYSTEMS, I. **OptQuest**. 2013.

PATTERSON, D.; HENNESSY, J. **Computer organization and design**: the hardware/software interface. [S.l.]: Newnes, 2013.

PAWLIKOWSKI, K. Steady-state simulation of queueing processes: survey of problems and solutions. **ACM Computing Surveys (CSUR)**, [S.l.], v.22, n.2, p.123–170, 1990.

PAWLIKOWSKI, K.; JEONG, H.; LEE, J. On credibility of simulation studies of telecommunication networks. **IEEE Communications Magazine**, [S.l.], v.40, n.1, p.132–139, 2002.

PEES, S. et al. LISA: machine description language for cycle-accurate models of programmable dsp architectures. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 36. **Proceedings...** [S.l.: s.n.], 1999. p.933–938.

PIMENTEL, A.; ERBAS, C.; POLSTRA, S. A systematic approach to exploring embedded system architectures at multiple abstraction levels. **IEEE Transactions on Computers**, [S.l.], v.55, n.2, p.99–112, 2006.

PIMENTEL, A. et al. Exploring embedded-systems architectures with Artemis. **Computer**, [S.l.], v.34, n.11, p.57–63, 2001.

PURNAPRAJNA, M.; REFORMAT, M.; PEDRYCZ, W. Genetic algorithms for hardware–software partitioning and optimal resource allocation. **Journal of Systems Architecture**, [S.l.], v.53, n.7, p.339–354, 2007.

QIU, M. et al. Voltage assignment with guaranteed probability satisfying timing constraint for real-time multiproceesor DSP. **The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology**, [S.l.], v.46, n.1, p.55–73, 2007.

QUAN, W.; PIMENTEL, A. A Hybrid Task Mapping Algorithm for Heterogeneous MPSoCs. **ACM Transactions on Embedded Computing Systems (TECS)**, [S.l.], v.14, n.1, 2014.

RAO, S. S.; RAO, S. S. **Engineering optimization**: theory and practice. [S.l.]: John Wiley & Sons, 2009.

REFAAT, K.; HLADIK, P. Efficient stochastic analysis of real-time systems via random sampling. In: EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS (ECRTS). **Anais...** [S.l.: s.n.], 2010. p.175–183.

RETHINAGIRI, S. K. et al. An efficient power estimation methodology for complex risc processor-based platforms. In: VLSI. **Proceedings...** [S.l.: s.n.], 2012. p.239–244.

ROBINSON, S. **Simulation**: the practice of model development and use. [S.l.]: John Wiley & Sons, 2004.

RUGGIERO, M. et al. Scalability analysis of evolving SoC interconnect protocols. In: INTERNATIONAL SYMPOSIUM ONS YSTEM ON CHIP (SOC). **Anais...** [S.l.: s.n.], 2004. p.169–172.

RUGGIERO, M. et al. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE: PROCEEDINGS. **Proceedings...** [S.l.: s.n.], 2006. p.3–8.

RUSU, C. et al. Energy-efficient policies for request-driven soft real-time systems. In: EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS. **Anais...** [S.l.: s.n.], 2004. p.175–183.

SANGIOVANNI-VINCENTELLI, A. et al. Benefits and challenges for platform-based design. In: DESIGN AUTOMATION CONFERENCE, 41. **Proceedings...** [S.l.: s.n.], 2004. p.409–414.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. **IEEE Design & Test of Computers**, [S.l.], v.18, n.6, p.23–33, 2001.

SARASWAT, P. K.; POP, P.; MADSEN, J. Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. In: REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM (RTAS). **Anais...** [S.l.: s.n.], 2010. p.89–98.

SATISH, N. R.; RAVINDRAN, K.; KEUTZER, K. Scheduling task dependence graphs with variable task execution times onto heterogeneous multiprocessors. In: ACM INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE, 8. **Proceedings...** [S.l.: s.n.], 2008. p.149–158.

SCHMITZ, M. T.; AL-HASHIMI, B.; ELES, P. **System-level design techniques for energy-efficient embedded systems**. [S.l.]: Springer, 2004.

SEMICONDUCTORS, N. **SWIM graphics library**. 2011.

SEMICONDUCTORS, N. **LPC43xx ARM Cortex-M4/M0 multi-core microcontroller - User manual**. 2014.

SILVA, B. et al. AMALGHMA An Environment for Measuring Execution Time and Energy Consumption in Embedded Systems. In: IEEE INTERNATIONAL CONFERENCE ON SYSTEMS, MAN AND CYBERNETICS. **Proceedings...** IEEE, 2014.

SINHA, A.; CHANDRAKASAN, A. P. JouleTrack: a web based tool for software energy profiling. In: DESIGN AUTOMATION CONFERENCE, 38. **Proceedings...** [S.l.: s.n.], 2001. p.220–225.

SIVANANDAM, S. N.; DEEPA, S. N. **Introduction to genetic algorithms**. [S.l.]: Springer, 2007.

SMITH, J. Practical: a shoutcast server. **Practical OCaml**, [S.l.], p.293–308, 2007.

SONNTAG, S.; GRIES, M.; SAUER, C. SystemQ: bridging the gap between queuing-based performance evaluation and systemc. **Design Automation for Embedded Systems**, [S.l.], v.11, n.2-3, p.91–117, 2007.

TALBI, E. **Metaheuristics**: from design to implementation. [S.l.]: John Wiley & Sons, 2009. v.74.

TANENBAUM, A.; BOS, H. **Modern operating systems**. 4.ed. [S.l.]: Pearson, 2014.

TAVARES, E. et al. Model-driven software synthesis for hard real-time applications with energy constraints. **Design Automation for Embedded Systems**, [S.l.], v.14, n.4, p.327–366, 2010.

TEICH, J. Hardware/software codesign: the past, the present, and predicting the future. **Proceedings of the IEEE**, [S.l.], v.100, n.Special Centennial Issue, p.1411–1430, 2012.

THIELE, L. et al. A framework for evaluating design tradeoffs in packet processing architectures. In: DESIGN AUTOMATION CONFERENCE, 39. **Proceedings...** IEEE, 2002. p.880–885.

THIELE, L.; PERATHONER, S. **Model-Based Design for Embedded Systems**. [S.l.]: CRC Press, 2010.

VAHID, F.; GIVARGIS, T. **Embedded System Design**: a unified hardware/software introduction. [S.l.]: John Wiley & Sons, 2001.

WEISBERG, S. **Applied linear regression**. [S.l.]: John Wiley & Sons, 2014.

WILLIAMSON, D.; SHMOYS, D. **The design of approximation algorithms**. [S.l.]: Cambridge University Press, 2011.

WOLF, M. **Computers as components**: principles of embedded computing system design. [S.l.]: Elsevier, 2012.

WOLF, M. **High-Performance Embedded Computing**. [S.l.]: Morgan Kaufmann Publishers, 2014.

WOLF, W.; JERRAYA, A.; MARTIN, G. Multiprocessor system-on-chip (MPSoC) technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.27, n.10, p.1701–1713, 2008.

YALDIZ, S.; DEMIR, A.; TASIRAN, S. Stochastic modeling and optimization for energy management in multicore systems: a video decoding case study. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.27, n.7, p.1264–1277, 2008.

YIU, J. **The Definitive Guide to the ARM Cortex-M0**. [S.l.]: Newnes, 2011.

YIU, J. **The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors**. [S.l.]: Newnes, 2013.

ZAMORA, N. H.; HU, X.; MARCULESCU, R. System-level performance/power analysis for platform-based design of multimedia applications. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, [S.l.], v.12, n.1, 2007.

ZEIGLER, B. P.; PRAEHOFER, H.; KIM, T. G. **Theory of modeling and simulation**: integrating discrete event and continuous complex dynamic systems. [S.l.]: Academic Press, 2000.

ZITZLER, E. **Evolutionary algorithms for multiobjective optimization**: methods and applications. [S.l.]: Shaker, 1999. v.63.

ZITZLER, E.; LAUMANNS, M.; THIELE, L. **SPEA2**: improving the strength pareto evolutionary algorithm. [S.l.]: Eidgenössische Technische Hochschule Zürich (ETH), Institut für Technische Informatik und Kommunikationsnetze (TIK), 2001.

Apêndice

A
Bloco recurso sem preempção

A Figura A.1 mostra o modelo atômico P-DEVS que representa um processador com sistema operacional em que preempções não estão autorizadas. O modelo é muito similar ao modelo de recurso que suporta preempção apresentado na Figura 5.6. Ele possui uma porta de entrada e uma de saída para cada uma dos k blocos tarefa associados ao bloco recurso (linhas 3 e 4): $jobin_1$, ..., $jobin_k$ são as portas de entrada, e $jobout_1$, ..., $jobout_k$ são as portas de saída. As portas de entrada e saída são usadas, respectivamente, para receber novas cargas de trabalho e notificar as tarefas sobre a finalização de sua respectiva carga de trabalho. O estado do modelo é representado por uma tupla que determina (linha 5): (i) qual carga de trabalho está sendo executada no momento ($currentjob$), (ii) se a carga de trabalho já iniciou sua execução ($has_started$), e (iii) o estado da fila de prioridades com as cargas de trabalho prontas para executar ($priorityqueue$).

O modelo faz uso das seguintes funções auxiliares: $Insert(l,x)$, que insere na fila de prioridades l a carga de trabalho x; e $Pop(l)$, que retorna e remove da fila de prioridades l a carga de trabalho de maior prioridade. As seguintes constantes são usadas no modelo: $context\_switch\_delay$, que representa o *overhead* de uma mudança de contexto pelo sistema operacional; e $max\_priority$, que representa a prioridade máxima do sistema.

Sempre que um bloco recurso recebe um evento de entrada (linha 7), isso significa que novas cargas de trabalho chegaram. Quando isso acontece, as novas cargas de trabalho são inseridas na fila de prioridades (linhas 9-11). O *overhead* de uma mudança de contexto é adicionado assim que a carga de trabalho é recebida. Caso o bloco recurso esteja ocioso no momento da chegada (linha 12), então a carga de trabalho de maior prioridade é imediatamente preparada para executar (linhas 19-21). Caso contrário, é preciso verificar se a carga de trabalho atual já começou a executar (linha 13), se a carga já iniciou, então ela não pode sofrer preempção, mesmo que uma carga de trabalho de maior prioridade tenha chegado. Assim, o tempo restante para executar a carga atual é atualizado (linhas 14 e 15). Caso a carga atual ainda não tenha iniciado, então ela pode sofrer preempção se uma carga de trabalho de maior prioridade foi recebida (linhas 17 e 18).

$Resource_{np} = (X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$
$Job = \{exec\_time \in \mathbb{R}^+, priority \in \mathbb{N}, id \in \mathbb{N}\}$
$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ tal que $IPorts \in \{\text{“}jobin_1\text{”}, \ldots, \text{“}jobin_k\text{”}\}, X_{jobin_i} \in Job$
$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ tal que $OPorts \in \{\text{“1”}, \ldots, \text{“}k\text{”}\}, Y_i \in \{\text{“}done\text{”}\}$
$S = \{current\_job \in Job \cup \{\text{“}nil\text{”}\}, has\_started \in \{true, false\}, priorityqueue \in \{job \in Job\}*\}$

$\mathbf{s' = \delta_{ext}(s, e, x^b)}$:
    $s' := s$
    foreach x **in** x^b **do**
        $x.v.exec\_time := x.v.exec\_time + context\_switch\_delay$ // adiciona o *overhead* de mudança de contexto
        $Insert(s'.priorityqueue, x.v)$ // insere a nova carga de trabalho
    if $s'.current\ job \neq \text{“}nil\text{”}$ **then** // se o recurso não está ocioso
        if $e > 0$ **then**
            $s'.current\ job.exec\_time := s'.current\ job.exec\_time - e$ // atualiza o tempo restante para terminar a carga de trabalho atual
            $s'.current\ job.has\_started := true$ // indica que a carga de trabalho atual já começou e por isso ela não pode sofrer preempção
        else if $e = 0 \wedge has\_started = false$ **then** // caso a carga de trabalho atual ainda não tenha começado
            $Insert(s'.priorityqueue, s'.current\ job)$
            $s'.current\ job := Pop(s'.priorityqueue)$ // a carga de trabalho atual pode sofrer preempção caso não tenha começado e outra carga de trabalho de maior prioridade tenha chegado
    else
        $s'.current\ job := Pop(s'.priorityqueue)$ // seleciona e remove a carga de trabalho com maior prioridade na fila
        $s'.has\_started := false$

$\sigma = \mathbf{ta(s)}$:
    if $s.current\ job \neq \text{“}nil\text{”}$ **then**
        $\sigma := s.current\ job.exec\_time$
    else
        $\sigma := +\infty$ // recurso ocioso

$\mathbf{y^b = \lambda(s)}$:
    if $current\ job.id \neq max\_priority$ **then** // verifica se a carga de trabalho finalizada representa a execução da tarefa de tratamento de mensagens interprocessador
        $y.p := current\ job.id,\ y.v := \text{“}done\text{”},\ y^b := \{y\}$ // avisa o bloco tarefa que a carga de trabalho enviada terminou

$\mathbf{s' = \delta_{int}(s)}$:
    $s'.current\ job := \text{“}nil\text{”}, s'.priorityqueue := s.priorityqueue$
    if $s'.priorityqueue \neq ()$ **then** // se a fila não estiver vazia
        $s'.current\ job := Pop(s'.priorityqueue)$ // inicia a execução da próxima carga de trabalho
        $s'.has\_started := false$

$\mathbf{s' = \delta_{con}(s, x^b)}$:
    $s' := \delta_{ext}(\delta_{int}(s), 0, x^b)$ // a carga de trabalho atual precisa ser finalizada primeiro

Figura A.1: Modelo atômico P-DEVS que representa um processador com sistema operacional em que preempções não estão autorizadas (bloco recurso sem preempção).