
# EXPLORANDO A EQUIVALÊNCIA ENTRE UMA MÁQUINA QUÍMICA ABSTRATA E COMPUTAÇÃO DATAFLOW

Rui Rodrigues de Mello Junior

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Gabriel Antoine Louis Paillard
Leandro Santiago de Araújo

Rio de Janeiro
Março de 2022

EXPLORANDO A EQUIVALÊNCIA ENTRE UMA MÁQUINA QUÍMICA ABSTRATA E COMPUTAÇÃO DATAFLOW

Rui Rodrigues de Mello Junior

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientadores: Felipe Maia Galvão França
Gabriel Antoine Louis Paillard
Leandro Santiago de Araújo

Aprovada por: Prof. Felipe Maia Galvão França
Prof. Gabriel Antoine Louis Paillard
Prof. Leandro Santiago de Araújo
Prof. Claudio Luis de Amorim
Prof. Diego Leonel Cadette Dutra
Prof. Tiago Assumpção de Oliveira Alves
Prof. Edson Borin

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2022

Mello Junior, Rui Rodrigues de

Explorando a equivalência entre uma máquina química abstrata e computação dataflow/Rui Rodrigues de Mello Junior. – Rio de Janeiro: UFRJ/COPPE, 2022.

XVIII, 241 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França
Gabriel Antoine Louis Paillard
Leandro Santiago de Araújo

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2022.

Referências Bibliográficas: p. 221 – 230.

1. Gamma. 2. Dataflow. 3. Computação Paralela. 4. Equivalência Computacional. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

*"Em essência, somos um*
*conjunto de reações químicas"*
*(Marcelo Gleiser)*

*"Eu levo em mim o que eu não esqueço"*
*(Allan Dias Castro)*

*Dedico este trabalho ao meu filho Bernardo por me tornar um homem melhor e por me ensinar mais do que aprender comigo. Sem você nada disso teria sido possível.*

# Agradecimentos

Agradeço primeiramente a Deus pela minha vida, dos meus familiares e amigos, e por sua infinita misericórdia, permitir que chegasse até aqui.

Ao meu filho Bernardo, que mesmo sem saber como, me motiva sempre a ser melhor e alcançar coisas inimagináveis. O papai te ama filho! Aos meus pais, Sueli e Ruy, por toda educação e amor irrestritos. Sem o apoio e dedicação de vocês eu nada seria. Obrigado por me fazerem ser o homem que sou. À minha esposa por todo o apoio e ajuda com o Bernardo, enquanto eu estive ausente. Ao meu irmão Rafael por todo o companheirismo, apoio, dedicação e amizade, hoje e sempre. OSS! À todos os meus familiares que acreditaram no meu potencial e me incentivaram a buscar sempre mais.

Ao Shihan Lirton dos Reis Monassa (*in memorian*), por ter contribuído para a formação do meu caráter e por ter me ensinado a criar o intuito do esforço, dentre tantas lições. OSS!

À Rubens Pailo, grande amigo, companheiro desde o mestrado, por todas as dúvidas esclarecidas, sugestões e apoio. Muito obrigado.

Ao Instituto de Pesquisas da Marinha, especialmente aos amigos CMG(RM1) Da Mota e CMG (RM1) Andrada pelo grande incentivo e apoio.

Ao Programa de Engenharia de Sistemas e Computação (PESC/COPPE/UFRJ) por todo o suporte fornecido que me permitiu desenvolver este trabalho. Em especial gostaria de agradecer aos professores Jayme Szwarcfiter, Claudio Amorim, Valmir Barbosa, Leandro Marzulo pelos conhecimentos transmitidos nas disciplinas cursadas. Agradeço também aos funcionários do PESC pelo suporte e apoio prestados. Aos amigos do PESC: Evandro, Vitor Cruz, Bruno e Caio pelo companheirismo e motivação do dia a dia.

Ao professor Felipe Maia Galvão França, por despertar em mim a paixão pela ciência. Por todo o apoio irrestrito e por confiar em mim, muitas vezes mais do que eu mesmo. Aos professores Gabriel Louis Antoine Paillard e Leandro Santiago de Araújo, por todo apoio, compreensão, atenção e por contribuírem em muito para a realização deste trabalho.

Ao prestimoso apoio do CENAPAD-UFC, por apoiar a realização deste trabalho, permitindo a execução dos experimentos em seu supercomputador.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

# EXPLORANDO A EQUIVALÊNCIA ENTRE UMA MÁQUINA QUÍMICA ABSTRATA E COMPUTAÇÃO DATAFLOW

Rui Rodrigues de Mello Junior

Março/2022

Orientadores: Felipe Maia Galvão França
Gabriel Antoine Louis Paillard
Leandro Santiago de Araújo

Programa: Engenharia de Sistemas e Computação

Atualmente, a busca por desempenho computacional, tanto em aplicações de computação científica quanto em aplicações de uso geral, aponta para a computação paralela como área de interesse para sobrepor as dificuldades encontradas nos paradigmas computacionais utilizados tradicionalmente. Dentre os modelos computacionais paralelos onde programas podem ser desenvolvidos de maneira natural e transparente, Gamma e Dataflow apresentam uma surpreendente similaridade. Entretanto, a implementação do paradigma computacional Gamma apresenta vários desafios no que diz respeito ao escalonamento decorrente para a adequação às arquiteturas disponíveis. Neste trabalho demonstramos pela primeira vez a equivalência entre os modelos computacionais Gamma e Dataflow, onde além da apresentação da similaridade, apresentamos as provas formais de equivalência entre os modelos. Por ocasião desta equivalência, também propomos a implementação da primeira ferramenta de conversão entre os modelos Dataflow e Gamma, o *GFlow*. Diante dos desafios relacionados às implementações de Gamma, propomos o *GSink*. Trata-se da primeira implementação de um ambiente de execução de programas Gamma que permite a execução paralela de instâncias de diversas reações. Para tanto, utilizamos um mecanismo de escalonamento baseado em reversão de arestas de *sinks* de um grafo dirigido acíclico. Através de resultados experimentais demonstramos a corretude tanto das conversões propostas pelo *GFlow*, quanto dos resultados de execução do *GSink*.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

# EXPLORING THE EQUIVALENCE BETWEEN AN ABSTRACT CHEMICAL MACHINE AND DATAFLOW COMPUTING

Rui Rodrigues de Mello Junior

March/2022

Advisors: Felipe Maia Galvão França
Gabriel Antoine Louis Paillard
Leandro Santiago de Araújo

Department: Systems Engineering and Computer Science

Currently, the search for computational performance, both in scientific computing applications and general-purpose applications, points to parallel computing as an area of interest to overcome the existing difficulties in traditionally used computational paradigms. Among the parallel computational models where programs can be developed in a natural and transparent way, Gamma and Dataflow present a surprising similarity. However, the implementation of the Gamma computational paradigm presents several challenges with regard to the resulting scheduling to adapt to the available architectures. In this work, we demonstrate for the first time, the equivalence between Gamma and Dataflow computational models, where we present the similarity and the formal equivalence proofs between these two models. Due to this equivalence, we also propose the implementation of the first conversion tool between the Dataflow and Gamma models, called *GFlow*. Faced with the challenges related to Gamma implementations, we also propose *GSink*. It consists of the first implementation of a Gamma program execution environment that allows the parallel execution of instances of several reactions. For that, we use a mechanism based on scheduling by edges reversal of an acyclic directed graph. Through experimental results, we demonstrate the correctness of both the conversions proposed by *GFlow* and the results of *GSink* execution.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Lista de Siglas

| | |
|---|---|
| CMP | Chip Multiprocessor, p. 89 |
| DAG | Directed Acyclic Graph, p. 210 |
| DDG | Dynamic Dataflow Graph, p. 210 |
| DLP | Data-Level Parallelism, p. 12 |
| EP | Elemento de Processamento, p. 81 |
| ILP | Instruction-Level Parallelism, p. 12 |
| IPqM | Instituto de Pesquisas da Marinha, p. 3 |
| IoT | Internet of Things, p. 231 |
| MCC | Minimum Clique Covering, p. 112 |
| ML | Machine Learning, p. 32 |
| MPI | Message Passing Interface, p. 23 |
| PPDE | Par de Plots em Dois Estágios, p. 4 |
| SCUA | Sistema de Consciência Situacional Unificada por Aquisição de Informações Marítimas, p. 4 |
| SER | Scheduling by Edge Reversal, p. 112 |
| SLR | Systematic Literature Review, p. 231 |
| TALM | Architecture and Language for Multi-threading, p. 3 |
| TBB | Threading Building Blocks, p. 3 |
| THLL | TALM High-Level Language, p. 79 |
| TLP | Thread-Level Parallelism, p. 12 |

# Capítulo 1

# Introdução

## 1.1 Contextualização

Nota-se um aumento pela busca de desempenho computacional, onde tanto as aplicações de computação científica quanto aquelas de computação de propósito geral possuem tarefas que requerem alta capacidade computacional. Por outro lado, apesar do grande avanço no que diz respeito à tecnologia computacional, os computadores sequenciais aproximam-se do seu limite físico no que diz respeito ao desempenho de *hardware* [1, 2]. Por exemplo, o aumento de desempenho dos processadores esbarra na barreira física de capacidade de dissipação de calor e energia, entre outras limitações. Assim, a computação paralela surge como uma potencial alternativa para superar tais limitações. Entretanto, programar de forma paralela pode não ser uma tarefa simples. Realizar as atividades de distribuição e controle de tarefas entre processadores, controle de carga, gerenciamento de troca de mensagens entre os mesmos, além de realizar toda a modelagem do problema de forma a extrair o máximo de paralelismo, tornam bastante árdua e complexa a tarefa de programar de maneira paralela. Dessa forma, o uso de modelos onde as características de computação paralela podem ser expressas de uma forma simplificada tem assumido um lugar de destaque.

Dentre os modelos computacionais onde a exposição do paralelismo acontece de forma mais intuitiva, destaca-se o modelo Dataflow [3]. Em contraste com o modelo de von Neumann, onde a execução da computação é guiada pelo fluxo de controle, no modelo dataflow a execução do programa é dirigida pelos dados. Assim, a operação pode ser executada à medida em que seus operandos estiverem disponíveis e prontos. O modelo dataflow pode ser representado por um grafo direcionado, onde os vértices representam as operações a serem executadas e as arestas as dependências de dados entres estas operações. Dessa maneira, assim que um vértice possuir os dados necessários às suas arestas de entrada, a operação poderá ser executada.

Ainda neste contexto, surge o Gamma (***G**eneral **A**bstract **M**odel for **M**ultiset m**A**nipulation*), um formalismo para especificação de programas baseado na reescrita paralela de multiconjuntos. O conceito foi, inicialmente proposto em 1986 por BANÂTRE e LE MÉTAYER [4]. Em Gamma, a única base de dados existente é o multiconjunto, onde todos os dados utilizáveis são representados por elementos deste. O modelo de execução é não determinístico, uma vez que os elementos do multiconjunto podem reagir livremente e de maneira naturalmente paralela, tornando transparente ao programador detalhes inerentes à implementação do paralelismo. Gamma baseia-se em uma metáfora de reações químicas onde podemos enxergar o multiconjunto como uma solução química composta por diversas moléculas (elementos) e diversas ações (reações químicas) ocorrem de acordo com condições pré definidas (condição de reação). Dessa forma, Gamma coloca-se como importante ferramenta para superar os limites tecnológicos existentes por consistir em um modelo conceitualmente paralelo e que permite expressar problemas de uma maneira simples e natural.

Considera-se Gamma um modelo adequado para utilização em ambientes distribuídos, podendo ser visto como um modelo computacional onde o processamento permeia os recursos computacionais disponíveis [5]. Tal conceito tem por objetivo transformar Gamma em um modelo computacional paralelo eficiente onde conceitos como localidade e computação aproximativa poderão ser utilizados. Entretanto, algumas melhorias são necessárias para alavancar o desempenho das implementações de Gamma existentes. Por exemplo, as implementações de Gamma utilizadas por DE ALMEIDA *et al.* [6] possuem um gargalo computacional relacionado ao gerenciamento e distribuição do multiconjunto, gerado por uma política de gerenciamento centralizada e execução sequencial. Assim, investir em um mecanismo de escalonamento que permita execução paralela de reações apresenta uma boa alternativa para indicar caminhos de melhoria de desempenho do modelo.

O modelo computacional Gamma apresenta um potencial para utilização de técnicas de computação aproximada (Apêndice B). A possibilidade de interromper a computação antes que a mesma encontre um estado de terminação global pode ser utilizada em um ambiente Gamma, uma vez que as execuções das reações efetuam refinamentos sucessivos no multiconjunto. Da mesma forma, poderão ser utilizadas técnicas para relaxar a precisão da computação em um ambiente Gamma. Entretanto, é importante observar que existem domínios de aplicação específicos onde este tipo de equalização entre precisão e desempenho são possíveis. Dentre eles, podemos citar aplicações de processamento de imagens e fusão de dados.

Por outro lado, o modelo Dataflow apresenta diversos avanços significativos comparados ao paradigma Gamma, seja em viés teórico, quanto em viés prático. Estudos abordando execução especulativa e fora de ordem em um grafo dataflow [7], reuso

aplicado a traços de instruções dataflow [8], além de outros trabalhos desenvolvidos alavancaram a pesquisa relacionada ao modelo. Da mesma forma, diversas ferramentas foram propostas envolvendo o modelo Dataflow, como por exemplo, Intel® *Threading Building Blocks* (TBB) [9], *OpenMP* [10], *TensorFlow* [11, 12], *Fastflow* [13], arquitetura *TALM* [14], projeto *SUCURI* [15], entre outros.

Outro exemplo de área de atuação é a fusão de dados aplicada ao acompanhamento de contatos no meio militar naval [16, 17], onde o paralelismo coloca-se como solução viável para problemas complexos, podendo ser utilizado em um equilíbrio entre desempenho e precisão. Tal domínio de aplicação realiza aquisição de dados de diversos sensores. Assim, boa parte das informações recebidas (ruídos) são descartadas, o que faz com que a ideia de diminuição da precisão da computação possa ser levada em consideração em prol de um desempenho satisfatório.

Este trabalho propõe explorar benefícios advindos do estudo de equivalência entre os modelos computacionais Gamma e Dataflow. Desta maneira, avanços referentes à estudos já desenvolvidos para um paradigma poderão ser estendidos ao outro, de forma a tornar ambos modelos mais robustos.

## 1.2 Motivação

Com forte vocação em pesquisa e desenvolvimento de Sistemas Digitais Operativos para a Marinha do Brasil, o Instituto de Pesquisas da Marinha (IPqM), vem trabalhando ao longo de mais de cinco décadas em projetos aplicados a automação, controle e sistemas necessários para as tarefas operativas. Para atender as demandas específicas da Marinha do Brasil, como por exemplo, fornecer lógicas de controle de máquinas compatíveis com as necessidades e características de operação de nossos navios, além de aplicações que atendam aos requisitos de operação naval, se faz necessário ter o domínio de todas as camadas de software que compõem tais sistemas computacionais. Desta forma, é possível projetar, modelar e implementar funcionalidades que atendam exatamente às necessidades da Marinha do Brasil no que diz respeito a tais sistemas, garantindo independência tecnológica. No âmbito dos sistemas operativos, é comum a existência de diversas fontes de dados (sensores ativos ou passivos), capazes de detectar alvos (embarcações, aeronaves ou submarinos). Um dos problemas presentes na existência de múltiplas fontes de dados é a duplicação da informação, na medida em que um alvo pode ter sido detectado por mais de uma fonte. Desta forma, faz-se necessário correlacionar alvos de diferentes fontes, no tempo e no espaço, com algoritmos de Fusão de Dados complexos e custosos computacionalmente. A resolução do problema de acompanhamento de múltiplos contatos é limitada pela falta de capacidade computacional e de algoritmos paralelos eficientes [18].

Desta maneira, em 2016 [17] verificamos a adequabilidade do modelo computacional Gamma no que diz respeito à expressar um algoritmo de fusão de dados para acompanhamento de contatos utilizado por projetos desenvolvidos pela Marinha do Brasil, mais especificamente pelo IPqM. Trata-se da primeira implementação paralela do algoritmo de Par de Plots em dois Estágios (PPDE) [19]. O PPDE apresenta uma releitura de um algoritmo tradicionalmente utilizado para fusão de dados para acompanhamento de contatos, onde foi proposta a utilização de dois estágios de processamento, permitindo com que o algoritmo seja paralelizado [16]. Vale ressaltar que este algoritmo encontra-se implementado em alguns sistemas desenvolvidos pela Marinha, como por exemplo, o Sistema de Consciência Situacional Unificada por Aquisição de Informações Marítimas (SCUA). Assim, tendo em vista a adequabilidade e o potencial de Gamma em expressar problemas deste domínio de aplicação, surge uma das motivações deste trabalho, que consiste em investir em um modelo computacional vantajoso para um domínio de aplicação de interesse para a Marinha do Brasil.

Conforme veremos nos capítulos seguintes, Gamma consiste em um modelo não determinístico. Desta maneira, tanto a escolha dos elementos que serão utilizados como candidatos à reagir, quanto a execução das instâncias de reações, ocorre de maneira não determinística. Tal característica inerente ao modelo revela o potencial de Gamma para ser utilizado com técnicas de computação aproximativa [5]. Muitos sistemas e aplicações podem tolerar alguma perda de precisão em prol de desempenho, como por exemplo, processamento de mídia (áudio, vídeo e imagem), reconhecimento de padrões, mineração de dados e fusão de dados. Assim, pela característica de reescrita paralela de multiconjuntos presente no modelo Gamma, a computação pode ser finalizada antes que se alcance o estado de terminação global do programa, onde o multiconjunto pode ser visto a qualquer momento como uma solução parcial e aproximada do problema. Assim, explorar a característica não determinística do modelo também consiste em uma motivação, que também traz vantagens para outras áreas de pesquisa como Algoritmos Evolutivos, Algoritmos Genéticos e Otimização.

Gamma possui potencial para utilização em ambientes distribuídos [5], devido a possibilidade de distribuição do multiconjunto pela rede, explorando o conceito de localidade, onde os dados possam ser processados próximo ao local onde residem. Além da possibilidade de execução distribuída de reações, tendo em vista uma equalização entre a computação que será executada e as capacidades dos dispositivos disponíveis. Desta forma, conceitos como *Edge Computing* [20] e *In Situ Computing* [21] podem vir a ser adequados ao modelo.

Tendo em vista a visualização de Gamma como plataforma computacional onde a computação permearia os recursos computacionais disponíveis, em 2020 propusemos

uma ambiente de execução de programas Gamma onde a arquitetura foi baseada em um grafo dataflow [5]. Desta forma Gamma pode ser visto como modelo computacional adequado para IoT e Sistemas Altamente Heterogêneos, onde, em ambos os casos, os recursos computacionais disponíveis apresentam diferenças consideráveis em termo de capacidade computacional, consumo energético, entre outros.

Por fim, a possibilidade de extensão de benefícios entre os modelos Gamma e Dataflow, materializada através do estudo da equivalência é considerada por si só uma motivação, uma vez com que contribui para alavancar a pesquisa e identificar diversos trabalhos futuros em ambos os modelos envolvidos. Além disso, a maneira concisa e simples em programar Gamma em um ambiente naturalmente paralelo e que se demonstrou bastante adequado para alguns domínios de aplicação, também constitui uma importante motivação para o desenvolvimento desta tese.

## 1.3 Etapas da Pesquisa

O modelo computacional Gamma fornece a possibilidade de expressar problemas de uma maneira simples e naturalmente paralela. Ou seja, detalhes de implementação do paralelismo são transparentes para o programador. Gamma fornece um estilo de programação mais robusto que os paradigmas tradicionais, do ponto de vista do desenvolvimento de programas paralelos, uma vez que não existe a preocupação com restrições de sequencialidade. Gamma foi utilizado para expressar problemas reais da Marinha do Brasil, no domínio de aplicação de fusão de dados para classificação de contatos [17]. Dessa forma, a inspiração deste trabalho surge em investir em um modelo computacional onde problemas complexos de fusão de dados puderam ser expressos paralelamente de maneira transparente e menos complexa.

Por outro lado, dentre as implementações de Gamma existentes, verificamos gargalos computacionais no que diz respeito à centralização do gerenciamento do multiconjunto. Desta forma, PAILLARD *et al.* [22], propuseram um novo escalonador para Gamma baseado em um mecanismo de reversão de arestas.

Gamma e Dataflow apresentam uma surpreendente similaridade. Assim, MELLO *et al.* [23] apresentaram pela primeira vez a equivalência entre os modelos computacionais Gamma e Dataflow, onde um programa escrito em Gamma poderia ser convertido em um grafo dataflow aproveitando uma série de benefícios advindos dos estudos desenvolvidos para este paradigma computacional, além de contribuir para ambos os modelos e para a versatilidade no desenvolvimento de programas paralelos. Vale ressaltar que tal estudo de equivalência apresenta transformações de um programa Gamma em um grafo dataflow e vice e versa, fornecendo a possibilidade de transformação entre os dois modelos de maneira bidirecional.

Diante da percepção e apresentação da similaridade entre os modelos, realizamos uma extensão do trabalho inicial, onde propusemos a prova formal de equivalência entre os modelos citados [24].

Tendo em vista a adequabilidade de Gamma para aplicações de fusão de dados aplicados ao ambiente militar naval, seu potencial para ser utilizado em ambientes distribuídos, a possibilidade de investir em um ambiente que proponha algumas soluções para os obstáculos enfrentados pelas implementações de Gamma, e benefícios advindos do estudo de equivalência, em [5] propusemos a *"Fluid Computing"*. Trata-se de uma proposta de arquitetura para execução de programas Gamma onde o modelo de execução era baseado em um grafo dataflow que utilizava um protocolo de comunicação baseado em interesses. A intenção foi investir em Gamma como um modelo computacional que permeasse os recursos computacionais disponíveis. Entretanto, algumas características da atual implementação do protocolo baseado em interesses utilizado, postergaram a proposta inicial.

Assim, a intenção inicial e mais óbvia voltou a tomar grandes proporções: investir na implementação de um ambiente de execução para programas Gamma baseado em [22]. O mecanismo de execução era bastante promissor e, pela primeira vez dentre as implementações de Gamma fornecidas, a possibilidade de execução de instâncias de diversas reações poderia ser fornecida, num modelo onde a coexistência de tais instâncias ocorre através de um grafo acíclico dirigido. Surge assim uma consequência deste estudo, o *GSink*, um *Runtime* para Gamma que implementa as características acima descritas.

Entretanto, o estudo de equivalência conceitual carecia de uma implementação. A implementação de uma conversão entre os modelos Dataflow e Gamma seria interessante para a versatilidade e aumento de expressividade de Gamma. Dessa forma, em conjunto com a intenção em fornecer a implementação do *GSink* surge a motivação em desenvolver o *GFlow*: uma ferramenta de conversão Dataflow-Gamma.

Tendo em vista o exposto, a Figura 1.1 apresenta um resumo dos eventos e decisões que deram origem às etapas de pesquisa que compõem este estudo. O desencadeamento dos fatos teve origem por ocasião da dissertação de mestrado que explorou a implementação de um algoritmo de fusão de dados segundo o paradigma Gamma [18].

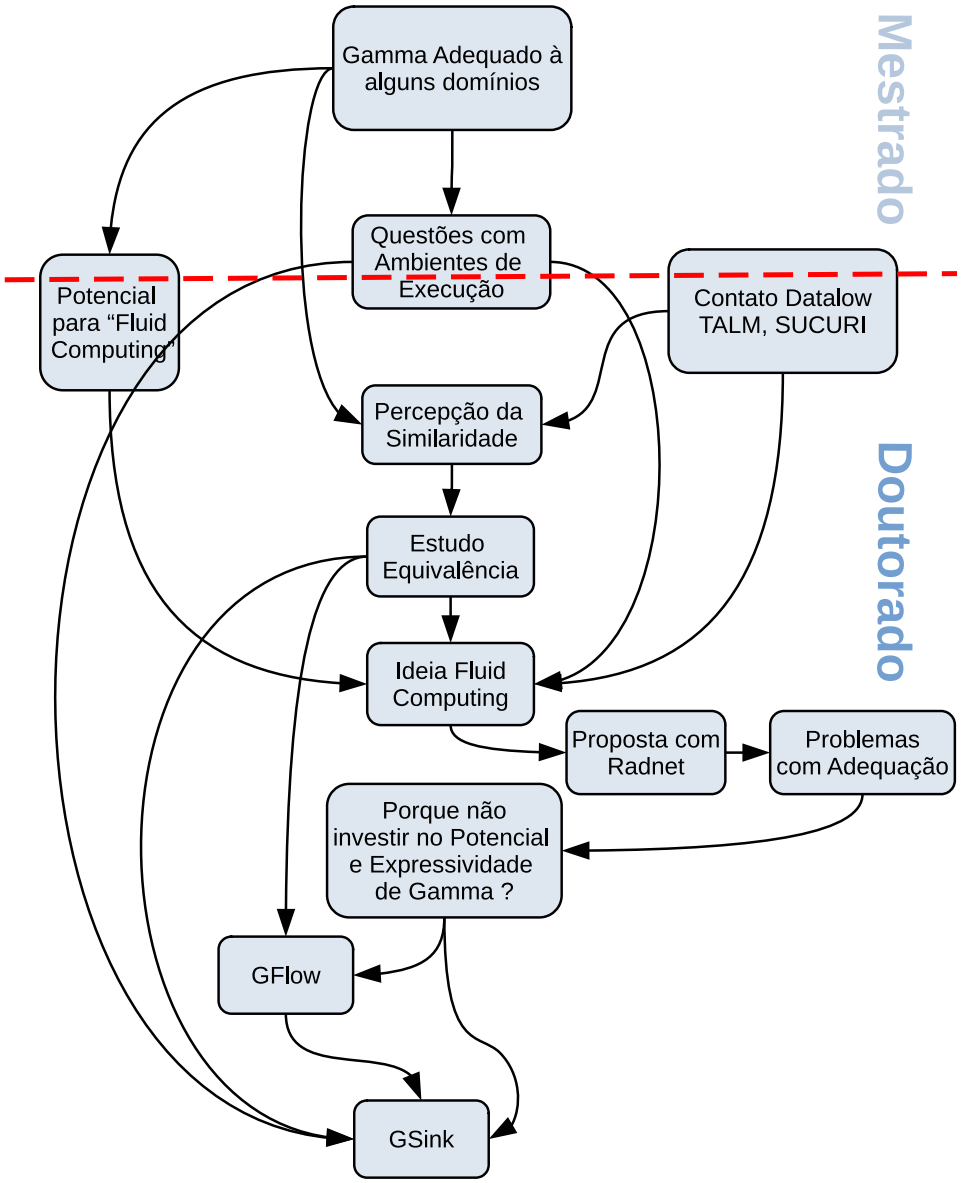


Figura 1.1: Resumo dos eventos que deram origem às etapas de pesquisa desta tese.

## 1.4 Objetivos e Contribuições

Como objetivo geral deste trabalho de pesquisa, pretende-se investir no paradigma computacional Gamma de forma a explorar sua equivalência com o modelo dataflow. Para tanto, podemos elencar os seguintes objetivos específicos:

- Apresentar a similaridade entre os modelos Gamma e Dataflow;
- Fornecer a prova formal de equivalência entre os modelos citados;
- Apresentar a implementação de uma ferramenta de conversão entre os modelos envolvidos; e

- Apresentar a implementação de um ambiente de execução para códigos Gamma.

O trabalho apresentado neste estudo contribui tanto para o modelo Gamma quanto para o modelo Dataflow, uma vez que a equivalência apresentada permite que estudos desenvolvidos para um modelo possam vir a ser utilizados pelo outro modelo, como por exemplo execução de códigos Dataflow de maneira especulativa e fora de ordem [7] e reutilização de traços de instruções [25].

A apresentação da similaridade e prova formal de equivalência entre Gamma e Dataflow já são contribuições importantes para a comunidade científica, pois trata-se de trabalhos inovadores que permitem a versatilidade na programação no que diz respeito a escolha do modelo mais adequado para determinada situação. Por exemplo, Gamma mostrou-se bastante adequado para utilização em um contexto de fusão de dados para classificação de contatos em um ambiente militar naval [17].

Tendo em vista materializar as contribuições realizadas por ocasião do estudo de equivalência, o *GFlow* consiste na primeira implementação da conversão entre os modelos Dataflow e Gamma. Tal fato contribui não só para a versatilidade no desenvolvimento de programas paralelos mas também traz maior expressividade ao modelo. Além disso, a proposta do *GFlow* também contribui para o modelo Dataflow, uma vez que permite com que este modelo computacional possa vir a se beneficiar dos avanços realizados para o paradigma Gamma.

A proposta da implementação de um ambiente de execução que permita com que instâncias de diversas reações coexistam e sejam executadas de maneira paralela contribui sobremaneira para o modelo Gamma. Assim, o *GSink* consiste na primeira implementação a contemplar tais características, que visam sobrepujar alguns obstáculos e gargalos computacionais de implementações de Gamma anteriores. Apesar de nossa implementação não ter o objetivo inicial de extração de desempenho, o método de escalonamento utilizado é bastante promissor e abre um novo horizonte de pesquisas para ambientes de execução.

Por fim, o modelo computacional Gamma tem potencial para ser utilizado em um ambiente onde a computação permeie os recursos computacionais disponíveis, permitindo o escalonamento dinâmico e transparente de uma aplicação alvo. Desta forma, uma série de classes de aplicações poderiam ser beneficiadas, como por exemplo, fusão de dados para acompanhamento de contatos. Conforme visto na Seção 1.2, o investimento no modelo Gamma tende a trazer benefícios aos algoritmos de fusão de dados utilizados e desenvolvidos pela Marinha do Brasil. Desta forma, a realização deste trabalho também contribui para os sistemas presentes neste domínio de aplicação de interesse da Marinha.

## 1.5 Estrutura do Texto

No segundo Capítulo deste trabalho iremos abordar uma revisão dos principais conceitos e modelos abordados nesta pesquisa. Desta forma veremos os detalhes inerentes ao formalismo Gamma, iniciando pelos principais conceitos envolvidos. O estilo de programação Gamma será explicitado com alguns exemplos de programas típicos. Após isto, veremos alguns padrões de transformações que ocorrem no multiconjunto, finalizando com o estado da arte em Gamma. Ainda no Capítulo 2 abordaremos o modelo dataflow. Serão abordados os principais conceitos e características do modelo necessários ao entendimento e futura aplicação nos estudos de equivalência, como por exemplo: tipos de arquitetura dataflow e considerações sobre detalhes importantes do modelo.

A equivalência entre os modelos computacionais Gamma e Dataflow, será apresentada no Capítulo 3, onde veremos a ideia principal por trás da similaridade, estudos iniciais visando apresentar a redução de códigos e algoritmos de conversão entre os modelos citados. Finalizando este terceiro Capítulo, apresentaremos uma extensa prova formal de equivalência entre os modelos computacionais Gamma e Dataflow.

No quarto capítulo, apresentaremos as duas implementações propostas. Inicialmente apresentaremos o *GFlow*, iniciando por uma revisão sobre detalhes necessários a respeito da arquitetura TALM (*TALM is an Architecture and Language for Multi-threading*). Em seguida apresentamos uma descrição da ferramenta, passando por benefícios, contribuições e considerações sobre a implementação, onde aspectos mais técnicos serão apresentados. Posteriormente, ainda no Capítulo 4, apresentamos o *GSink*. Serão apresentadas algumas considerações sobre as implementações de Gamma anteriormente fornecidas. Após isso, descrevemos o *GSink*, seu mecanismo de escalonamento, benefícios, contribuições e detalhes à respeito da implementação da ferramenta.

O Capítulo 5 é dedicado aos experimentos fornecidos e resultados obtidos, tanto para o *GFlow*, quanto para o *GSink*. Neste caso, o foco maior de nossos experimentos foi a corretude, tanto das conversões e resultados da computação apresentados pelo *GFlow*, quanto na montagem do código a ser executado pelo *Runtime* do *GSink*. Entretanto, para o *GSink*, fornecemos ainda uma segunda categoria de experimentos voltados à exploração do potencial de desempenho de nosso mecanismo de escalonamento.

As conclusões, resumo das contribuições e indicação de trabalhos futuros são fornecidos no sexto Capítulo desta tese.

No Apêndice A a listagem de publicações produzidas durante a realização desta tese será apresentada, seguida de experimentos que demonstram o potencial de Gamma a ser utilizado com técnicas de computação aproximativa, no Apêndice B. Por fim, o Apêndice C apresenta uma proposta de arquitetura publicada em 2020.

# Capítulo 2

# Fundamentação Teórica

Este Capítulo será dedicado a rever os principais conceitos e informações inerentes aos modelos computacionais estudados por ocasião do desenvolvimento desta tese. Desta forma, na Seção 2.1, iremos abordar o paradigma Gamma e posteriormente o modelo computacional dataflow, na Seção 2.2.

## 2.1 Gamma

A presente seção é dedicada a uma abordagem mais minuciosa do paradigma computacional Gamma, onde falaremos sobre os principais conceitos inerentes ao modelo computacional em questão, abordaremos detalhes do estilo de programação Gamma e finalizaremos com o estado da arte, elencando os principais trabalhos e aplicações que utilizaram Gamma como base.

### 2.1.1 Contextualização

Nos últimos 70 anos, a tecnologia computacional vivenciou incríveis progressos, desde que o primeiro computador de propósito geral foi proposto, conforme afirmado por HENNESSY e PATTERSON [1]. Entretanto, de acordo com a revisão da lei de Moore de 1965, atualmente os computadores sequenciais aproximam-se de seu limite físico de desempenho. Neste contexto, pode-se citar três importantes limitações ao aumento de desempenho [2]: ***Power Wall***: onde o aumento do desempenho dos processadores encontra-se limitado pela capacidade de dissipação de calor e energia; ***Memory Wall***: a eficiência das operações de transferência de dados entre memória e processador limitam o incremento de desempenho computacional; e ***Frequency Wall***: A frequência do cristal oscilador limita o aumento de desempenho computacional [2].

Por outro lado, o modelo sequencial de computação baseado na arquitetura proposta por von Neumann [26], teve um importante papel no projeto de linguagens de programação criadas no passado, basicamente por dois motivos [27]: modelos sequenciais apresentavam uma adequada abstração dos algoritmos, uma vez que os representavam intuitivamente como uma "receita" a ser seguida; e implementações eram realizadas sobre arquiteturas compostas por um único processador, refletindo tal visão sequencial abstrata.

Dessa forma, o paradigma imperativo de programação é o mais utilizado por tais linguagens, que utilizam um operador de sequencialidade para produção de programas. Tal operador foi criado pois as linguagens imperativas constituem uma abstração para a arquitetura de von Neumann, transformando o programa numa listagem sequencial de instruções a serem executadas, o que nem sempre traduz a característica do problema a ser solucionado e sim o modelo computacional empregado. Visando maior confiança e menor quantidade de erros na construção de programas, é importante que os mesmos sejam desenvolvidos inicialmente com foco no problema a ser resolvido e somente depois detalhes inerentes à linguagem ou à arquitetura subjacente sejam levados em consideração.

Para que esta derivação sistemática possa ser atingida, o programador deve ser capaz de expressar uma versão abstrata do problema a ser solucionado em uma linguagem de alto nível, livre de artificialidades sequenciais. Em um segundo momento, tal versão abstrata do problema poderá ser implementada em vários tipos de arquiteturas, onde então as imposições de sequencialidade serão levadas em consideração. Um exemplo de uma tentativa inicial de um formalismo de alto-nível não-imperativo para especificação de programas foram as linguagens funcionais. Entretanto, elas fazem uso maciço de recursividade, tanto nas estruturas de dados quanto nos programas, o que pode ser encarado como uma forma disfarçada de utilização de sequencialidade [28]. Esta dificuldade de separar a expressão do problema a ser resolvido de sua implementação propriamente dita, faz também com que a tarefa de programar paralelamente seja bastante complicada, pela necessidade de gerenciamento de várias linhas de controle sequenciais ao mesmo tempo.

Dessa forma, a computação paralela surge como um dos mecanismos propostos, tanto para fornecer uma abstração para especificação de programas, quanto para superar as limitações descritas anteriormente, através de estudos para explorar o paralelismo a nível de *threads* (*TLP - Thread-Level Parallelism*), instruções (*ILP - Instruction-Level Parallelism*) e dados (*DLP - Data-Level Parallelism*) [1].

### 2.1.2 O paradigma Gamma

Neste contexto surge o Gamma, acrônimo para **G**eneral **A**bstract **M**odel for **M**ultiset m**A**nipulation, inicialmente proposto por BANÂTRE e LE MÉTAYER [4] em 1986. Gamma pode ser definido como um formalismo para especificação de programas, baseado na reescrita paralela de multiconjuntos. Tal especificação de programas ocorre de maneira elegante e naturalmente paralela através da utilização do operador Gamma ($\Gamma$), inspirado na disciplina de programação de DIJKSTRA [29]. Assim, a principal característica de um programa em Gamma é que este consiste em um modelo de execução não determinístico, uma vez que os elementos deste multiconjunto podem interagir livremente, de forma naturalmente paralela e sem nenhum tipo de restrição de ordem no acesso a valores, introduzindo o chamado modelo caótico de execução [30]. Portanto, Gamma propõe um formalismo que possibilita a abstração dos detalhes que tornam difícil a tarefa de desenvolver programas em linguagens de programação paralelas clássicas ou tradicionais.

O modelo computacional Gamma baseia-se em uma metáfora de reações químicas, onde existe uma base de dados única, chamada de multiconjunto (solução química) composta por diversos elementos (moléculas). Diversas ações (reações químicas) são especificadas para serem executadas sobre os elementos do multiconjunto, de acordo com uma série de condições (condição de reação). Assim, a computação ocorre como uma sucessão de reações químicas que ocorrem com as moléculas, caso algumas regras para as reações sejam cumpridas. Tais regras indicam os tipos de moléculas que podem reagir conjuntamente, bem como o resultado produzido pela reação. A ideia é que as moléculas possam interagir livremente, com as reações podendo ocorrer a qualquer ordem, inclusive ao mesmo tempo, caso as condições apropriadas estejam presentes. A computação termina quando nenhuma reação puder mais ocorrer sobre nenhum subconjunto de moléculas, momento no qual atinge-se uma solução química estável, representando o término da execução do programa.

A estrutura de dados básica no Gamma é o multiconjunto (*Multiset* ou *Bag*), que consiste na mesma estrutura conhecida como conjunto (*set*), exceto pelo fato de ser permitida a existência de múltiplas ocorrências de um mesmo elemento. O multiconjunto não possui nenhuma restrição no acesso aos seus elementos, nem impõe nenhum tipo de hierarquia entre os mesmos, possibilitando uma representação ideal para os dados em Gamma.

Um programa típico em Gamma consiste, basicamente, em pares de funções formadas por condições/ações, executados sobre os elementos do multiconjunto. A execução ocorre através da modificação do multiconjunto pela exclusão, inclusão e transformação dos elementos existentes. Conforme vimos, neste modelo, o fim da computação ocorre quando um estado estável é alcançado, o que corresponde a um

estado onde todas as reações terminaram suas execuções, ou seja, nenhuma reação consegue mais cumprir as condições para reagir.

Esta estrutura de controle que torna possível a execução de um programa segundo o paradigma Gamma é o operador Γ, que, conforme [31], pode ser formalmente definido por:

$$\begin{aligned} &\Gamma((R_1, A_1), ..., (R_m, A_m))(M) = \\ &if\ \forall\ i \in [1, m], \forall\ x_1, ..., x_n \in M, \neg R_i(x_i, ..., x_n) \\ &\qquad then\ M \\ &else\ let\ x_1, ..., x_n \in M, let\ i \in [1, m]\ such\ that\ R_i(x_i, ..., x_n)\ in \\ &\Gamma((R_1, A_1), ..., (R_m, A_m))((M - \{x_1, ..., x_n\}) + A_i(x_1, ..., x_n)) \end{aligned} \tag{2.1}$$

Os pares de funções $(R_i, A_i)$, são aplicadas ao multiconjunto $(M)$ e especificam as reações a serem utilizadas e suas condições para execução. A aplicação do par $(R_i, A_i)$ em M resulta em substituir em $M$ um subconjunto de elementos $(x_1, ..., x_n)$ tal que $R_i(x_1, ..., x_n)$ seja verdadeiro pelos elementos de $A_i(x_1, ..., x_n)$. Caso nenhum elemento satisfaça a condição de reação, o resultado da computação é o próprio $M$ inicial. Caso contrário, o resultado é o multiconjunto $M$ subtraído do subconjunto de elementos $(x_1, ..., x_n)$ acrescido do subconjunto especificado pela condição de reação $A_i(x_1, ..., x_n)$, ou seja, $((M - \{x_1, ..., x_n\}) + A_i(x_1, ..., x_n))$. Enquanto existirem condições verdadeiras para a execução das reações existentes, estas irão sendo executadas (reagindo) e consequentemente transformando o multiconjunto. É importante perceber que se uma ou mais condições de reações forem satisfeitas para alguns subconjuntos do multiconjunto simultaneamente, a decisão de qual reação irá executar ocorre de maneira não determinística, uma vez que estas reações poderão ser executadas independentemente e simultaneamente, o que confere ao modelo computacional Gamma uma característica naturalmente paralela [31].

### 2.1.3 Estilo de programação

Agora abordaremos o estilo de programação Gamma. Considere como objeto de estudo o trecho de código abaixo, que corresponde a um programa em Gamma que encontra o menor elemento de um multiconjunto:

$$
\begin{aligned}
Menor(M) = \Gamma(R, A)\ \ (M)\ \ where \\
R(x, y) = x \geq y \\
A(x, y) = y
\end{aligned}
\tag{2.2}
$$

Aqui, ao comparar elementos dois a dois $(x, y)$, caso o elemento $x$ seja maior ou igual a $y$, $x$ é retirado do multiconjunto e $y$ é mantido. Caso contrário ($x < y$), nenhuma ação é executada no multiconjunto.

Tomemos como exemplo o multiconjunto formado pelos seguintes elementos: $\{4, 15, 7, 8, 5\}$. Uma possível execução do programa para este multiconjunto seria:

***Passo 1*** - após a escolha dos elementos $(15, 7)$, como par $(x, y)$, o elemento 15 seria excluído e o 7 mantido no multiconjunto, pois $x \geq y$, tendo como resultado parcial o multiconjunto $\{4, 7, 8, 5\}$.

***Passo 2*** - Da mesma forma ao selecionar $(8, 7)$, teríamos $\{4, 7, 5\}$.

***Passo 3*** - Agora, caso o par $(4, 7)$ tenha sido escolhido, nenhuma operação seria realizada sobre o multiconjunto, uma vez que a condição de reação $(x \geq y)$, não foi satisfeita.

***Passo 4*** - Após a seleção dos elementos $(7, 4)$ o multiconjunto seria transformado para $\{4, 5\}$ e, finalmente, seria reduzido a $\{4\}$, quando o par $(5, 4)$ fosse utilizado como par $(x, y)$ - ***Passo 5***.

Conforme mencionado anteriormente, a interação entre os elementos é livre e ocorre paralelamente, de forma que diversos pares $(x, y)$, poderiam ter sido comparados ao mesmo momento. A Figura 2.1, ilustra a execução do exemplo descrito acima.

Observando o exemplo descrito acima, repare que a definição do programa (expresso em 2.2) não diz nada a respeito da ordem em que os elementos serão selecionados para a realização das comparações. Dessa forma, se vários pares disjuntos de elementos satisfazem a condição, as reações podem ser realizadas em paralelo. Nas linguagens de programação mais tradicionais, a primeira decisão a ser tomada para se implementar este algoritmo seria a respeito de qual estrutura de dados utilizar para armazenamento dos elementos do multiconjunto. Nas linguagens imperativas a escolha mais usual seria utilizar um array, já nas linguagens declarativas poderia ser uma lista. O programa seria definido como uma iteração pelo *array*, ou por uma caminhada recursiva pela lista. Em qualquer um dos casos, a estrutura utilizada imporia restrições na ordem que os elementos são acessados, apresentando um contraste com a ideia de Gamma, onde o multiconjunto não possui qualquer tipo de ordem imposta, possibilitando que os elementos atômicos interajam livremente [32].

Já o exemplo constante em [28] e descrito a seguir, demonstra de uma maneira

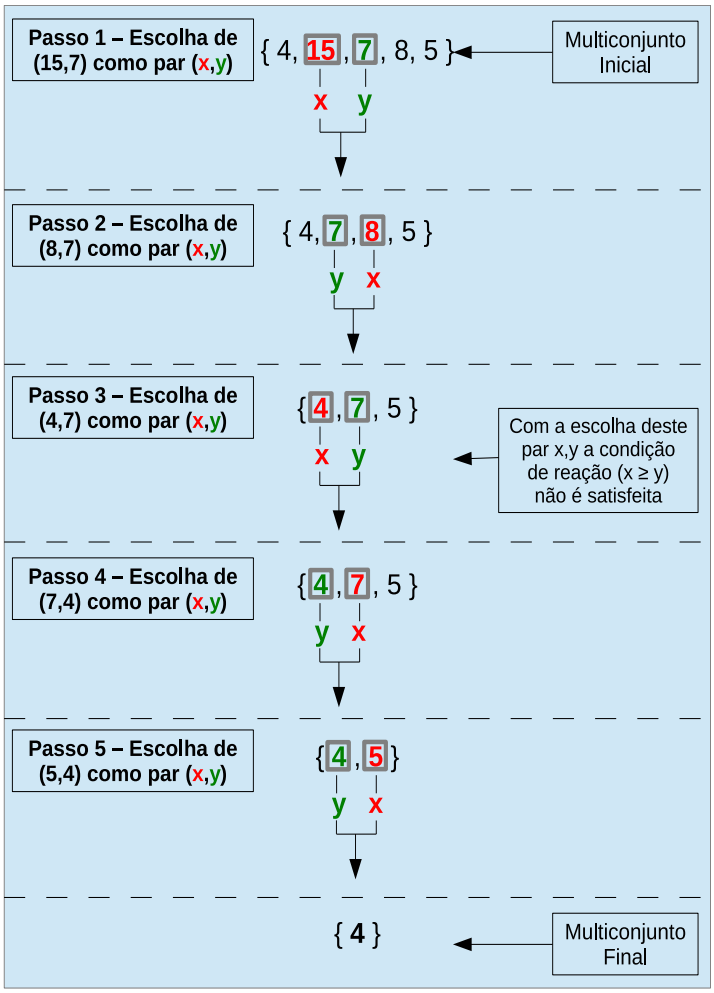


Figura 2.1: Exemplo da execução de um programa em Gamma.

elegante e concisa um programa em Gamma que se propõe a encontrar os números primos menores ou iguais a N:

$$\begin{aligned} Prime(M) = \Gamma(R, A)\ (\{2, ..., N\})\ \ where \\ R(x, y) = multiple(x, y) \\ A(x, y) = y \end{aligned} \tag{2.3}$$

A função *multiple* é verdadeira se e somente se $x$ for múltiplo de $y$. Portanto, caso tal condição seja satisfeita, $x$ é retirado do multiconjunto. Assim, números primos não serão descartados e, ao final da execução, restarão somente os números primos menores ou iguais a $N$, no multiconjunto $\{2, ..., N\}$.

### 2.1.4 Padrões de projeto de reações

Como vimos nas seções anteriores, o multiconjunto é a única representação dos dados em Gamma. Este pode ser formado por elementos que possuem valores simples ou compostos, como por exemplo tuplas [17]. Dessa forma, o resultado da computação é obtido através de operações sobre esta base de dados. Tais operações transformam o multiconjunto através de métodos de Relaxamento (*Relaxation*), Expansão (*Data Expansion*) e Redução de Dados (*Data Reduction*) [33].

Relaxamento consiste em um método onde o multiconjunto vai se transformando a cada iteração, correspondendo a eventuais resultados preliminares. Assim, a cada iteração, os elementos indesejáveis vão sendo excluídos até que se alcance um resultado final. Expansão de dados é uma técnica que corresponde a decompor os elementos iniciais em elementos indivisíveis, incrementando a quantidade de elementos do multiconjunto. Já a redução seria a operação antagônica à expansão. Ou seja, reduzir o multiconjunto a um único elemento. BANÂTRE e LE MÉTAYER [33] ilustram estas técnicas com um exemplo que calcula o enésimo termo de uma série de Fibonacci. Para tanto um número inicial $N$ é decomposto em outros números (Expansão), que são somados para produzir o resultado esperado (Redução).

Assim, tendo em vista os métodos acima citados para transformação de um multiconjunto, são apresentados cinco esquemas básicos que traduzem alguns padrões recorrentes em programas Gamma: ***T****ransmuter*, ***R****educer*, ***OP****timizer*, ***E****xpander* e ***S****elector*, que juntos formam o acrônimo *TROPES* ([32] e [27]).

***Transmuter***: aplica a mesma operação em todos os elementos do multiconjunto enquanto a condição de reação for verdadeira. Por exemplo, imaginemos um multiconjunto cujos elementos são formados por tuplas do tipo $(x, y, s)$. Uma operação que corresponde ao padrão *transmuter* seria, por exemplo, uma reação que calcule $s = x + y$ para todas as tuplas que estão contidas neste multiconjunto [1].

$$T(C, f) = x \longrightarrow f(x) \Longleftarrow C(x) \tag{2.4}$$

***Reducer***: corresponde exatamente à Redução de Dados (*Data Reduction*) explicada anteriormente. Um padrão onde o tamanho do multiconjunto é reduzido. Por exemplo, para um multiconjunto formado pelas mesmas tuplas do tipo $(x, y, s)$, uma reação que ao comparar um par de tuplas $(x1, y1, s1)$ e $(x2, y2, s2)$, substitua as duas tuplas por uma do tipo $(x1 + x2, y1 + y2, s1 + s2)$.

[1] As definições formais apresentadas para o padrão TROPES foram retiradas de [32].

$$R(C, f) = x, y \longrightarrow f(x, y) \Longleftarrow C(x, y) \tag{2.5}$$

***OPtimizer***: tem por objetivo otimizar o multiconjunto de acordo com um critério específico, sem alterar a estrutura do multiconjunto. É parecido com o padrão *transmuter*, mas ao contrário deste, só transforma os elementos que atendem a um critério determinado além de não alterar a quantidade nem a estrutura dos elementos do multiconjunto. Tomando o mesmo exemplo do padrão *transmuter*, imaginemos uma reação que calcule $s = x + y$, para tuplas do tipo $(x, y, s)$ mas se e somente se a condição $x > y$ for satisfeita.

$$O(<, f_1, f_2, S) = x, y \longrightarrow f_1(x, y), f_2(x, y) \Longleftarrow (f_1(x, y), f_2(x, y)) < (x, y) \\ and\ S(x, y)\ and\ S(f_1(x, y), f_2(x, y)) \tag{2.6}$$

***Expander***: correspondente à Expansão de Dados (*Data Expansion*) mencionada no segundo parágrafo desta seção. Consiste em decompor os elementos de um multiconjunto em elementos básicos. Um exemplo poderia ser construído por uma reação que decompõe cada tupla $(x, y, s)$ em elementos $x$, $y$ e $s$.

$$E(C, f_1, f_2) = x \longrightarrow (f_1(x), f_2(x)) \Longleftarrow C(x) \tag{2.7}$$

***Selector***: já o padrão *selector* remove elementos específicos de um multiconjunto, agindo como um filtro. Por exemplo, uma reação que elimina do multiconjunto toda e qualquer tupla $(x, y, s)$ cujo valor de $s$ seja maior que um dado valor $N$.

$$S_{i,j}(C) = x_1, ..., x_i \longrightarrow x_j, ..., xi \Longleftarrow C(x_1, ..., x_i) \\ (where\ 1 < j \leqslant (i + j)) \tag{2.8}$$

### 2.1.5 Estado da Arte

Até agora vimos os principais conceitos e mecanismos que definem o modelo computacional Gamma e seu estilo de programação. Nas seções a seguir veremos os principais trabalhos baseados em Gamma e algumas importantes implementações.

#### 2.1.5.1 Extensões

Com o passar do tempo, foram sendo verificadas algumas deficiências do Gamma, dentre as quais podemos relacionar as seguintes como principais [34]:

- Inexistência de operadores que permitam combinação de programas;
- Dificuldade para estruturação de dados ou especificação de estratégias de controle; e
- Dificuldade de alcançar um bom nível de eficiência em qualquer implementação da linguagem, devido à inexistência de um modelo arquitetural que comporte a explosão combinatorial imposta pela sua semântica.

Tendo em vista estas deficiências encontradas, foram propostas algumas extensões linguísticas ao Gamma original que visam incrementar a capacidade de representação sintática e semântica do formalismo, das quais as principais serão explanadas nas seções seguintes.

**– Gamma Estruturada**

A proposta de Gamma é fornecer um formalismo para especificação de programas com a menor quantidade de restrição possível. Dessa maneira, numa primeira análise, o fato de não fornecer dados estruturados nem estruturas de controle parece estar bastante adequado para a proposta de Gamma. Entretanto, quando surge a necessidade de modelar uma estrutura de dados específica para a resolução de algum problema, como por exemplo, o uso de uma lista encadeada ou uma árvore, a tarefa de programar em Gamma torna-se mais complexa, e o código resultante também acaba se tornando complexo e menos legível.

Desta forma, em 1997, foi proposta uma extensão do Gamma, chamada de *Structured Gamma*, por FRADET e LE MÉTAYER [34]. Basicamente o Gamma Estruturado fornece um multiconjunto estruturado e consequentemente suporte a tipos.

Um multiconjunto estruturado possibilita uma ordenação de endereçamento entre seus elementos, o que preserva o princípio da localidade, já que os dados são independentes, evitando que o multiconjunto tenha que ser manipulado por inteiro. Assim, se condições de reações forem satisfeitas por diferentes conjuntos de elementos do multiconjunto, reações podem ocorrer de forma independente e paralela. Tal

estruturação de multiconjuntos funciona como uma espécie de vizinhança entre as moléculas de uma solução química. Em outras palavras, multiconjuntos estruturados são uma facilidade sintática usada para tornar a organização dos dados explícita [34].

Já o suporte a estruturação de dados existente nesta extensão ao Gamma original, fornece ao programador uma maneira de criar suas estruturas de dados de forma concisa através de uma gramática livre de contexto. Dessa forma é possível representar a maioria das estruturas de dados existentes. A utilização de tipos garante que a estrutura do multiconjunto esteja consistente com o tipo de dado representado, garantindo assim que a estrutura definida não se degenere [31].

**– Composição de Operadores**

Os exemplos apresentados até aqui utilizam somente uma reação para expressar a computação necessária à um programa Gamma. Entretanto, visando maior modularidade e melhor estruturação para expressão de alguns problemas, é desejável que exista alguma forma de conectar mais de uma reação o que permite a produção de programas mais complexos a partir de programas mais simples, além de permitir algum tipo de restrição no acesso ao multiconjunto, o que pode vir a ser útil em alguns casos.

Desta forma, uma importante extensão do Gamma introduziu os conceitos de operadores sequenciais e paralelos. Em [35], são abordados os conceitos e propriedades de tais operadores.

HANKIN *et al.* [35] utilizam a notação $P_1 \circ P_2$ para expressar sequencialidade e $P_1 + P_2$ indicando paralelismo, onde $P_1$ e $P_2$ representam duas reações quaisquer. No primeiro caso, $P_1 \circ P_2$, o multiconjunto estável obtido pela execução de $P_1$ é passado como argumento para a reação $P_2$, indicando uma dependência na execução destas reações. Já a composição $P_1 + P_2$ indica a independência entre $P_1$ e $P_2$ uma vez que estas podem ocorrer em qualquer ordem, possivelmente em paralelo, e que a computação termina quando nenhuma das duas reações puderem mais reagir.

Outras composições semânticas da linguagem foram propostas, como por exemplo em [36, 37].

**– High Order Gamma**

Como vimos anteriormente, a utilização de operadores de sequencialidade e paralelismo proporciona ao programador a possibilidade de desenvolvimento de melhores estratégias de controle para programas em Gamma. Dessa forma, uma outra abordagem para introdução de composição de operadores em uma linguagem consiste

em fornecer ao programador uma maneira de defini-los como programas de ordem superior [27] (*Higher Order Programs*). Dessa maneira, a primeira extensão de ordem superior de Gamma foi proposta em 1994 por MÉTAYER [38]. Esta inspiração de escrever programas de alta ordem vem das linguagens funcionais, as quais implementam a manipulação de programas como se estes fossem dados comuns.

Até agora, Gamma utilizava, basicamente, dois tipos de termos: multiconjuntos e programas. Por programa, entende-se um conjunto de regras de reescrita de multiconjuntos, ou seja, um programa consiste numa coleção de pares que envolvem condições de reação e ações a serem executadas. Já o multiconjunto é a base de dados única do modelo computacional em pauta. A proposta desta extensão de ordem superior é unificar estas duas categorias de expressões em uma única notação de configuração, chamada de "configuração ativa". Em outras palavras, agora foi fornecida a possibilidade de utilizar os programas como elementos do multiconjunto, o que altera a condição para o término de um programa em Gamma, uma vez que além de não existir nenhum subconjunto de elementos que satisfaçam a condição de reação, é preciso também que não haja nenhuma "configuração ativa"no multiconjunto.

Outros trabalhos envolvendo extensões de ordem superior foram propostos como em [39–41].

#### 2.1.5.2 Aplicações e Implementações

Como resultado do estudo e pesquisa envolvendo o paradigma Gamma, algumas implementações e aplicações foram propostas. Nesta seção abordaremos os principais trabalhos neste sentido.

**– Aplicações**

Na área de processamento de imagens, podemos destacar algumas aplicações, onde Gamma é utilizada. A primeira delas [27] consiste numa aplicação para o reconhecimento da topografia tridimensional da rede vascular cerebral contida em duas radiografias. Uma versão da mesma aplicação havia sido implementada em outra linguagem antes, entretanto, o autor afirma que ela estava ficando enorme e difícil de gerenciar. Com o uso de Gamma, foi possível obter um melhor entendimento dos pontos chave da aplicação, além de reduzir o número de erros. Ainda na área de processamento de imagens, Gamma foi utilizada na geração de fractais para modelar o aumento de objetos biológicos [32]. Em [33] foi apresentada uma aplicação escrita em Gamma para realizar a detecção de bordas de objetos em imagens representadas em escala de cinza. Considerando os exemplos expostos, Gamma parece ser adequada para representar esta classe de algoritmos, talvez pela razão básica de que muitos dos tratamentos no campo de processamento de imagens são naturalmente

expressos como uma coleção de aplicações locais de regras específicas.

BANÂTRE e LE MÉTAYER [33] exibem uma série de programas em Gamma para vários domínios de aplicação, como por exemplo, problemas de processamento de *strings*, problemas de grafos, problemas geométricos, problemas de ordenação, e problemas de sincronização de processos. Na categoria de grafos, um dos programas apresentados realiza o cálculo do caminho mais curto (*shortest path*) entre os nós de um grafo dirigido ponderado. Dentre os problemas geométricos, foi apresentada uma aplicação escrita em Gamma que encontra o fecho convexo de um conjunto de pontos em um plano, o qual é definido como o menor polígono convexo contendo todos estes pontos. Já na categoria de sincronização de processos, o problema dos filósofos famintos foi abordado no qual o desafio básico é prover uma correta alocação de recursos, garantindo que não haja problemas como deadlocks ou starvation. Neste exemplo, o uso de Gamma ocorreu de uma maneira um pouco diferente, pois o interesse não estava propriamente no resultado final do processamento, e sim nos possíveis valores do multiconjunto durante a computação.

Por fim, é importante mencionar um problema de Fusão de Dados para acompanhamento de contatos no contexto militar naval. Trata-se da primeira implementação em ambiente paralelo de um problema de Fusão de Dados utilizado pela Marinha do Brasil. Aqui, o algoritmo PPDE (Par de Plots em Dois Estágios) foi implementado utilizando-se Gamma [17].

– **Implementações**

Com relação às implementações de Gamma, [27] divide estas em implementações para ambientes de memória distribuída e memória compartilhada. No primeiro caso, ambiente de memória distribuída, temos duas possíveis abordagens, que diferem na forma de controle, seja este centralizado ou distribuído.

No **controle centralizado** (memória distribuída), os elementos do multiconjunto estão distribuídos na memória local de cada processador e existe um controlador central, conectado a todos os processadores e responsável pela gerência de troca de mensagens. Como exemplo de implementações deste caso podemos citar a *Conection Machine* [42], o *Maspar* [43], entre outras. A Connection Machine é um computador paralelo com $64K$ processadores. Em [42] os autores propõem um modelo síncrono de execução em Gamma e implementam tal modelo na *Connection Machine*.

Já no caso de ambiente de **controle distribuído** (em memória distribuída), toda a troca de mensagens entre os processadores ocorre de maneira assíncrona. Não existe a figura do controlador central e cada processador conhece somente seus processadores vizinhos. Como exemplo, pode-se citar a solução implementada na

*Intel iPSC2 Machine* [44].

Temos ainda as implementações para ambientes de memória compartilhada. Neste caso, o modelo Gamma é visto como um modelo de memória compartilhada. Uma arquitetura de *software* específica foi proposta em [45].

É importante mencionar as três implementações de Gamma citadas abaixo, baseadas na implementação do Prof. Juarez Muylaert, descrita em [31]:

- *Gamma-Sequencial*;
- *Gamma-MPI*; e
- *Gamma-GPU*.

As duas primeiras implementações foram desenvolvidas por Juarez Muylaert e Simon Gay. A implementação denominada *Gamma-Sequencial* utiliza somente um processador para a execução das reações e não permite a execução em *hardware* paralelo. Neste caso, a sensação de paralelismo é obtida através da eventual alternância entre a execução das reações em um único processador. Ou seja, conforme vimos anteriormente, o formalismo Gamma possui um paralelismo inerente que corresponde à livre execução de reações em um multiconjunto. Assim, nesta implementação, tal característica de paralelismo é simulada através da alternância entre as execuções das reações envolvidas. Esta abordagem sequencial foi expandida pelos mesmos autores para a execução em um *hardware* paralelo onde a comunicação entre os diversos processadores ocorre através da utilização de uma interface de troca de mensagens, mais especificamente a *Message Passing Interface* (MPI) [46]. Nesta implementação, chamada de *Gamma-MPI*, as reações são executadas em processadores distintos de um ambiente de execução paralela, o que envolve o envio do multiconjunto para diversos processadores. A gerência do envio do multiconjunto e da execução das reações nos diversos processadores é realizada através de um conjunto de “células" fornecidas pela implementação. Vale mencionar que *Gamma-MPI* foi estendido por Gabriel Antoine Louis Paillard em 1999 [31, 47] com enfoque na criação e utilização de tipos de dados o que caracteriza esta extensão como uma implementação de Gamma Estruturada. Já a terceira implementação, chamada de *Gamma-GPU* foi implementada por Rubens Pailo em 2015 [6, 30] e consiste na extensão de *Gamma-MPI* para a execução em um *hardware* paralelo com suporte à GPU.

**– Outros trabalhos relevantes**

Finalmente, serão listados abaixo alguns trabalhos importantes que não foram elencados nos tópicos acima por estarem classificados entre extensões do Gamma e

implementações.

*CHAM*, um acrônimo para *CHemical Abstract Machine* foi proposta em 1992 por BERRY e BOUDOL [48]. Este trabalho foi especificamente importante pelos conceitos dos mecanismos de *membrane* e *airlock* trazidos. *Membranes* são utilizadas para encapsular soluções e forçar a ocorrência de reações localmente, ao passo que o mecanismo de *airlock* é utilizado para especificar as comunicações entre estas soluções encapsuladas e o ambiente.

HOCL (*High-Order Chemical Language*) foi proposto em 2009 por WANG e PRIOL [49]. Trata-se de uma linguagem baseada em uma extensão de alto nível do Gamma. Tem por objetivo fornecer uma maneira de definir regras para execução de reações e organização de elementos pelo ponto de vista HOCL de programação. Juntamente com o guia HOCL de programação, foi fornecido um compilador HOCL desenvolvido em JAVA.

Em [50], tendo em vista uma arquitetura orientada a serviços (SOA), os autores propõem a utilização de métodos baseados na observação da natureza (onde citam o pioneirismo de Gamma porém utilizam o HOCL - High Order Chemical Language), para proporcionar uma abordagem descentralizada para o mecanismo de execução de serviços web. Assim, eles fornecem um modelo de execução de alto nível que permite a execução de serviços compostos de maneira descentralizada. Os autores apresentam uma prova de conceito, através da implantação de um protótipo de *software* que implementa os conceitos sugeridos mostrando a viabilidade da solução.

## 2.2 Dataflow

Conforme foi abordado no capítulo introdutório deste trabalho (Capítulo 1), utilizamos dois modelos computacionais paralelos como objeto principal de estudo. Assim, após termos abordado aspectos relevantes sobre o paradigma Gamma, nas seções inicias deste Capítulo, a presente sessão destina-se a apresentar as principais características do segundo modelo computacional paralelo estudado: o modelo Dataflow.

### 2.2.1 Modelo Dataflow

A arquitetura de execução sequencial de instruções vem guiando o projeto de computadores desde o início da história da computação. Esta execução sequencial é exatamente o que descreve o modelo de von Neumann. Neste modelo, existe um *Program Counter* (PC) que aponta para a próxima instrução a ser executada. Desta forma, instruções vão sendo executadas, em sequência, e o PC vai sendo incrementado sempre para a próxima instrução a ser executada, num esquema que chamamos de fluxo de controle. Tal modelo foi incrementado a partir de técnicas como *pipeli-*

*ning*, predição de desvios, renomeamento de registradores e escalonamento dinâmico, numa tentativa de possibilitar o potencial paralelismo pela utilização deste modelo intrinsecamente sequencial [7].

Uma alternativa para o aumento de desempenho computacional foi a introdução de processadores *multicore*, onde tarefas complexas poderiam ser divididas e distribuídas para serem executadas em paralelo nos diversos elementos de processamento disponíveis. Entretanto, a abordagem multicore revela algumas questões importantes, como por exemplo, o custo computacional que a comunicação entre os processos demanda, necessidade de conhecimento de características específicas de *hardware* por parte do programador, características de problemas dependentes dos dados, o que dificulta o desenvolvimento de uma solução paralela. Dessa forma, em 1970 [51] foi apresentado o dataflow como modelo computacional alternativo.

O modelo dataflow, ao contrário dos modelos sequenciais guiados por um fluxo de controle, respeita o princípio de fluxo de dados para execução de um programa. Uma instrução necessita de operandos de entrada para produzir um resultado, que será utilizado por outras instruções posteriormente, dessa forma, a ordem de execução é dada pela disponibilidade dos operandos necessários para o processamento das instruções. Em outras palavras, no modelo dataflow, as instruções podem executar à medida em que seus operandos de entrada estejam disponíveis.

Modelos computacionais baseados em fluxo de dados tornaram-se atraentes pois consistem em uma alternativa aos modelos sequenciais e pela compatibilidade com modernos conceitos de programação, além de oferecer uma solução eficiente para exploração de concorrência em alta escala [52]. Também podemos citar as vantagens do modelo dataflow consistir em um modelo descentralizado, que elimina a necessidade de uma estrutura centralizada para controlar a execução das instruções (como por exemplo um contador de programa), além do dataflow explicitar o paralelismo entre os diferentes caminhos de fluxo de dados [53].

No modelo de fluxo de dados, programas podem ser representados por um grafo direcionado. Desta forma, seus vértices representam as instruções a serem executadas e as arestas as dependência de dados. Desta forma, um nó (ou vértice) possui arestas de entrada, que representam os operandos necessários à execução da instrução e nós de saída, representando o destino dos dados produzidos como resultado da operação expressa pela instrução. Um nó é ativado para execução assim que todos os operandos de entrada tenham sido recebidos e, assim, estes dados são consumidos pela execução da instrução produzindo novos dados que serão disponibilizados em uma de suas arestas de saída. Assim, uma aresta direcionada $A \rightarrow B$ significa que a instrução $B$ necessita de um operando produzido pela instrução $A$.

A Figura 2.2 representa um exemplo de grafo dataflow e seu código equivalente em linguagem de alto nível. Repare que existem nós de ativação especiais respon-

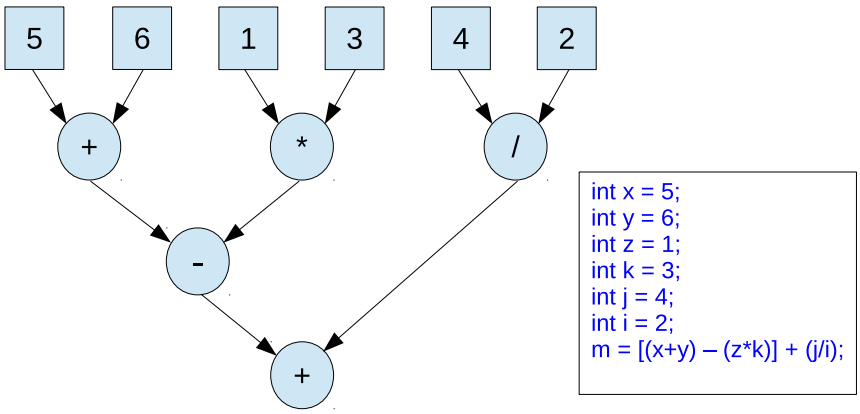


Figura 2.2: Exemplo de um grafo dataflow e seu respectivo código em linguagem de programação baseada no paradigma imperativo.

sáveis por colocar dados iniciais $(5, 6, 1, 3, 4, 2)$ em vértices específicos, disparando a execução do programa. Observe ainda que a representação de grafo permite visualizar que algumas instruções podem ser ativadas simultaneamente, executando em paralelo, como ocorre com as instruções de soma e multiplicação no exemplo da figura.

No que diz respeito aos modelos de execução dataflow, podemos citar o modelo baseado em *tokens* e o modelo baseado em estruturas. No primeiro modelo, os dados trafegam de um vértice do grafo dataflow a outro vértice, através de arestas, encapsulados em pacotes de dados chamados *tokens*. Já no modelo baseado em estruturas, cada vértice do grafo pode implementar uma ou mais estruturas de dados em suas arestas [54].

### 2.2.2 Arquiteturas Dataflow

Apesar da teoria do modelo dataflow ser simples e eficiente, a construção de uma máquina dataflow esbarra em alguns desafios computacionais. Por exemplo, as arestas deveriam ser representadas por filas com capacidades ilimitadas (pois dados podem ser recebidos e enviados em descompasso com o consumo dos mesmos pelos respectivos vértices), o que é impossível de ser implementado em uma memória física real. Além disso, qualquer quantidade de instruções deveria poder ser executada em paralelo, o que torna-se inviável pela quantidade finita de processadores. Dessa forma, é perceptível que existem limitações para a construção de um *hardware* que atenda exatamente o que foi proposto no modelo teórico. Dessa forma, algumas arquiteturas foram propostas [55] como soluções viáveis para implementações do modelo dataflow.

#### 2.2.2.1 Tipos de Arquiteturas

Em 1974, DENNIS e MISUNAS [56] propuseram a chamada **Arquitetura Estática**. Neste tipo de arquitetura, cada aresta de entrada só carrega um *token* de dado. Desta forma, a ativação de um determinado nó ocorre quando um *token* estiver disponível em cada aresta de entrada. A implementação se dá através da inserção de arestas de reconhecimento que possuem sentido oposto a cada aresta do grafo dataflow original. Assim, antes de iniciar a execução, cada nó deve receber todos os *tokens* de reconhecimento correspondentes. A arquitetura estática é simples e rápida na detecção de atividade de um nó. Entretanto, possui o problema de aumento de tráfego pelo crescimento de *tokens* de reconhecimento além de limitar a execução de *loops*, uma vez que todas as operações da iteração anterior devem ser finalizadas para então prosseguir para as operações da iteração atual (não permite paralelizar operações de iterações distintas).

Já em 1983, a **Arquitetura Dinâmica** foi proposta por ARVIND e CULLER [57]. Esta arquitetura incrementa o paralelismo do modelo dataflow pelo fato de permitir que a execução de cada iteração de um *loop* possa ser executada fora de ordem. Ou seja, operações de *loops* distintos podem ser executadas sem que seja necessário aguardar o término da execução de todas as operações de uma iteração. Assim, por ocasião da implementação, cada *token* recebe um rótulo específico identificando a qual iteração este *token* pertence. Um nó é ativado quando todos os seus *tokens* de entrada possuem o mesmo rótulo (tag) da iteração em que o vértice se encontra. Ao contrário do modelo estático, cada aresta pode conter uma grande quantidade de *tokens*. A principal desvantagem do modelo é o *overhead* extra obtido pela associação de rótulos de iteração ao invés de uma *flag* indicando se o nó está ativo, o que acontece no modelo estático.

Em meados dos anos 90, as pesquisas envolvendo o modelo dataflow foram retomadas. Dentre tais estudos, verificou-se que existia possibilidade de intersecção entre os modelos dataflow e von Neumann, surgindo o que denominou-se **Dataflow Híbrido** [58]. Assim, estudos exploraram o desempenho de diversos níveis de granularidade de aplicações em máquinas dataflow, desde granularidade fina (dataflow tradicional), até granularidade grossa (tornando o modelo próximo ao sequencial, devido à execução em série de grandes quantidades de instruções por um vértice) [59].

### 2.2.3 Considerações sobre o modelo Dataflow

Antes de finalizarmos os conceitos referentes ao modelo dataflow, faz-se necessário analisar com maiores detalhes algumas importantes estruturas e detalhes do modelo em questão.

#### 2.2.3.1 Desvios

Desvios condicionais são considerados um problema no modelo dataflow, uma vez que existe uma dificuldade e alto custo envolvido na descrição de desvios de controle, ou seja, como não há registradores nem contador de programa no modelo dataflow, os desvios de controle tem de ser convertidos em desvio de dados. Tal desvio de dados pode ser implementado de duas formas no modelo dataflow: através de **instruções seletoras** ou **instruções *steer*** [60].

Instruções seletoras possuem três operandos de entrada, dentre os quais dois são dados provenientes de caminhos distintos do grafo e um corresponde a um dado Booleano. Desta forma, o operando Booleano seleciona qual dos dois outros operandos será propagado para o grafo através da aresta de saída. Tal estratégia permite com que uma maior quantidade de instruções possa ser executada em paralelo, entretanto, isso leva a um desperdício, uma vez que somente um dos dados de entrada será propagado para o grafo. A Figura 2.3 representa um grafo dataflow com um exemplo de um desvio condicional implementado através de uma instrução seletora (representada por um vértice expresso por um triângulo invertido). Veja que as instruções de subtração e multiplicação podem ser executadas em paralelo. Entretanto, a instrução de multiplicação foi executada de maneira desnecessária, pois seu resultado não será utilizado, dados os valores dos operandos do exemplo.

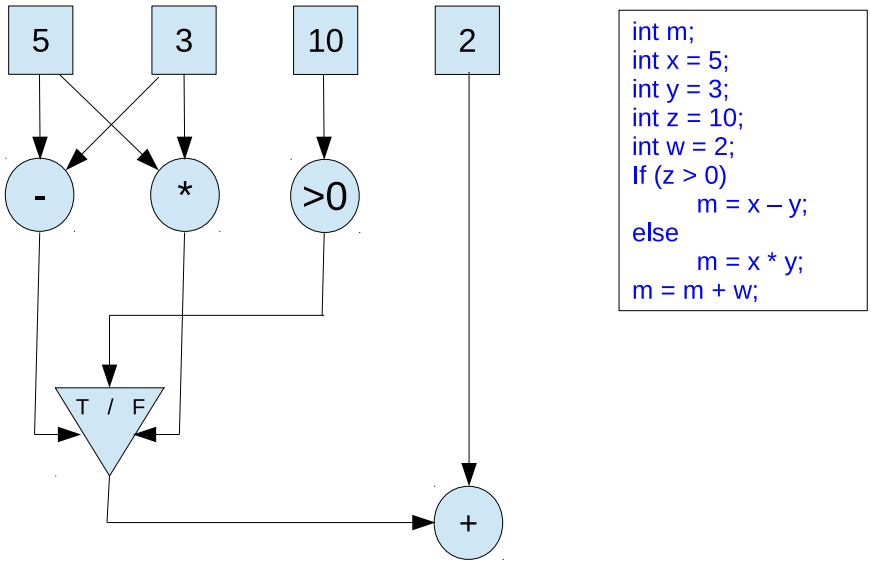


Figura 2.3: Exemplo de utilização de instrução seletora em um grafo dataflow.

Uma outra maneira de implementar desvios condicionais em um grafo dataflow é através de instruções *Steer*. Nesse tipo de instrução, existem dois operandos de entrada, onde um deles é um operando Booleano, responsável pela seleção. O outro operando representa um dado que será repassado para somente uma das duas saídas existentes na instrução (escolhida mediante o valor Booleano recebido como

entrada). Este tipo de abordagem pode ser utilizada na implementação tanto de desvios cíclicos quanto acíclicos, ao contrário de instruções seletoras, que só podem ser utilizadas na implementação de desvios acíclicos [60]. A Figura 2.4 representa o mesmo exemplo da Figura 2.3, entretanto com a utilização de *steers* na implementação dos desvios. Repare que para este exemplo, não é realizada a execução desnecessária da operação de multiplicação.

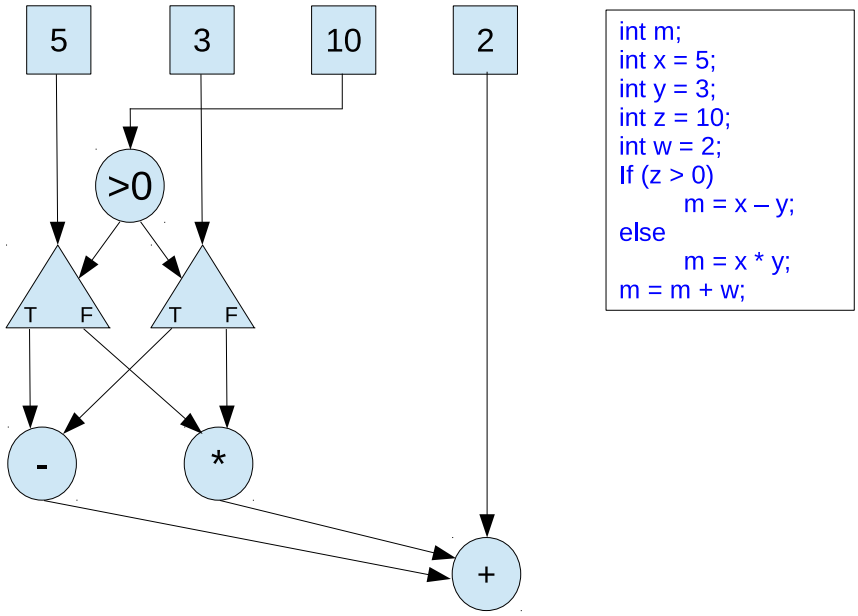


Figura 2.4: Exemplo de utilização de instruções Steer em um grafo dataflow.

#### 2.2.3.2 Laços

Outro assunto que merece destaque é a implementação de laços em dataflow. Laços são boas fontes de se explorar paralelismo, seja entre instruções de uma mesma iteração (o que se traduz na forma tradicional de explorar o paralelismo em dataflow) ou entre instruções que pertencem a iterações distintas. Entretanto, para aproveitar o paralelismo entre instruções de diferentes iterações se faz necessário distinguir os operandos, para que não exista a possibilidade de correspondência entre dados de iterações distintas. Existem duas soluções para evitar este tipo de ocorrência:

- Dataflow Estático: não permite a execução de instruções de iterações seguintes sem que as operações da iteração atual estejam totalmente encerradas. Isso diminui a exploração de paralelismo mas facilita o controle;
- Dataflow Dinâmico: permite a execução de instruções de outras iterações, mediante a utilização de um mecanismo que rotula os operandos. Esta estratégia aumenta a possibilidade de exploração do paralelismo na implementação de laços.

Desta forma, no dataflow dinâmico, existe a necessidade de rotular cada operando com uma *tag* que identifica a qual iteração este pertence. Assim, uma instrução poderá executar quando possuir todos os seus operandos de entrada e se estes operandos possuírem o mesmo rótulo de iteração (mesma tag). A Figura 2.5 mostra a implementação de um laço em dataflow e seu respectivo código em linguagem de alto nível. Repare que os rótulos dos operandos são incrementados a cada iteração através dos *Inctags* (vértices representados por losangos), o que diferencia os operandos de cada iteração. Repare ainda a necessidade de inserção de nós do tipo *Steer* criando desvios condicionais permitindo a inserção das diversas iterações do loop.

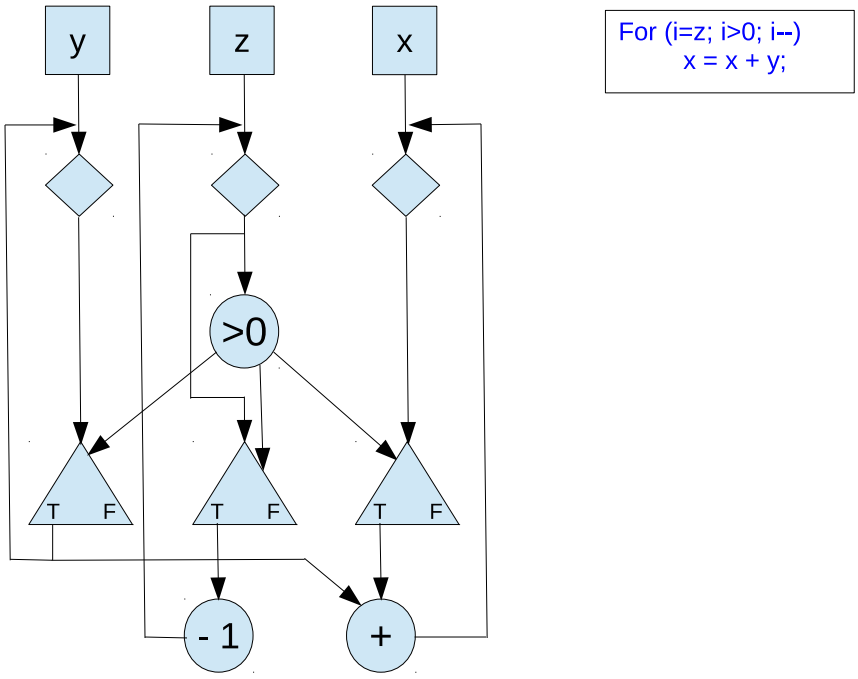


Figura 2.5: Exemplo de implementação de um laço em um grafo dataflow [61].

#### 2.2.3.3 Outras considerações

Apesar das vantagens advindas do modelo dataflow, alguns pontos importantes merecem ser mencionados, como por exemplo a incompatibilidade com linguagens imperativas. Tal incompatibilidade ocorre devido aos acessos à memória. Em linguagens imperativas, a memória serve como um elemento de estado global da máquina, exatamente como o banco de registradores. O banco de registradores é eliminado nas arquiteturas dataflow pois todos os operandos de entrada das instruções são enviados diretamente por outras instruções [61]. Assim, é preciso ordenar os acessos à memória de acordo com a ordem do programa para que não haja perda semântica. Outra alternativa seria utilizar a memória para armazenar informação do contexto

local das instruções.

A alocação de tarefas é outro aspecto que merece destaque, por impor sérios desafios ao modelo dataflow. Existem diferentes estratégias para escalonamento aos elementos processadores [60], que influenciam diretamente no desempenho da arquitetura. Aqui, o desafio é prover uma alocação eficiente de tarefas, evitando contenções e reduzindo ao máximo o custo computacional envolvido na comunicação entre estes elementos processadores.

### 2.2.4 Estado da Arte

O projeto *SUCURI* [15] consiste em uma biblioteca escrita em linguagem Python que permite programação e execução dataflow em alto nível, cuja estrutura é dividida em: *Grafo*, *Escalonador* e *Worker*. A programação na *SUCURI* consiste inicialmente na identificação de funções com boa capacidade de paralelização. Posteriormente, um grafo dataflow será instanciado de forma que cada vértice deste grafo estará associado a uma função previamente identificada. As arestas do referido grafo dataflow correspondem às dependências de dados entre tais funções. Os resultados experimentais da *SUCURI* são promissores e sugerem a mesma como uma boa opção para paralelização.

Analogamente, o Intel® *Threading Building Blocks* (TBB) [9], oferece uma abordagem completa para expressar paralelismo em C++, projetado para prover uma camada abstrata para auxiliar programadores no desenvolvimento de código com múltiplas linhas de execução. Consiste em uma biblioteca C++ com recursos e características bastante similares à *SUCURI*, para programação paralela em processadores multicores. Assim, o *TBB* permite a identificação de tarefas de maneira a representarem nós de um grafo dataflow, onde as arestas correspondem às dependências de dados entre estas tarefas. A possibilidade de especificação de tais tarefas leva a uma programação de mais alto nível do que escrever diretamente código para gerenciar ***threads***. Uma outra funcionalidade do *TBB* é o uso de *templates* para instanciar mecanismos como pipelines.

O *OpenMP* [10] consiste em uma *API* para programação paralela em sistemas com memória compartilhada, fornecendo um conjunto de diretivas, bibliotecas e variáveis de ambientes fornecidas tendo em vista a atividade de descrição de tarefas paralelas. O objetivo é fornecer ao desenvolvedor uma interface simples e flexível para paralelizar aplicações. A API é bastante utilizada em modelos *fork/join* onde uma *thread* mestra coordena a execução de *threads* escravas. O *OpenMP* também disponibiliza anotações para serem utilizadas na descrição de laços paralelos, onde ocorre divisão automática de tarefas seguindo diretivas e escalonamento e granularidade definidas pelo desenvolvedor.

O *TensorFlow* [11, 12] é uma plataforma completa de código aberto para *Machine Learning* (ML), que opera em larga escala e em ambientes heterogêneos. Seu modelo computacional é baseado em grafos dataflow, onde os vértices podem ser mapeados para diferentes máquinas em um *cluster* e, dentro de cada máquina, para *CPUs*, *GPUs* e outros dispositivos. O TensorFlow oferece suporte a uma variedade de aplicativos, mas visa principalmente o treinamento e a inferência com redes neurais profundas.

O *Fastflow* [13] é um *framework* de programação explicitamente desenvolvido para dar suporte à linguagens de alto nível para aplicações de *streaming*. Foi projetado para multicores de memória compartilhada, tendo sido implementado como uma pilha de bibliotecas de *templates* C++. Os resultados experimentais do *Fastflow* demonstraram sua eficiência comparados a outros *frameworks* clássicos como *Cilk*, *OpenMP* e Intel® TBB.

## 2.3 Discussões

Gamma é um paradigma computacional não determinístico proposto em 1986, onde o arcabouço computacional traz uma metáfora às reações químicas. Assim sendo, a base de dados existente, o multiconjunto, assemelha-se à uma solução química. As operações computacionais (reações) ocorrem livremente sobre o multiconjunto uma vez que as condições necessárias (condições de reação) sejam satisfeitas. O não determinismo do paradigma está relacionado à escolha dos elementos do multiconjunto que irão ser fornecidos às reações para, possivelmente, reagir.

Por outro lado, o modelo dataflow pode ser expresso através de um grafo, onde as operações computacionais (expressas pelos vértices deste grafo), ocorrem mediante disponibilidade dos dados necessários, expressos pelas arestas de entrada de um determinado vértice. Dessa maneira, caso os dados estejam prontos, a operação ocorre, através de um escalonamento que respeita o fluxo de dados, ao invés do fluxo de controle, conforme arquiteturas baseadas no modelo sequencial de von Neumann.

Ambos modelos citados possuem um paralelismo implícito e natural, seja pela capacidade de Gamma distribuir a execução das reações de forma a cobrir o multiconjunto de maneira simples e naturalmente paralela, seja potencial de exploração de execução paralela de instruções que respeitam o fluxo de dados, no modelo dataflow.

# Capítulo 3

# Equivalência entre Gamma e Dataflow

## 3.1 Considerações sobre Similaridade

Os modelos computacionais Gamma e Dataflow apresentam uma surpreendente similaridade. Ambos modelos apresentam formas naturais de expressar o paralelismo, de maneira que detalhes de implementação tornam-se transparentes para o programador. Desta forma, em [23] apresentamos pela primeira vez a similaridade entre Gamma e Dataflow. Posteriormente, em [24] apresentamos a prova formal de equivalência entre os modelos computacionais em questão.

A ideia principal por trás de tal similaridade é mostrar que um vértice de um grafo dataflow e suas arestas de entrada e saída, apresentam equivalência com uma reação escrita em código Gamma e os respectivos dados manipulados por ela. Desta forma, é possível converter um grafo dataflow em seu respectivo código Gamma e vice e versa. Com isso, contribui-se para a versatilidade e extensão de benefícios para ambos modelos, conforme apresentado no Capítulo 1. Assim, nas próximas seções iremos abordar detalhes acerca desta similaridade e apresentaremos a prova formal de equivalência entre os modelos computacionais Gamma e Dataflow.

### 3.1.1 Dataflow para Gamma

Tomemos como base o código abaixo escrito em linguagem de alto nível baseado no paradigma imperativo:

```
int x = 1;
int y = 5;
int k = 3;
int j = 2;
int m;
m = (x + y) - (k * j);
```

Tal programa pode ser representado pelo grafo dataflow apresentado na Figura 3.1 onde todos os vértices e arestas possuem um rótulo com o objetivo de facilitar o entendimento do processo de conversão que será descrito posteriormente.

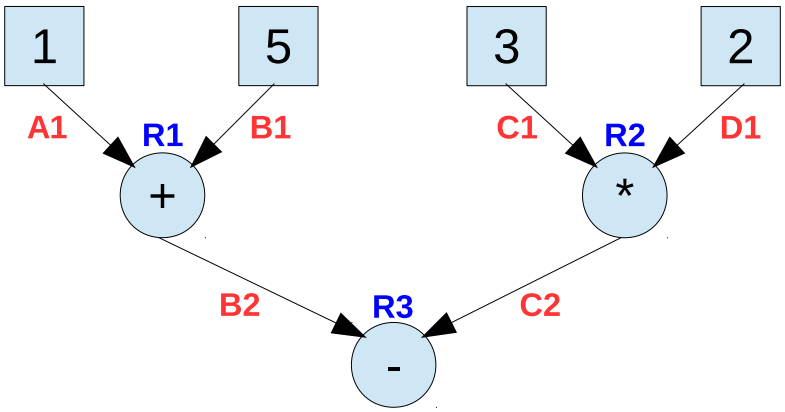


Figura 3.1: Exemplo 1 - Grafo dataflow.

Conforme apresentado na Seção 2.2, um programa pode ser representado por um grafo direcionado, onde vértices e arestas correspondem à operações e dependências de dados, respectivamente. Um vértice possui operandos de entrada, o que indica que a instrução necessita de operandos para iniciar sua execução e operandos de saída, que são os resultados produzidos por este vértice. Desta maneira, um vértice inicia sua operação quando todos seus operandos de entrada estiverem disponíveis.

Na Figura 3.1, a operação de subtração, representada pelo vértice $R3$, só pode ser executada após os dados $B2$ e $C2$ terem sido produzidos pelas operações (vértices) $R1$ e $R2$, respectivamente.

Para converter o grafo dataflow apresentado na Figura 3.1 em um código Gamma, todos os vértices serão convertidos em reações e as arestas em elementos do multiconjunto. O multiconjunto inicial será formado pelas arestas iniciais (arestas de saída dos vértices representados por quadrados). Tendo em vista o processo de conversão, necessitaremos atribuir rótulos, tanto para os vértices, quanto para as arestas do grafo dataflow. Os vértices darão origem às reações, desta forma tais rótulos serão utilizados na identificação das reações equivalentes. Já as arestas irão compor o multiconjunto, seja no momento de criação deste, seja em tempo de execução, onde elementos vão sendo inseridos, modificados e excluídos do multiconjunto. Assim, visando esta identificação unívoca dos elementos em Gamma (que são relacionados às arestas do grafo dataflow), a atribuição dos rótulos às arestas se faz necessária. Como precisaremos armazenar informação sobre valores e rótulos (informação rotulada por *tags*), os elementos de nosso multiconjunto serão representados por n-tuplas de dois elementos. Assim, a aresta $A1$ corresponde ao elemento $[1, A1]$, onde a primeira informação refere-se ao valor da aresta e a segunda o rótulo da aresta (*label* ou *tag*).

Desta forma, temos o seguinte multiconjunto inicial:

```
M = {[1, A1], [5, B1], [3, C1], [2, D1]}
```

O processo de transformação visa converter todo vértice em uma reação que irá manipular e produzir alguns dados. Por exemplo, o vértice $R1$ consome os elementos $[1, A1]$ e $[5, B1]$ produzindo o dado $[1 + 5, B2]$, conforme apresentado abaixo:

```
R1 = replace [id1, 'A1'], [id2, 'B1']
by [id1 + id2, 'B2']
```

Onde $[id1, `A1']$ significa uma tupla rotulada com a tag $A1$ e que possui algum valor no primeiro campo ($id1$). Repare que não existe condição de reação definida para a reação $R1$, desta forma, quando forem selecionados elementos com rótulos $A1$ e $B1$, esta reação irá executar. Assim, podemos produzir o seguinte código Gamma equivalente ao grafo dataflow apresentado na Figura 3.1:

```
R1 = replace [id1, 'A1'], [id2, 'B1']
by [id1 + id2, 'B2']

R2 = replace [id1, 'C1'], [id2, 'D1']
by [id1 * id2, 'C2']

R3 = replace [id1, 'B2'], [id2, 'C2']
by [id1 - id2, 'm']
```

Observe que em algumas implementações utilizadas, como [62], o programador pode introduzir operadores sequenciais e paralelos, relacionados à ordem de execução das reações, representados por " ; " e " | ", respectivamente. Em nossos exemplos, estamos considerando somente o operador paralelo, o que significa que todas as reações podem executar em paralelo, ou seja, $R1 \mid R2 \mid R3 \mid ... \mid Rn$.

Agora, considere um segundo exemplo, representado pelo seguinte código em alto nível:

```
For (i=z; i>0; i--)
    x = x + y;
```

O grafo dataflow correspondente ao código acima é apresentado na Figura 3.2.

Este exemplo ilustra o conceito de loops e estruturas de decisão em um grafo dataflow. Conforme vimos na Seção 2.2, para estruturas de decisão, o referido grafo dataflow utiliza o operador ***Steer*** (representado por triângulos na Figura 3.2). Este operador recebe dois operandos de entrada: o primeiro corresponde a um valor, enquanto que o segundo corresponde a um sinal de controle Booleano. Se o sinal de controle for *TRUE*, a saída marcada como *TRUE* recebe o valor do dado de

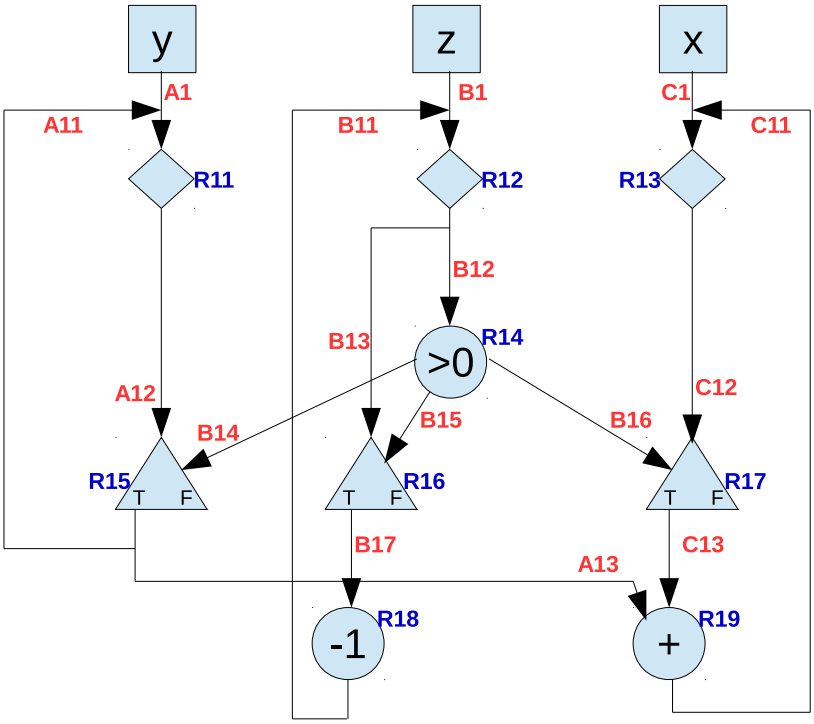


Figura 3.2: Exemplo 2 - Grafo dataflow.

entrada, caso contrário, a saída marcada como *FALSE* transmite o valor do dado de entrada. Como podemos observar na Figura 3.2, o grafo apresenta três operadores do tipo *Steer*, rotulados por $R15$, $R16$ e $R17$. Todos os sinais Booleanos de controle recebidos pelos operadores *Steer* são produzidos mediante a comparação com zero, expressa pelo vértive $R14$.

Considere o *Steer* rotulado por $R16$, na Figura 3.2. Tal operador possui dois operandos de entrada ($B13$ e $B15$) e produz somente a saída marcada como *TRUE* ($B17$). O motivo da tupla que representa os elementos do multiconjunto ter agora 3 elementos será explicado a seguir. O código Gamma correspondente pode ser representado por:

```
R16 = replace [id1,‘B13’,v], [id2,‘B15’,v]
by [id1,‘B17’,v]
if id2 == 1
by 0
else
```

Caso o sinal de controle Booleano ($B15$) seja *TRUE* ($id2 == 1$), os elementos $B13$ e $B15$ serão substituídos pelo elemento $B17$, que irá conter o valor do dado de $B13$. Caso contrário, estes dois elementos serão excluídos do multiconjunto e nenhum outro elemento será inserido (conforme descrito na cláusula *else*).

Outro tipo especial de vértice é o *Inctag*, representados no grafo dataflow do exemplo da Figura 3.2 pelos losangos rotulados por $R11$, $R12$ e $R13$. Este tipo de

operador é responsável por incrementar o rótulo de iteração de cada dado (operando), identificando dados de diferentes iterações. Portanto, uma operação só poderá ser executada utilizando dados de entrada pertencentes a mesma iteração (dados que possuam mesmo rótulo de iteração). Por este motivo, para este exemplo, os dados serão representados no multiconjunto por uma n-tupla composta por três elementos: dado, rótulo da aresta e rótulo de iteração. Assim, o multiconjunto inicial para o grafo dataflow apresentado na Figura 3.2 é:

```
M = {[y, A1, 0], [z, B1, 0], [x, C1, 0]}
```

Perceba que, para os elementos do multiconjunto inicial, todos os rótulos de iteração são iguais a zero. Estes valores serão incrementados a cada iteração como efeito da execução dos operadores *Inctag*.

Considere agora o exemplo de *Inctag* ainda na Figura 3.2. O vértice $R11$ recebe somente um operando de entrada ($A1$ para a primeira iteração ou $A11$ para as demais iterações) e produz somente uma saída ($A12$). O código Gamma correspondente, para o vértice $R11$ pode ser representado por:

```
R11 = replace [id1,x,v]
by [id1,‘A12’,v+1]
if (x==‘A1’) or (x==‘A11’)
```

Repare que esta reação somente incrementa o rótulo de iteração e modifica (transforma) o rótulo da aresta do referido dado. De acordo com as transformações apresentadas no primeiro exemplo e conforme as observações sobre os operadores *Steer* e *Inctag*, o código Gamma correspondente ao grafo dataflow apresentado na Figura 3.2, pode ser expresso através das nove reações descritas a seguir:

```
R11 = replace [id1,x,v]
by [id1,‘A12’,v+1]
if (x==‘A1’) or (x==‘A11’)

R12 = replace [id1,x,v]
by [id1,‘B12’,v+1], [id1,‘B13’,v+1]
if (x==‘B1’) or (x==‘B11’)

R13 = replace [id1,x,v]
by [id1,‘C12’,v+1]
if (x==‘C1’) or (x==‘C11’)

R14 = replace [id1, ‘B12’, v]
by [1,‘B14’,v], [1,‘B15’,v], [1,‘B16’,v]
If id1 > 0
by [0,‘B14’,v], [0,‘B15’,v], [0,‘B16’,v]
```

```
else

R15 = replace [id1,‘A12’,v], [id2,‘B14’,v]
by [id1,‘A11’,v], [id1,‘A13’,v]
If id2 == 1
by 0
else

R16 = replace [id1,‘B13’,v], [id2,‘B15’,v]
by [id1,‘B17’,v]
If id2 == 1
by 0
else

R17 = replace [id1,‘C12’,v], [id2,‘B16’,v]
by [id1,‘C13’,v]
If id2 == 1
by 0
else

R18 = replace [id1,‘B17’,v]
by [id1 - 1,‘B11’,v]

R19 = replace [id1,‘A13’,v], [id2,‘C13’,v]
by [id1+id2,‘C11’,v]
```

### 3.1.2 Gamma para Dataflow

De maneira similar às transformações que foram apresentadas na seção anterior, para transformarmos um código Gamma em seu equivalente grafo dataflow, a ideia básica consiste em transformar cada reação em um vértice e cada dado manipulado por esta reação em arestas do grafo.

Tomemos mais uma vez o exemplo do código Gamma abaixo, equivalente ao grafo dataflow apresentado na Figura 3.1:

```
R1 = replace [id1, ‘A1’], [id2, ‘B1’]
by [id1 + id2, ‘B2’]

R2 = replace [id1, ‘C1’], [id2, ‘D1’]
by [id1 * id2, ‘C2’]

R3 = replace [id1, ‘B2’], [id2, ‘C2’]
by [id1 - id2, ‘m’]
```

Neste exemplo, a partir da reação $R1$, o vértice $R1$ é criado contendo a operação

de soma descrita pela cláusula “by":

```
R1 = replace [id1, ‘A1’], [id2, ‘B1’]
by [id1 + id2, ‘B2’]
```

No grafo dataflow equivalente, o vértice $R1$ terá dois operandos de entrada $A1$ ([$id1$, ‘$A1$’]) e $B1$ ([$id2$, ‘$B1$’]), e produzirá um operando de saída $B2$ ([$id1 + id2$, ‘$B2$’]). Processo similar pode ser aplicado às reações $R2$ e $R3$. Finalmente, o multiconjunto inicial, composto pelos elementos, $[1, A1]$, $[5, B1]$, $[3, C1]$ e $[2, D1]$, dará origem aos vértices e arestas iniciais do grafo, representados por vértices quadrados. Portanto, podemos reproduzir o mesmo grafo dataflow apresentado na Figura 3.1, a partir das três reações mencionadas.

O mesmo raciocínio pode ser aplicado ao segundo exemplo apresentado na seção 3.1.1, composto por nove reações Gamma. As reações $R11$, $R12$ e $R13$ referem-se à operadores *Inctag*. A reação $R11$ possui um operando de entrada (conforme mencionado pela cláusula “replace"), entretanto, este operando pode ter o rótulo de aresta igual a $A1$ ou $A11$ (de acordo com a condição de reação $if(x ==$ ‘$A1$’) or ($x ==$ ‘$A11$’)). A operação do *Inctag* pode ser identificada pelo incremento do campo de rótulo de iteração (*by* [$id1$, ‘$A12$’, $v + 1$]). Desta maneira, este tipo de vértice (representado por um losango) receberá um operando de entrada ($A1$ ou $A11$) e irá produzir somente um operando de saída $A12$.

Com relação ao operador *Steer*, representado pelas reações $R15$, $R16$, e $R17$, todos consomem dois elementos do multiconjunto e produzem elementos relacionados às condições de teste (expressas pelas cláusulas *by* e $if$). Em outras palavras, neste exemplo, somente as saídas com valor *TRUE* irão produzir elementos. Desta forma, o *Steer* pode ser identificado por sempre comparar dois elementos (valor e sinal de controle Booleano) e por possuir testes condicionais para cláusulas *TRUE* e *FALSE*. Portanto, este tipo de reação pode ser convertida em um vértice representado por um triângulo.

Considerando a reação $R17$, os operadores de entrada serão $C12$ e $B16$ (*replace* [$id1$,‘$C12$’,$v$], [$id2$,‘$B16$’,$v$]) e somente a saída *TRUE* será fornecida (*by* [$id1$,‘$C13$’,$v$]), criando a aresta rotulada por $C13$ caso $id2$ possuir valor *TRUE*. Assim, aplicando o mesmo processo utilizado no exemplo anterior, podemos reproduzir o grafo dataflow apresentado na Figura 3.2.

### 3.1.3 Reduções

A quantidade de reações Gamma, apresentados no primeiro e segundo exemplo da seção 3.1.1 pode ser reduzida. Este fato irá afetar diretamente a granularidade das operações nos modelos Gamma e dataflow. Assim, algumas reduções ou expansões podem ser realizadas.

Considerando o código Gamma composto pelas reações $R1$, $R2$ e $R3$, referentes à conversão da Figura 3.1, tais reações podem ser substituídas por apenas uma reação, da seguinte maneira:

```
Rd1 = replace [id1,‘A1’], [id2,‘B1’], [id3,‘C1’], [id4,‘D1’]
by [(id1+id2)-(id3*id4),‘m’]
```

Repare que, com esse código reduzido, a oportunidade de exploração do paralelismo de reações diminui, uma vez que esta reação somente irá executar quando todos os operandos forem escolhidos na ordem esperada para esta reação. Em outras palavras, a chance das condições de reação serem atendidas pode ser diminuída.

Da mesma forma, o código referente ao segundo exemplo (Figura 3.2) pode ser reduzido. Neste caso, gerenciamos o código para alcançar um total de seis reações, conforme apresentado a seguir:

```
Rd11 = replace [id1,x,v]
by [id1,‘A12’,v+1]
If (x==‘A1’) or (x==‘A11’)

Rd12 = replace [id1,x,v]
by [id1,‘B14’,v+1], [id1,‘B12’,v+1],
[id1,‘B16’,v+1]
If (x==‘B1’) or (x==‘B11’)

Rd13 = replace [id1,x,v]
by [id1,‘C12’,v+1]
If (x==‘C1’) or (x==‘C11’)

Rd14 = replace [id1,‘A12’,v], [id2,‘B14’,v]
by [id1,‘A11’,v], [id1,‘A13’,v]
If id2 > 0
by 0
else

Rd15 = replace [id1,‘B12’,v]
by [id1 - 1,‘B11’,v]
If id1 > 0
by 0
else

Rd16 = replace [id1,‘A13’,v], [id2,‘B16’,v],
[id3,‘C12’,v]
by [id1 + id3,‘C11’,v]
If id2 > 0
by 0
else
```

Para chegar a esta quantidade de seis reações, em um código inicialmente composto por nove, a técnica utilizada foi fazer com que alguns nós (do grafo dataflow equivalente) incorporem funcionalidades que seriam de outros. Para fins de melhor visualização, apresentaremos as reduções utilizadas em duas etapas. Inicialmente a Figura 3.3 apresenta o mesmo grafo apresentado na Figura 3.2, entretanto com o equivalente código em linguagem imperativa e em Gamma.

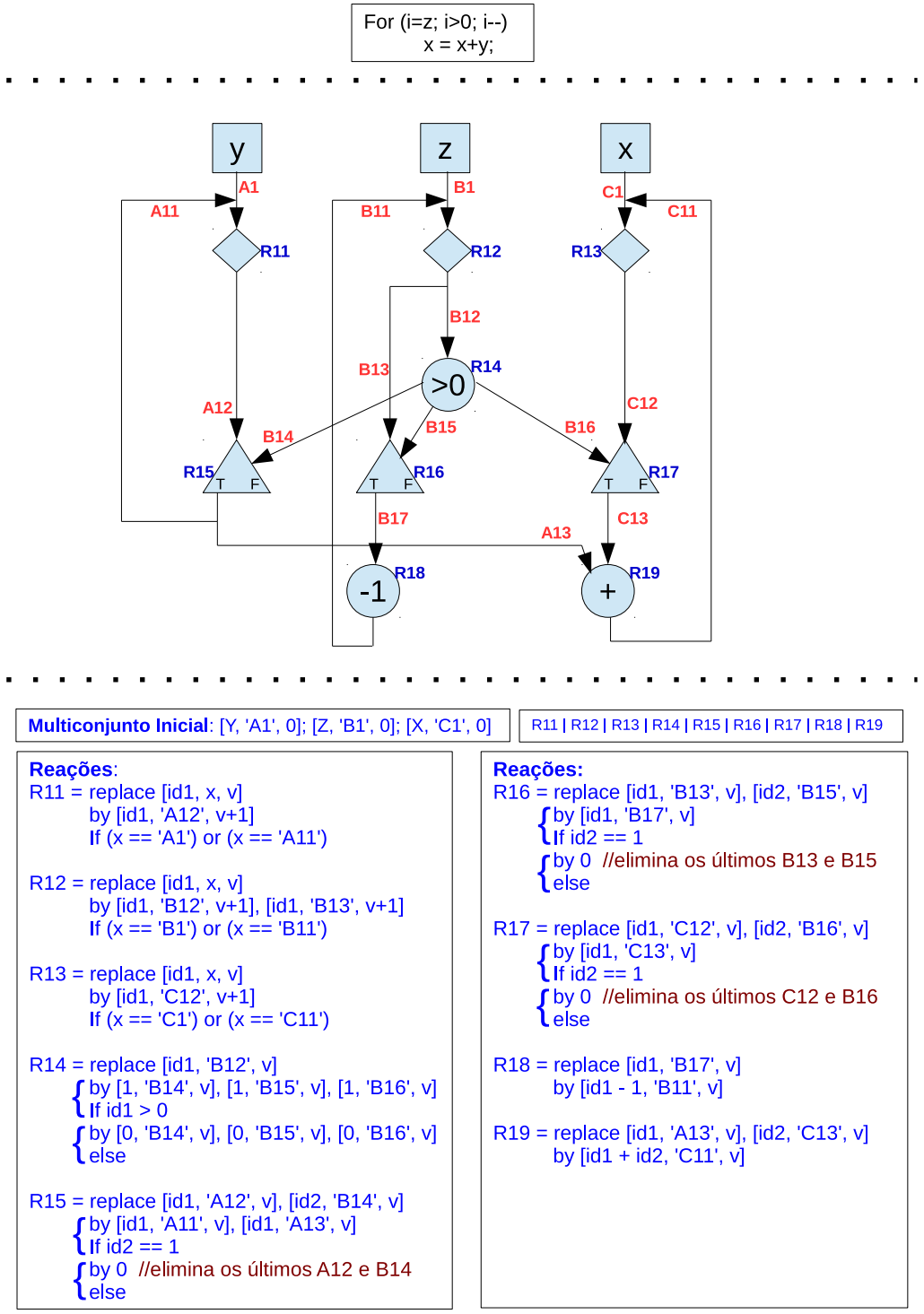


Figura 3.3: Grafo dataflow e Código Gamma Equivalente - Redução.

Na primeira etapa, ilustrada pela Figura 3.4, o objetivo foi incorporar as funcionalidades realizadas por $R14$ aos nós $R15$, $R16$ e $R17$. Com isso, retirarmos do grafo $R12$ e renomeamos os nós. As reações que precisaram ser alteradas aparecem na imagem em vermelho.

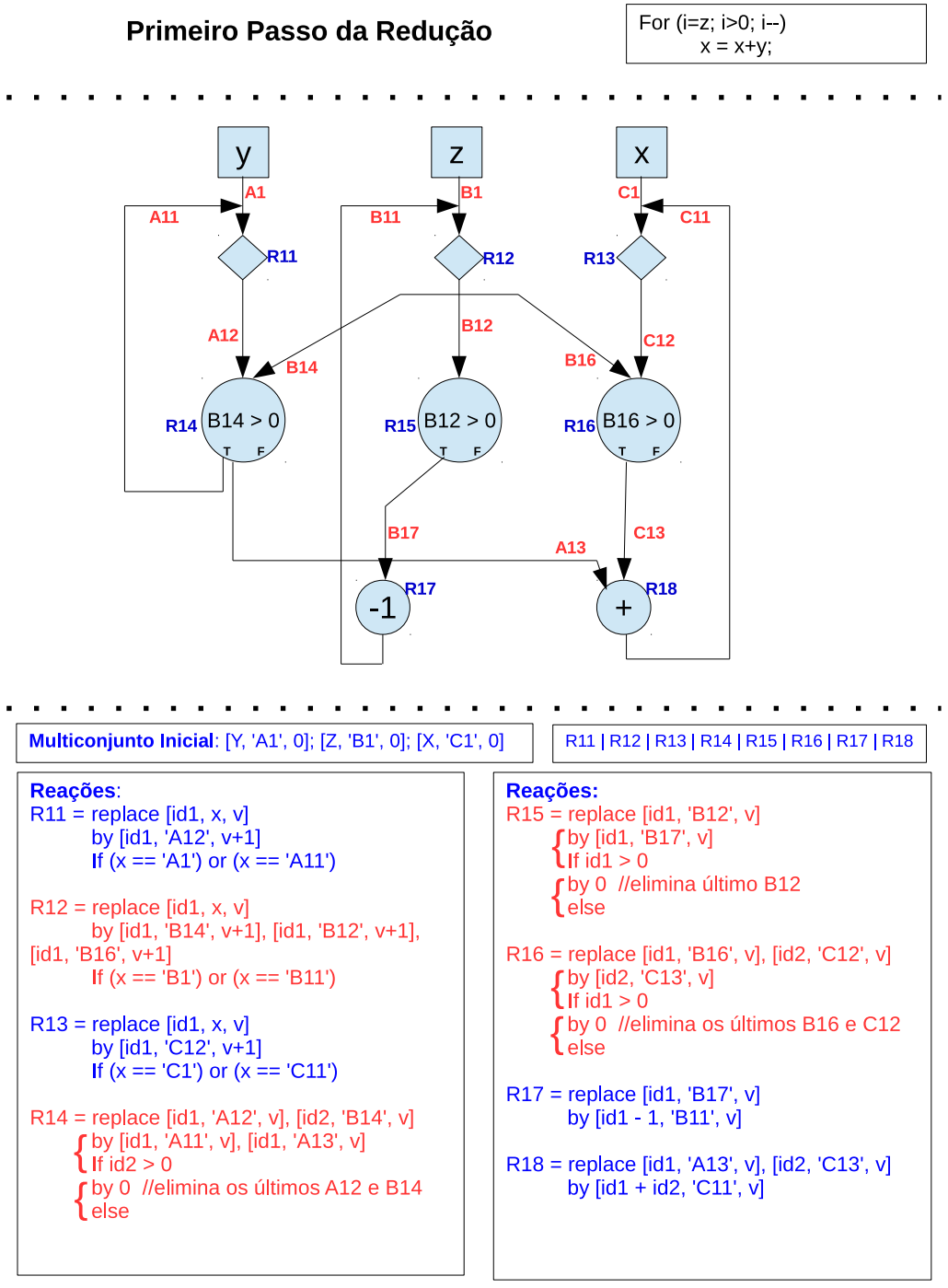


Figura 3.4: Grafo dataflow e Código Gamma Equivalente - Redução - Passo 1.

Já no segundo passo da referida redução (ilustrada pela Figura 3.5), o objetivo foi incorporar a $R15$ a funcionalidade desempenhada por $R17$ (decremento). Da mesma forma, incorporamos em $R16$ a soma antes realizada por $R18$.

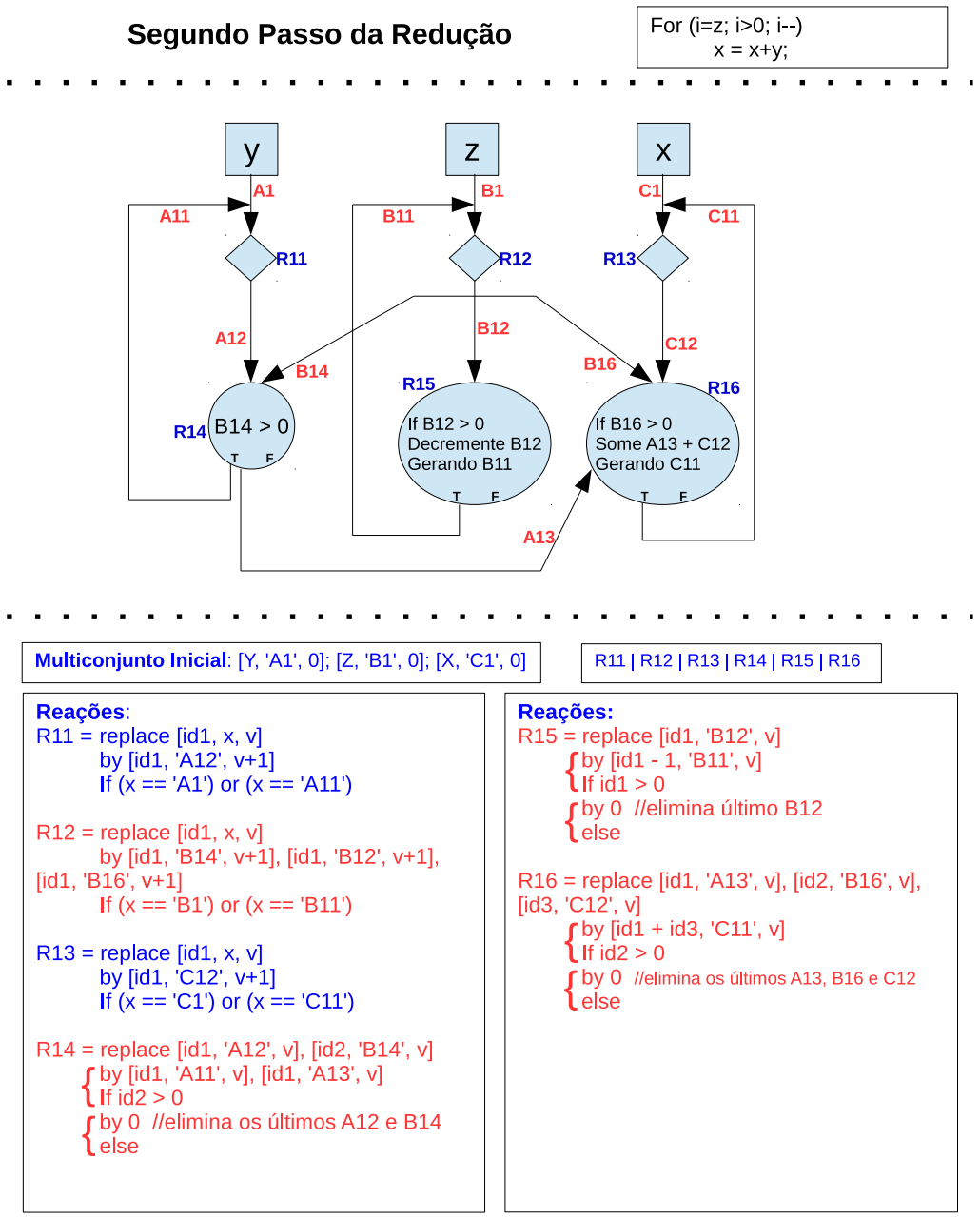


Figura 3.5: Grafo dataflow e Código Gamma Equivalente - Redução - Passo 2.

Observe que, conforme mencionado, tais reduções foram realizadas tendo em vista a análise do grafo dataflow, do ponto de vista de incorporação de funcionalidades entre nós. Entretanto, técnicas de redução podem ser aplicadas ao próprio código Gamma, onde a análise seria do ponto de vista os elementos consumidos e produzidos pelas reações. Assim, a intenção de apresentar estas reduções é mostrar a possibilidade de alteração de granularidade, tanto do ponto de vista do grafo dataflow, quanto das reações Gamma. O assunto merece um estudo a parte, que não será abordado neste trabalho.

## 3.2 Algoritmo de Conversão

Nesta seção serão apresentados os algoritmos utilizados nas transformações de um grafo dataflow para um código Gamma e vice-e-versa, conforme apresentado nas seções anteriores.

Generalizamos a sintaxe de um código Gamma em uma notação de uma gramática livre de contexto, conforme apresentado na Figura 3.6. Basicamente, a sintaxe é composta de duas partes: *replace list*, que descreve o número de elementos para inicializar a reação e *by list*, que especifica os elementos produzidos em *by output* controlados pelas condições descritas na cláusula *by condition*. Utilizando a notação de gramática livre de contexto, códigos Gamma podem ser criados através da leitura de estruturas de dados que mantém a *replace list* e *by list*.

O procedimento para gerar um programa escrito em Gamma a partir de um grafo dataflow é detalhado no Algoritmo 1. Inicialmente, é gerado um rótulo para cada nó do grafo dataflow (linhas 3-6). Conforme discutido na Seção 3.1.1, para permitir a utilização de instruções do tipo *Inctags*, cada elemento do multiconjunto deve ser formado a partir de uma n-tupla com três elementos *[value, label, tag]*. O multiconjunto inicial $M$ é formado pelos nós raízes na linha 9, uma vez que estes não possuem operandos de entrada. Os demais nós, adicionam parâmetros na *replace list* $R_L$ e na *by list* $B_L$ onde cada entrada contém os valores de saída $B_V$ e condições $B_C$. Nós do tipo *Steer* produzem duas entradas na *by list* associadas a cada caminho que os operandos de saída podem ser enviados (portas *TRUE* $t$ ou *FALSE* $f$), controladas pelo operando Booleano $X_1$. Nas linhas 21-22, nós *Inctag* somente incrementam o rótulo de elementos de entrada. Nós de operadores aritméticos e de comparação produzem suas operações em *by list* replicando elementos de saída com rótulo para cada nó de saída no grafo dataflow.

O procedimento para converter um código Gamma em um grafo dataflow é dividido em duas etapas: (1) gerar um grafo dataflow para cada reação e (2) mapear os elementos do multiconjunto ao grafo dataflow produzido na etapa 1.

A Etapa 1 é apresentada no Algoritmo 2. Considerando que cada reação é associada a um grafo dataflow, os nós raiz são obtidos por elementos na *replace list* $R_L$ nas linhas 2-4. Se *by list* $B_L$ não possuir expressão de condição, então nós aritméticos e as arestas, conectando cada elemento de entrada da *replace list* à operadores aritméticos, são criados nas linhas 18-21. Caso contrário, nós do tipo *Steer* são gerados com seus nós de comparação relacionados e suas portas *TRUE* são ligadas aos nós aritméticos nas linhas 13-16. Repare que somente a análise da sintaxe da reação não fornece informação suficiente para produzir nós do tipo *Inctag*. Loops são descritos implicitamente no programa Gamma com um número indeterminado de iterações.

**Algorithm 1:** Conversão de um grafo dataflow em um conjunto de reações Gamma.

**Input:** Grafo Dataflow $D(I, E)$, *values* array armazenando valor para cada nó

**Output:** Programa Gamma $G(R, M)$ correspondente à $D(I, E)$

$label \leftarrow []$; $l \leftarrow 0$;
$R \leftarrow \varnothing$; $M \leftarrow \varnothing$;
**foreach** *instruction* $i \in I$ **do**
    $label[i] \leftarrow l$;
    $l \leftarrow l + 1$;
**end**
**foreach** *instruction* $i \in I$ **do**
    **if** $i$ *is root* **then**
        $M \leftarrow M \cup \{[value(i), label[i], 0]\}$;
    **else**
        $R_L \leftarrow \varnothing$;// `Replace list`
        $B_L \leftarrow \varnothing$;// `By condition list`
        **if** $i$ *is Steer st with input* $s$ *and output ports* $t$ *and* $f$ **then**
            $R_L \leftarrow \{[x_0, label[s], tag], [x_1, label[st], tag]\}$;
            $B_V1 \leftarrow \{[x_0, label[t], tag]\}$;
            $B_C1 \leftarrow \{(x_1 == 1)\}$;
            $B_V2 \leftarrow \{[x_0, label[f], tag]\}$;
            $B_C2 \leftarrow \{(x_1 == 0)\}$;
            $B_L \leftarrow \{(B_V1, B_C1), (B_V2, B_C2))\}$;
        **else if** $i$ *is Inctag it with input* $s$ *and output* $o$ **then**
            $R_L \leftarrow \{[x_0, label[it], tag]\}$;
            $B_L \leftarrow \{[x_0, label[o], tag + 1]\}$;
        **else if** $i$ *is comparison operator op with inputs* $s1$ *and* $s2$ **then**
            $R_L \leftarrow \{[x_0, label[s1], tag], [x_1, label[s2], tag]\}$;
            **foreach** *output* $o$ *from* $i$ **do**
                $B_L \leftarrow \{[1, label[o], tag], (x_0\ op\ x_1)\}$;
                $B_L \leftarrow \{[0, label[o], tag], !(x_0\ op\ x_1)\}$;
            **end**
        **else if** $i$ *is arithmetic operator op with inputs* $s1$ *and* $s2$ **then**
            $R_L \leftarrow \{[x_0, label[s1], tag], [x_1, label[s2], tag]\}$;
            **foreach** *output* $o$ *from* $i$ **do**
                $B_L \leftarrow \{[x_0\ op\ x_1, label[o], tag]\}$;
            **end**
        **end**
        $R \leftarrow \{(R_L, B_L)\}$;
    **end**
**end**

**Syntax**

```
r = replace REPLACE_LIST
[by BY_OUT where BY_COND]   //optional command
…                           //repeat preceding command
```

**Terminals**

```
var = {elements of multiset}
comp_op = {<, >, <=, >=, !=, ==}
logic_op = {&&, ||, !}
arith_op = {+, -, *, /, %}
```

**Free-context grammar**

```
REPLACE_LIST =   (REPLACE_LIST) |
                 REPLACE_LIST, |
                 REPLACE_LIST, REPLACE_LIST, |
                 var
BY_OUT =         (BY_OUT) | BY_OUT, |
                 BY_OUT, BY_OUT |
                 ELEM
ELEM =           (ELEM) | ELEM arith_op ELEM | var
BY_COND =        (BY_COND) |
                 BY_COND logic_op BY_COND |
                 COMP_EXP
COMP_EXP =       ELEM comp_op ELEM
```

Figura 3.6: Gramática livre de contexto da sintaxe Gamma.

Como elementos do multiconjunto são escolhidos para serem processados por uma reação em tempo de execução, mapear elementos do multiconjunto na Etapa 2 é necessário para combinar todos os elementos do multiconjunto inicial $M$ para nós raízes do grafo dataflow. Este processo requer a replicação de grafos dataflow para cobrir todo o multiconjunto. Os elementos produzidos devem ser conectados ao grafo dataflow até o término do processamento das reações. Um exemplo é apresentado na Figura 3.7, onde o grafo dataflow gerado a partir da reação $R$ é instanciado 3 vezes para conectar todos os elementos do multiconjunto. O algoritmo que mapeia de maneira eficiente elementos para um grafo dataflow é complexo e está fora do escopo deste trabalho.

**Algorithm 2:** Conversão de uma reação em um grafo dataflow.

**Input:** Reação $R(R_L, B_L)$ com replace list $R_L$ e by list $B_L$
**Output:** Grafo Dataflow $D(I, E)$

1 $I \leftarrow \varnothing$; $E \leftarrow \varnothing$;
2 **foreach** *input element* $e \in R_L$ **do**
3 | $I \leftarrow$ node $i$; $value[i] \leftarrow e$;
4 **end**
5 **foreach** *by command* $(B_V, B_C) \in B_L$ **do**
6 | **if** $B_C$ *is not empty* **then**
7 | | **foreach** *comparison expression* $exp \in B_C$ **do**
8 | | | $I \leftarrow$ comparison node $c, \forall$ comparison operator $op \in exp$;
9 | | | $E \leftarrow (e, c), \forall e \in R_L$ used as input to comparison operator $op \in exp$;
10 | | | $I \leftarrow$ Steer node $st, \forall e \in R_L$ affected by result of comparison operator $op \in exp$;
11 | | | $E \leftarrow (e, st), \forall e \in R_L$ affected by result of operator $op \in exp$ related to Steer node $st$;
12 | | **end**
13 | | **foreach** *arithmetic expression* $exp \in B_V$ **do**
14 | | | $I \leftarrow$ arithmetic node $a, \forall$ arithmetic operator $op \in exp$;
15 | | | $E \leftarrow (se.true, a), \forall$ Steer node $se$ related with $e \in R_L$ used as input to arithmetic operator $op \in exp$;
16 | | **end**
17 | **else**
18 | | **foreach** *arithmetic expression* $exp \in B_V$ **do**
19 | | | $I \leftarrow$ arithmetic node $a, \forall$ arithmetic operator $op \in exp$;
20 | | | $E \leftarrow (e, a), \forall e \in R_L$ used as input to arithmetic operator $op \in exp$;
21 | | **end**
22 | **end**
23 **end**

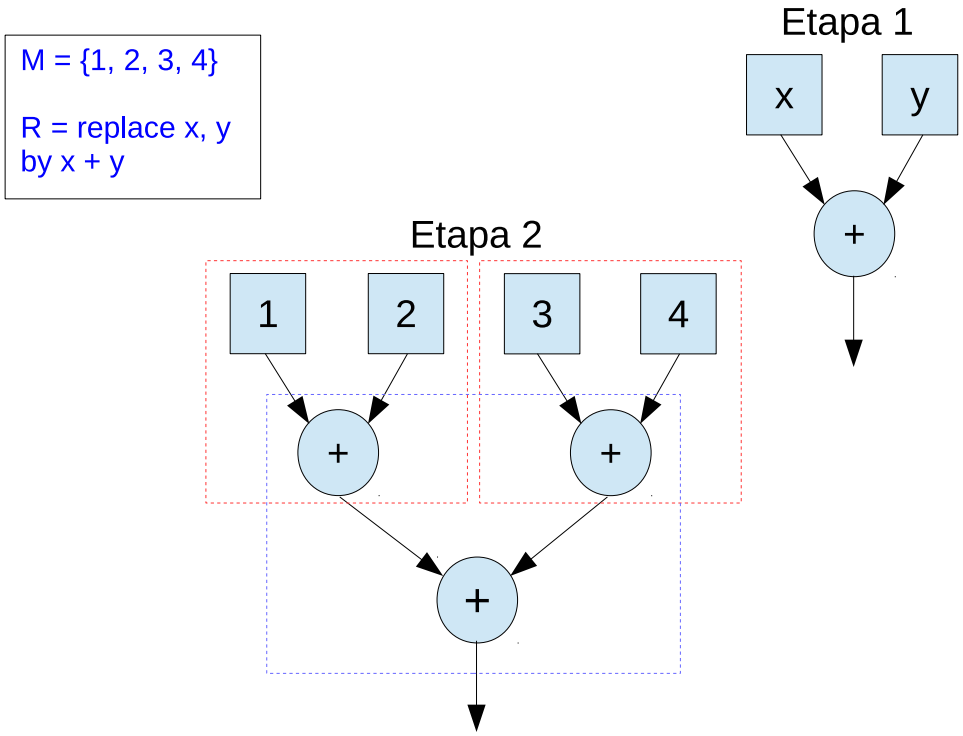


Figura 3.7: Exemplo de Gamma para Dataflow.

## 3.3 Prova Formal de Equivalência

Antes de abordar a prova de equivalência entre os modelos Gamma e Dataflow, é necessário generalizar as transformações entre estes modelos, uma vez que tais transformações serão utilizadas por ocasião da realização das provas formais. Inicialmente, analisaremos apenas aspectos relacionados à transformação de um grafo dataflow em um código escrito em Gamma, conforme relatado em [23], na Seção 3.3.2, iremos discutir detalhes acerca da transformação de um código Gamma para um grafo dataflow.

### 3.3.1 De um Grafo Dataflow para um código Gamma

#### 3.3.1.1 Análise Inicial

Conforme apresentado anteriormente (Seção 3.1), verificamos a possibilidade de equivalência entre os modelos Gamma e dataflow, onde os vértices de um grafo dataflow seriam transformados em reações Gamma e as arestas deste grafo seriam convertidas nos dados manipulados (consumidos e produzidos) pelas respectivas reações Gamma. Ademais, as ações e condições de cada reação seriam extraídas a partir da operação expressa pelo vértice e de seu relacionamento com as arestas. Desta forma, a quantidade de vértices de um grafo dataflow seria a mesma quantidade de reações do código Gamma equivalente.

Em nossa análise, o foco será o consumo e produção de dados que serão representados pelas arestas de determinado vértice. Por exemplo, um vértice que realiza uma soma pode necessitar de todos seus operandos de entrada para executar sua operação enquanto em um *Inctag*, somente uma aresta de entrada pode ser utilizada (normalmente a aresta de entrada de um *Inctag* que propaga o dado será responsável pela "ativação" de sua operação). Por outro lado, o dado produzido pelo vértice também pode ser disponibilizado por uma, algumas ou todas as arestas de saída, tal como um vértice do tipo *Steer*, onde somente uma das duas arestas de saída existentes receberá o dado a cada execução da operação (*TRUE* ou *FALSE*). Assim, para esta análise e por uma questão de generalização, estudaremos apenas as possibilidades de utilização das arestas de entrada e saída, abstraindo a operação expressa pelo vértice. É importante ressaltar que tais operações podem facilmente ser integradas ao código Gamma. Por exemplo, operações matemáticas são expressas pela cláusula $BY$ de um código Gamma, assim como o incremento de um rótulo de iteração de um *Inctag*. Um nó do tipo *Steer* não modifica nenhum valor, somente "redireciona" a saída de acordo com um sinal Booleano. Finalmente, um nó contendo uma operação de comparação possui arestas de saída com valores *TRUE* e *FALSE*. Em Gamma, a escolha do valor a ser atribuído à saída por um nó de comparação é realizada nas cláusulas $BY$ e $IF$ da descrição de uma reação Gamma. Outros tipos de operações expressas em um nó dataflow podem surgir, entretanto a análise dos dados a serem consumidos pelas arestas de entrada e produzidos pelas arestas de saída, continuará válida.

Para o primeiro passo desta generalização, vamos supor um vértice $R$ de um grafo dataflow qualquer. Tal vértice possui $n$ arestas de entrada $(I_1, I_2, ..., I_n)$ e $m$ arestas de saída $(O_1, O_2, ..., O_m)$, conforme apresentado na Figura 3.8.

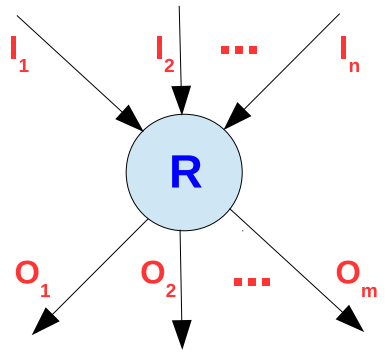


Figura 3.8: Vértice Dataflow genérico $R$.

#### 3.3.1.2 Análise das Entradas

Nesta seção, analisaremos o relacionamento entre um vértice e suas arestas de entrada, descrevendo o respectivo código Gamma para cada caso específico apresen-

tado. Desta forma, do ponto de vista do consumo de dados disponíveis nas arestas de entrada de um vértice dataflow, temos as seguintes possibilidades de utilização destas arestas: consumir dados de todas as arestas de entrada, consumir dados de somente 1 das $n$ arestas de entrada ou consumir dados de $k$ entre as $n$ arestas de entrada.

**– Consumir dados de todas as arestas de entrada**

Neste caso, a operação expressa pelo vértice deve utilizar os dados de todas as suas arestas de entrada. Portanto, a operação somente será realizada uma vez que todos os operandos de entrada estejam disponíveis.

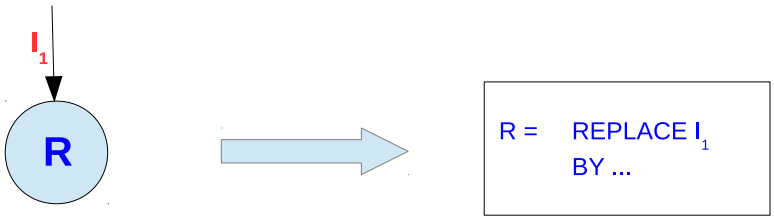


Figura 3.9: Vértice com uma única aresta de entrada.

A Figura 3.9, descreve a representação de um vértice que requer dados de sua única aresta de entrada ($I_1$) para executar a operação expressa por $R$.

Dessa forma, o código Gamma equivalente, utiliza a identificação explícita desta aresta (...*REPLACE* $I_1$...) em sua cláusula *REPLACE*. Assim, a reação Gamma só ocorrerá quando o elemento selecionado pela reação (que corresponde a uma aresta no grafo dataflow equivalente) for exatamente a aresta de entrada, neste caso $I_1$.

Outra possibilidade onde uma reação consome os dados de todas as suas arestas de entrada, seria quando o número de arestas é maior que um, conforme apresentado na Figura 3.10.

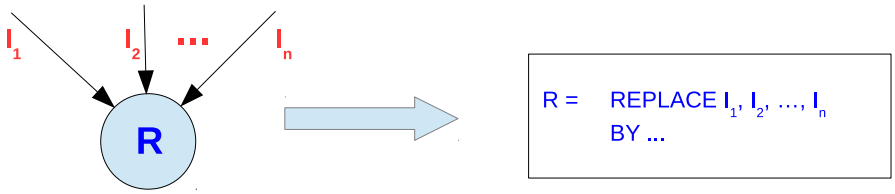


Figura 3.10: Vértice que necessita de dados de todas as suas $n$ arestas de entrada.

Semelhante ao caso onde o vértice possui uma única aresta de entrada, as identificações das arestas estarão presentes na cláusula *REPLACE* do código da reação (... *REPLACE* $I_1$, $I_2$,..., *In*...). Ou seja, esta reação irá executar quando os elementos selecionados pela reação corresponderem exatamente às arestas de

entrada de $R$.

### – Consumir dados de somente 1 das $n$ arestas de entrada

Agora iremos analisar o comportamento de um vértice que utiliza somente uma aresta, dentre as $n$ arestas de entrada disponíveis para executar sua operação, conforme apresentado em Figura 3.11. Isso significa que a primeira aresta de entrada que fornecer o dado será responsável por habilitar a execução da operação expressa pelo nó em questão.

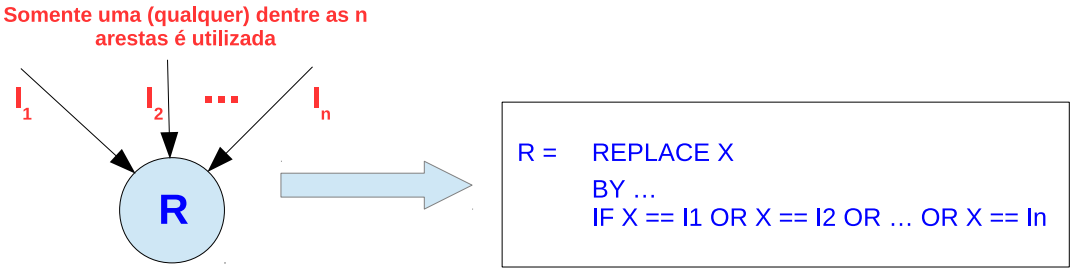


Figura 3.11: Vértice que necessita de dados de somente uma das suas $n$ arestas de entrada.

Devido à possibilidade de qualquer aresta de entrada ser utilizada, utilizamos uma variável auxiliar ($X$) na cláusula *REPLACE* do código Gamma. Esta variável $X$ é combinada com todas as identificações de aresta de entrada, expressas na cláusula *IF* (IF X == $I_1$ OR X == $I_2$ OR...OR X == $I_n$). Isso significa que $X$ tem a possibilidade de utilizar qualquer aresta de entrada. Portanto, quaisquer dados produzidos pelas arestas $I_1$ ou $I_2$ ou...ou $I_n$ irão satisfazer a condição de reação.

Observe que estamos considerando a possibilidade de que qualquer aresta de entrada possa produzir dados. Ou seja, não importa qual aresta disponibilizará os dados, o vértice é acionado quando receber dados de pelo menos uma de suas arestas de entrada. Entretanto, se o vértice necessita de dados de uma aresta específica, por exemplo $I_1$, a conversão para o código Gamma ocorre da mesma maneira conforme visto anteriormente (Consumir dados de todas as arestas de entrada), ou seja, a identificação da aresta segue a cláusula *REPLACE* (... REPLACE $I_1$ ...).

### – Consumir dados de $k$ entre as $n$ arestas de entrada

Neste caso, considere que o vértice $R$ necessita de $k$ arestas de entrada, onde $1 < k < n$, para executar a operação descrita em $R$. A Figura 3.12 demonstra o vértice $R$, suas arestas de entrada e o código Gamma equivalente, que consome dados de $k$ arestas de entrada.

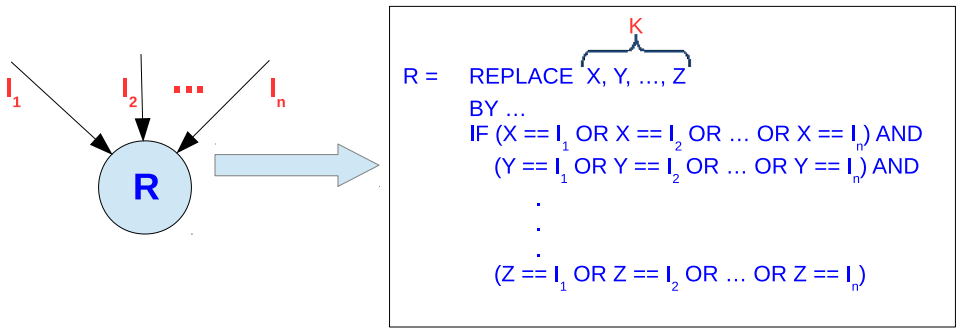


Figura 3.12: Vértice que necessita de dados de $k$ dentre suas $n$ arestas de entrada.

Note que este caso é a generalização do caso anterior, onde na cláusula *REPLACE* $k$ variáveis auxiliares são utilizadas (... REPLACE X, Y,... Z). Para cada variável auxiliar, é possível utilizar qualquer aresta de entrada, conforme expresso na cláusula *IF*:

```
IF  (X == I1 OR x == I2 OR... OR X == In) AND
    (Y == I1 OR Y == I2 OR... OR Y == In) AND
    .
    .
    .
    (Z == I1 OR Z == I2 OR... OR Z == In)
```

Observe que em um multiconjunto Gamma obtido a partir da conversão de um grafo dataflow, os elementos (que representam arestas) não possuem repetições, ou seja, os dados referentes a uma aresta são únicos.

Também é importante mencionar que o código Gamma equivalente a um grafo dataflow possui um multiconjunto inicial. Os elementos deste multiconjunto inicial representam as arestas iniciais do grafo, conforme apresentado na Figura 3.13.

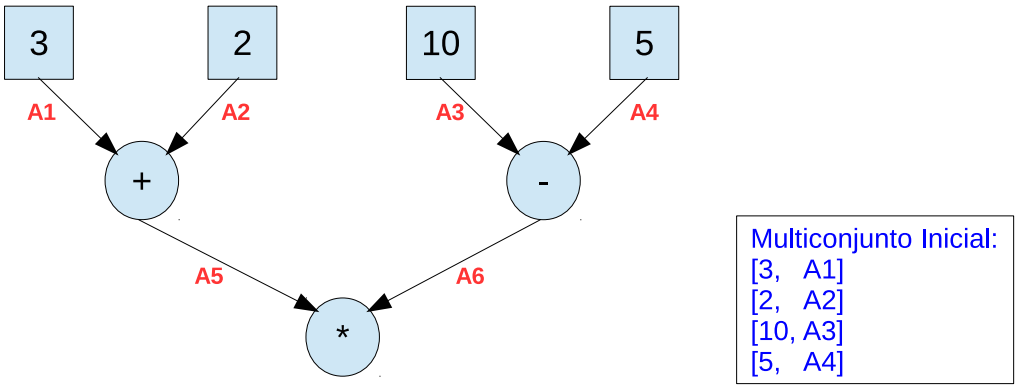


Figura 3.13: Exemplo de um multiconjunto inicial em Gamma, extraído de um grafo dataflow.

Observe na Figura 3.13 que o multiconjunto inicial consiste em tuplas referentes às arestas que inserem os dados iniciais no grafo (A1, A2, A3, e A4). Tuplas referentes às arestas $A5$ e $A6$ serão criadas em tempo de execução do código Gamma.

**– Generalização para análise das entradas**

Voltando à Figura 3.9, o código Gamma equivalente possui em sua cláusula *REPLACE* a identificação da aresta $I_1$ - a única aresta de entrada do vértice $R$. Entretanto, podemos substituir $I_1$ da cláusula *REPLACE* por $X$, adicionando a seguinte cláusula *IF*: (IF X == $I_1$), mantendo a mesma característica e funcionalidade da reação conforme apresentado a seguir:

```
R = REPLACE X
BY ...
IF X == I1
```

Se $X$ for igual a $I_1$, então a reação executa da mesma forma que apresentada no caso inicial (Consumir dados de todas as arestas de entrada). Assim, podemos utilizar a generalização apresentada na Figura 3.14 para analisar as arestas de entrada de qualquer vértice.

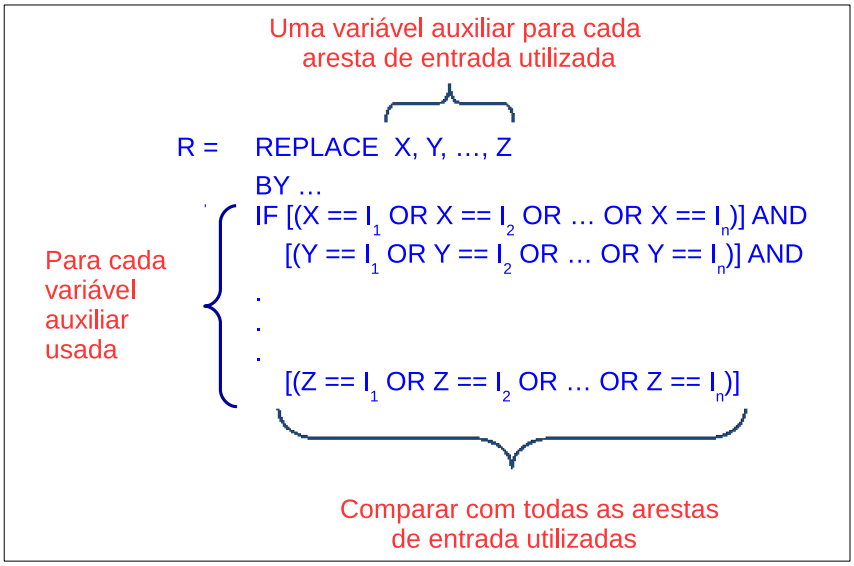


Figura 3.14: Generalização para a transformação de um grafo dataflow em um código Gamma do ponto de vista da análise das arestas de entrada.

A partir da generalização expressa em Figura 3.14 (que trata apenas da análise das arestas de entrada de um grafo), podemos derivar qualquer código Gamma a partir de um grafo dataflow configurando apenas a quantidade de variáveis

auxiliares utilizadas pela cláusula *REPLACE*. Se o vértice usa *1*, $k$ ou $n$ entradas, utilizaremos *1*, $k$ ou $n$ variáveis auxiliares. Para cada variável auxiliar, a cláusula *IF* analisa todas as entradas possíveis (*1*, $k$ ou $n$ - $I_1$, $I_2$, ..., $I_n$). A Figura 3.15 apresenta algumas comparações para a generalização proposta tendo em vista as arestas de entrada.

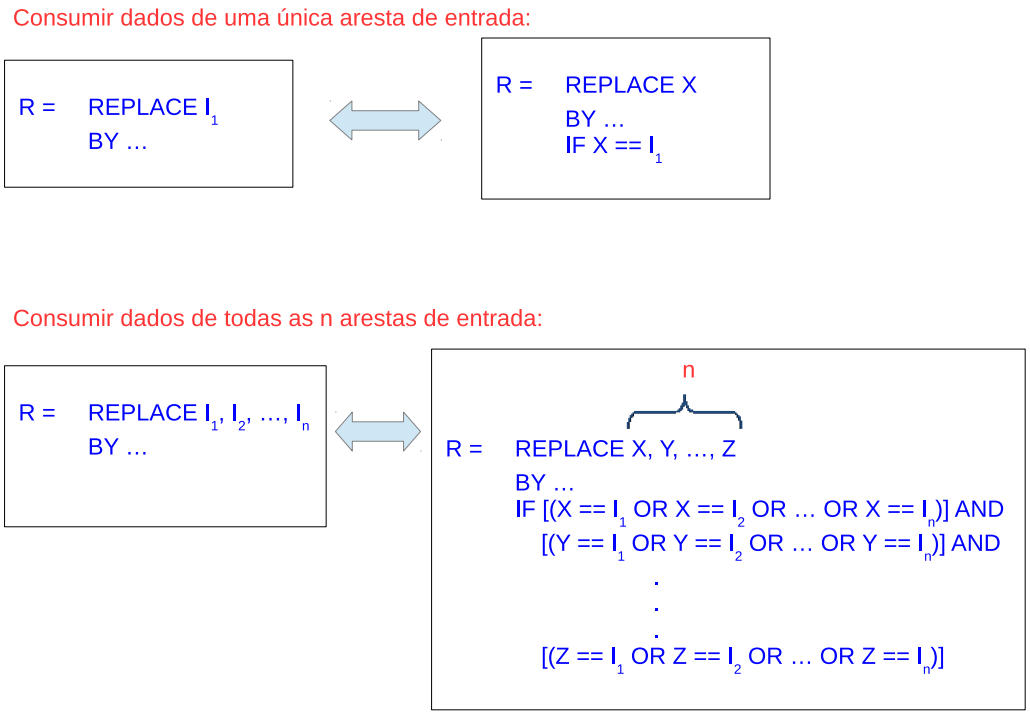


Figura 3.15: Detalhes sobre a generalização do tratamento das arestas de entrada.

### – Prova - Análise das Arestas de Entrada

Queremos mostrar que é possível fornecer um fragmento de um código de uma reação em Gamma que seja genérico ao ponto de poder ser utilizado na conversão de um vértice de um grafo dataflow em uma reação Gamma. Entretanto, neste primeiro momento, iremos fornecer tal fragmento de código do ponto de vista das arestas de entrada do vértice equivalente.

**Lemma 1.** *Seja $R$ um vértice de um grafo dataflow $G$ qualquer, $n$ a quantidade de arestas de entrada de $R$ e $R1$ a reação Gamma equivalente. O fragmento de código Gamma que generaliza o tratamento das arestas de entrada é dado pelo seguinte:*

```
R1 = REPLACE X, Y, ..., Z
BY ...
IF (X == I1 OR X == I2 OR ... OR X == In) AND
   (Y == I1 OR Y == I2 OR ... OR Y == In) AND
   .
   .
   .
   (Z == I1 OR Z == I2 OR ... OR Z == In)
```

*Demonstração.* A prova será realizada pelo princípio da indução finita nas arestas de entrada de $R$.

**Base:** Caso $n = 1$ ($R$ com somente uma aresta de entrada), por definição, o código Gamma equivalente corresponde a:

```
R1 = REPLACE X
BY ...
IF X == I1
```

O que faz valer o enunciado, pois como a quantidade de arestas é igual a 1, somente uma variável auxiliar seria utilizada (neste caso $X$) e só existiria uma aresta de entrada com a identificação $I_1$.

Já no caso de $R$ possuir duas arestas de entrada ($n = 2$), o fragmento de código seria dado por:

```
R1 = REPLACE X, Y
BY ...
IF (X == I1 OR X == I2) AND
   (Y == I1 OR Y == I2)
```

O que também vale, uma vez que neste caso duas variáveis auxiliares seriam utilizadas na cláusula *REPLACE* e estas seriam comparadas às arestas $I_1$ e $I_2$.

**Hipótese Indutiva:** o enunciado vale para uma quantidade $k$ de arestas de entrada. Desta forma, teríamos o código Gamma a seguir:

```
R1 = REPLACE X, Y, ..., Z
BY ...
IF (X == I1 OR X == I2 OR ... OR X == Ik) AND
   (Y == I1 OR Y == I2 OR ... OR Y == Ik) AND
   .
   .
   .
   (Z == I1 OR Z == I2 OR ... OR Z == Ik)
```

Onde, $X, Y, ..., Z$ (cláusula *REPLACE*) totalizaria a quantidade de k elementos.

**Passo Indutivo:** ao inserirmos mais uma aresta de entrada à quantidade de $k$ arestas, teríamos o seguinte código Gamma:

```
R1 = REPLACE X, Y, ..., Z, Z'
BY ...
IF [(X == I1 OR X == I2 OR ... OR X == Ik) OR (X == Ik+1)] AND
   [(Y == I1 OR Y == I2 OR ... OR Y == Ik) OR (Y == Ik+1)] AND
   .
   .
   .
   [(Z == I1 OR Z == I2 OR ... OR Z == Ik) OR (Z == Ik+1)] AND
   [(Z' == I1 OR Z' == I2 OR ... OR Z' == Ik) OR (Z' == Ik+1)]
```

Note que a quantidade de elementos selecionados pela cláusula *REPLACE* está diretamente relacionada com a quantidade de arestas de entrada de $R$. Assim, é correto afirmar que se as variáveis $X, Y, ..., Z$ referem-se a uma quantidade de $k$ arestas de entrada, ao utilizarmos mais uma aresta de entrada, totalizando $K + 1$ arestas, as variáveis $X, Y, ..., Z, Z'$ referenciar-se-ão a todas as $K + 1$ arestas de entrada. Como $K+1$ é exatamente uma unidade a mais que $k$ poderíamos substituir a cláusula *REPLACE* por $X, Y, ..., Z'$. Aplicando raciocínio semelhante, na cláusula *IF* temos uma linha de código referente a cada variável auxiliar. Ora, como a quantidade de arestas aumentou em uma unidade, a comparação em cada linha terá de ir até a arestas de entrada cuja *tag* é $I_k + 1$, abrangendo assim todas as arestas de entrada. Da mesma forma, será necessária mais uma linha onde a variável auxiliar $Z'$ será comparada a cada aresta de entrada utilizada. Igualmente, como a quantidade de $k$ precede $k+1$ arestas de entrada, é correto afirmar que cada linha de comparação da cláusula *IF* deverá seguir até $k+1$, podendo-se omitir o elemento $k$. Assim, temos o código apresentado a seguir:

```
R1 = REPLACE X, Y, ..., Z'
BY ...
IF (X == I1 OR X == I2 OR ... OR X == Ik+1) AND
   (Y == I1 OR Y == I2 OR ... OR Y == Ik+1) AND
   .
   .
   .
   (Z' == I1 OR Z' == I2 OR ... OR Z' == Ik+1)
```

Por fim, substituindo-se o elemento $k + 1$ por $n$, temos que a hipótese vale. □

#### 3.3.1.3 Análise das Saídas

Continuando com a análise genérica de um vértice, do ponto de vista de suas arestas de entrada e saída, visando a produção do correspondente código Gamma, agora iremos estudar somente os dados produzidos pela operação inerente ao vértice e disponibilizados através de suas arestas de saída. Da mesma forma como apresentado na Seção 3.3.1.2, a operação propriamente dita será abstraída nesta análise. Assim teremos as seguintes possibilidades de produção de dados através de arestas de saída: Produzir dados para todas as arestas de saída ou Produzir 1 ou $k$ dados através de uma das $m$ arestas de saída.

**– Produzir dados para todas as arestas de saída**

Na situação onde todas as arestas de saída irão produzir dados como resultado da operação do vértice, teremos dois casos distintos: produzir dados para a única aresta de saída existente ou para as $m$ arestas de saída existentes.

Para o primeiro caso (uma aresta de saída), a Figura 3.16 ilustra detalhes do código Gamma correspondente, somente no que diz respeito às saídas.

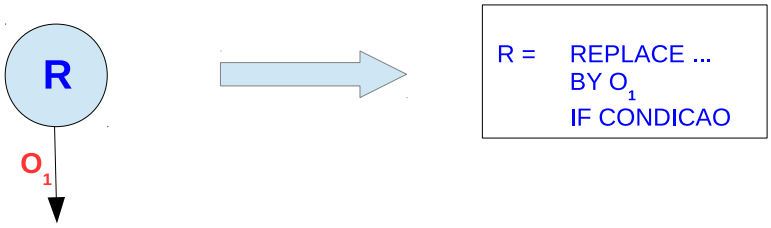


Figura 3.16: Vértice com uma única aresta de saída.

Repare que no caso da análise das arestas de saída, iremos focar na cláusula *BY* do código Gamma correspondente, já que esta cláusula é responsável por modificar o multiconjunto, inserindo, excluindo ou modificando elementos. Na Figura 3.16 temos que $O_1$ será criado assim que suas condições específicas forem satisfeitas. Se a condição for, por exemplo, encontrar os elementos no multiconjunto esta *CONDICAO* poderá ser suprimida ou substituída por *TRUE*. Ou seja, se a cláusula *REPLACE* for composta somente por $I_1$ (se a condição de reação for somente a existência da aresta $I_1$), esta *CONDICAO* poderá ser suprimida. Maiores detalhes serão vistos na Seção 3.3.1.4.

Para o caso onde todas as arestas de saída irão receber os dados produzidos, o código Gamma equivalente é apresentado na figura Figura 3.17.

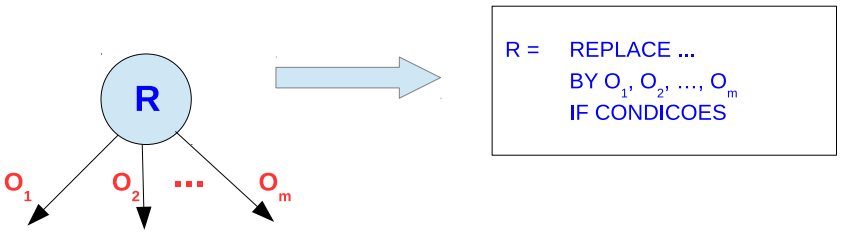


Figura 3.17: Vértice que irá transmitir seus dados em todas as suas $m$ arestas de saída.

**– Produzir 1 ou $k$ dados através de uma das $m$ arestas de saída**

Nesta situação, englobamos o caso de produção de 1 ou $k$, dentre as $m$ arestas de saída, com $k < m$. Ou seja, aqui iremos disponibilizar os dados para todas as arestas que satisfizerem suas condições específicas, conforme apresentado na Figura 3.18.

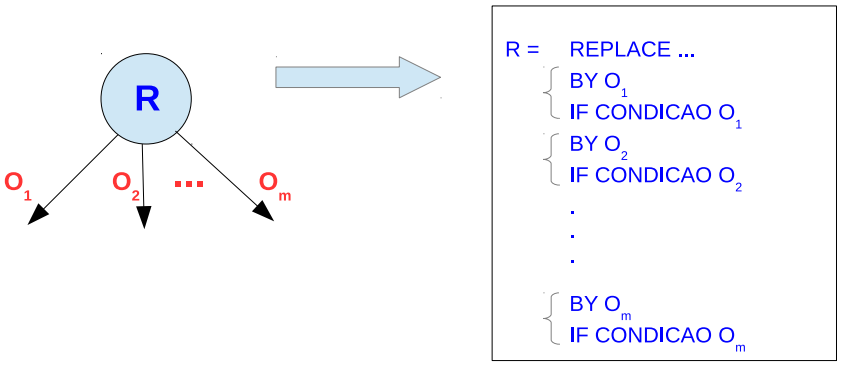


Figura 3.18: Vértice que envia seus dados produzidos através de 1 ou $k$ arestas de saída dentre as $m$ existentes.

Repare que sempre que uma saída satisfizer sua condição específica (explicitada na cláusula $IF$), o elemento correspondente será criado no multiconjunto (Cláusula $BY$). Note ainda que o código correspondente funciona para uma ou $k$ saídas, onde o controle da disponibilização dos dados é sempre dado pela cláusula $IF$ de cada $BY$ correspondente.

**– Generalização para análise das saídas**

Ainda tendo como base o fragmento de código Gamma apresentado na Figura 3.18, vimos que o mesmo poderá ser utilizado para disponibilizar dados tanto para 1 quanto para $k$ das $m$ arestas de saída disponíveis. Entretanto, note que o mesmo código também pode ser utilizado para a generalização do caso onde os dados são disponibilizados para todas as saídas, conforme vimos no caso de **produzir dados para todas as arestas de saída** e como apresentado a seguir na Figura 3.19.

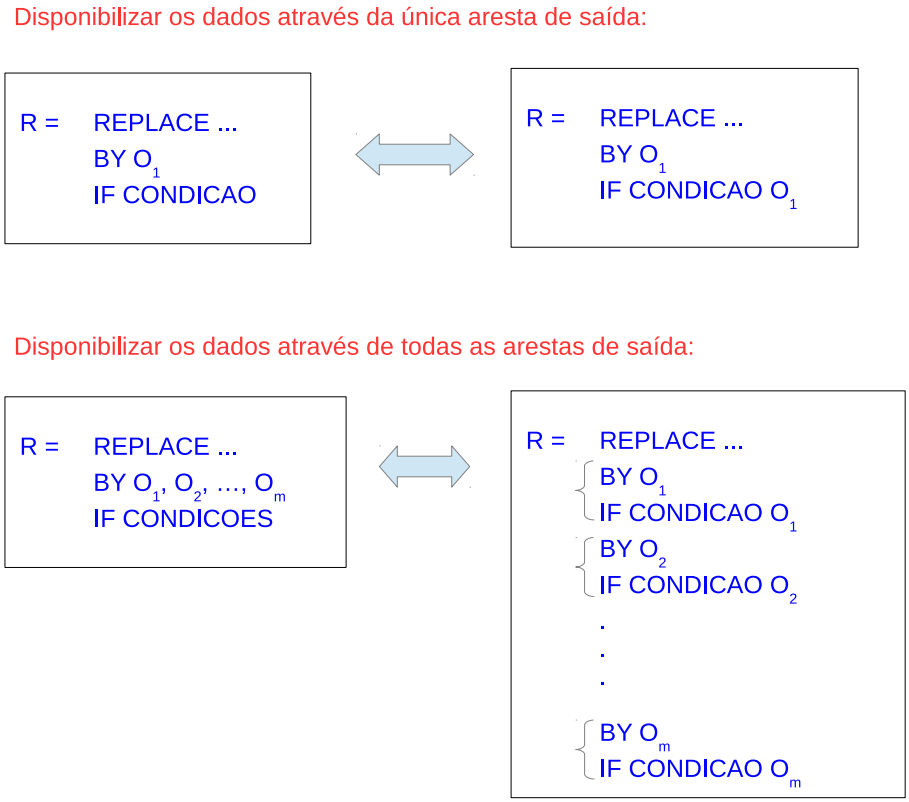


Figura 3.19: Detalhes sobre a generalização para o tratamento das arestas de saída.

Repare que para o caso onde todas as arestas de saída irão receber dados, poderemos utilizar o código generalizado, onde:

```
CONDICAO O1 == CONDICAO O2 == ... == CONDICAO Om.
```

**– Prova - Análise das Arestas de Saída**

Queremos mostrar que é possível fornecer um fragmento de um código de uma reação em Gamma que seja genérico ao ponto de poder ser utilizado na conversão de um vértice de um grafo dataflow em uma reação em Gamma. Entretanto, agora iremos fornecer tal fragmento de código do ponto de vista das arestas de saída do vértice equivalente.

**Lemma 2.** *Seja $R$ um vértice de um grafo dataflow $G$ qualquer, $m$ a quantidade de arestas de saída de $R$ e $R1$ a reação Gamma equivalente. O fragmento de código Gamma que generaliza o tratamento das arestas de saída é dado por:*

```
R1 = REPLACE ...
BY O1
IF CONDICAO O1

BY O2
IF CONDICAO O2
.
.
.
BY Om
IF CONDICAO Om
```

*Demonstração.* A prova será realizada pelo princípio da indução finita nas arestas de saída de $R$.

**Base:** caso $m = 1$ ($R$ com somente uma aresta de saída), por definição, o código Gamma equivalente corresponde a:

```
R1 = REPLACE ...
     BY O1
     IF CONDICAO O1
```

O que faz valer o enunciado, pois como a quantidade de arestas de saída é igual a 1, somente uma cláusula $BY$ seria utilizada, neste caso aquela referente a aresta $O_1$.

Já no caso de $R$ possuir duas arestas de saída ($m = 2$), o fragmento de código Gamma seria dado por:

```
R1 = REPLACE ...
     BY O1
     IF CONDICAO O1

     BY O2
     IF CONDICAO O2
```

O que também vale, uma vez que neste caso duas cláusulas $BY$ seriam utilizadas, correspondendo às duas arestas de saída, $O_1$ e $O_2$ utilizadas pelo vértice $R$.

**Hipótese Indutiva:** O enunciado vale para uma quantidade $K$ de arestas de saída de $R$. Desta forma teríamos o código apresentado a seguir:

```
R1 = REPLACE ...
BY O1
IF CONDICAO O1
BY O2
IF CONDICAO O2
```

```
.
.
.
BY Ok
IF CONDICAO Ok
```

**Passo Indutivo:** Ao inserirmos mais uma aresta de saída à quantidade de $k$ arestas teríamos o seguinte:

```
R1 = REPLACE ...
BY O1
IF CONDICAO O1

BY O2
IF CONDICAO O2
.
.
.
BY Ok
IF CONDICAO Ok

BY Ok+1
IF CONDICAO Ok+1
```

Note que a quantidade de cláusulas "$BY$ ... $IF$" existente está diretamente relacionada com a quantidade de arestas de saída de $R$. Assim, é correto afirmar que se as cláusulas $BY$ referentes à $O_1, O_2, ..., O_k$ referem-se a uma quantidade de $k$ arestas de saída, ao utilizarmos mais uma aresta de saída, totalizando $K + 1$ arestas, as cláusulas $BY$ referentes a $O_1, O_2, ..., O_k, O_{k+1}$ referenciar-se-ão a todas as $K + 1$ arestas de saída. Como $K + 1$ é exatamente uma unidade a mais que $K$, poderíamos substituir o intervalo de cláusulas $BY$ $O_1, O_2, ..., O_k, O_{k+1}$ pelo intervalo correspondente a $O_1, O_2, ..., O_{k+1}$. Assim temos que:

```
R1 = REPLACE ...
BY O1
IF CONDICAO O1

BY O2
IF CONDICAO O2
.
.
.
BY Ok+1
IF CONDICAO Ok+1
```

Por fim, substituindo-se o elemento $K+1$ por $m$, temos que a hipótese vale. □

#### 3.3.1.4 Generalização de um código Gama correspondente a um vértice dataflow qualquer

Após o estudo da geração do código Gamma, a partir de um grafo dataflow, do ponto de vista de suas arestas de entrada (realizado na Seção 3.3.1.2) e de suas arestas de saída (Seção 3.3.1.3), se faz necessária a análise conjunta destes aspectos visando a geração de um código Gamma completo, ou seja, considerando as arestas de entrada e arestas de saída de um vértice de um grafo dataflow.

Conforme vimos nas Seções anteriores, as arestas de entrada de um grafo dataflow afetam diretamente a cláusula *REPLACE* de um código Gamma, responsável pela escolha dos elementos para reagir, ao passo que as arestas de saída afetam a cláusula *BY*, responsável pela criação, modificação e exclusão de elementos no multiconjunto. Entretanto, tanto arestas de entrada quanto arestas de saída afetam a cláusula *IF*. A boa notícia é que estas características podem ser combinadas sem prejuízo à corretude.

Tomemos por exemplo o vértice $R_1$ da Figura 3.20, onde das duas entradas disponíveis somente uma é utilizada pela operação expressa por $R_1$. Além disso, $R_1$ disponibiliza seu produto por todas as três arestas de saída. O código Gamma correspondente encontra-se também na Figura 3.20.

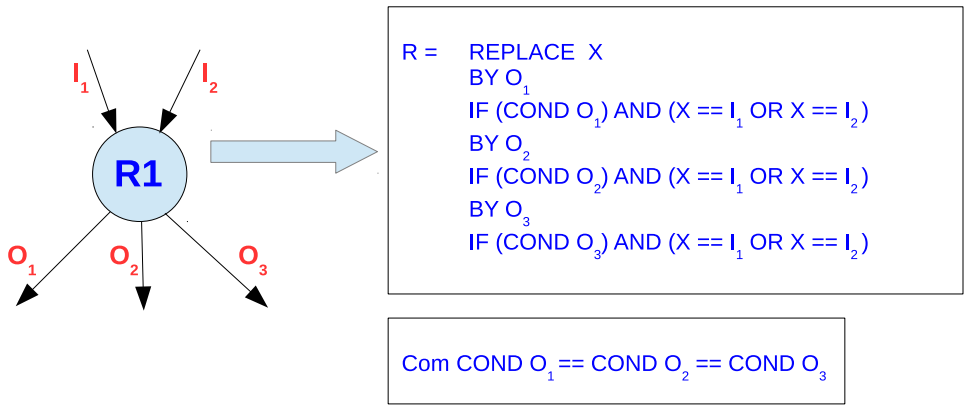


Figura 3.20: Exemplo de código Gamma para um vértice *R1* específico.

Repare que as condições específicas para a geração dos dados nas arestas de saída (COND $0_1$, COND $O_2$, and COND $O_3$) foram combinadas com as condições específicas para a escolha dos dados advindos das arestas de entrada (X == $I_1$ OR X == $I_2$) através da diretiva *AND*. Desta forma, podemos combinar as informações que afetam a cláusula *IF* sem perda da corretude da solução. Assim, na Figura 3.21, sugerimos um código Gamma para traduzir qualquer vértice dataflow, independente do comportamento de suas arestas de entrada e saída, sem levar em consideração a operação expressa pelo vértice.

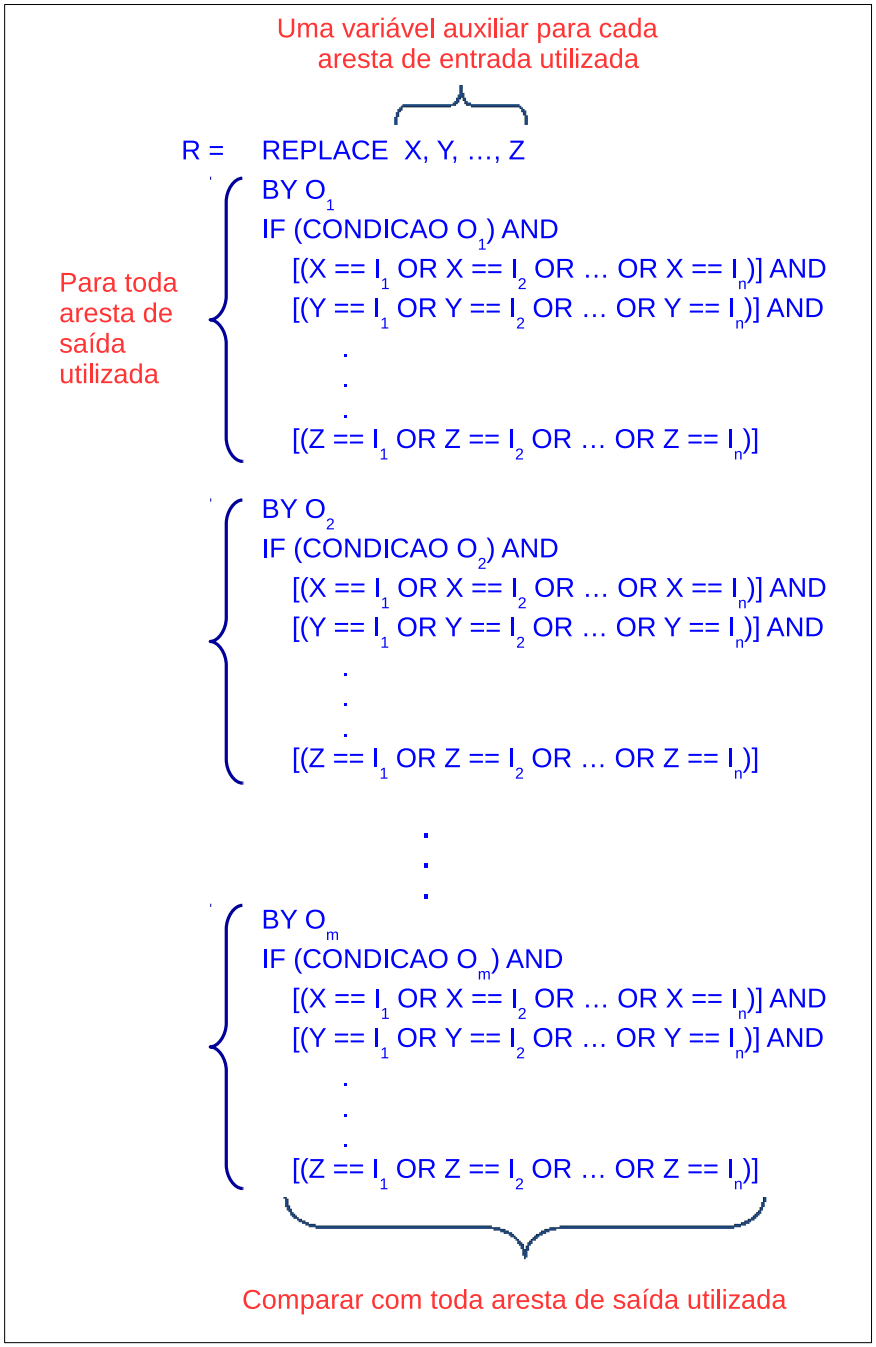


Figura 3.21: Reação Gamma genérica capaz de traduzir o comportamento de um vértice dataflow qualquer.

#### 3.3.1.5 Prova de equivalência da transformação de um grafo dataflow em um código Gamma

Nas duas provas anteriores, mostramos que é possível fornecer o código de uma reação em Gamma equivalente a um vértice dataflow, analisando as arestas de entrada e saída deste. Ou seja, sempre será possível exprimir o código de uma reação equivalente a um vértice de um grafo dataflow qualquer. Vale ressaltar que as reações que serão produzidas nesta conversão de Dataflow para Gamma possuem dependência de dados, entretanto o acoplamento é fraco entre si. Em outras palavras, cada reação utiliza dados produzidos por outra, contudo, o processamento executado pela reação não depende funcionalmente de outra. Dessa maneira, por exemplo, a atitude de incluir mais uma reação em determinado código Gamma não corresponde a uma atividade complexa.

Agora, queremos mostrar que é possível fornecer um código Gamma completo equivalente a um grafo dataflow de forma que cada vértice deste grafo irá corresponder a uma reação no código Gamma de destino.

**Lemma 3.** *Seja $G$ um grafo dataflow qualquer, $n$ a quantidade de vértices de $G$ e $R_n$ uma reação de índice $n$ de um código Gamma $C$. A quantidade de reações do código Gamma $C$, equivalente ao grafo dataflow $G$ é dada por:*

$$C = \{R_1, R_2, ..., R_n\}$$

*Demonstração.* A prova será realizada pelo princípio da indução finita na quantidade de reações de $C$.

**Base:** Caso $n = 1$ ($G$ com somente um vértice), por definição, o código Gamma equivalente é formado pela seguinte quantidade de reações:

$$C = \{R_1\}$$

O que faz valer o enunciado, pois como a quantidade de vértices é igual a 1, o código Gamma equivalente seria composto por somente uma reação.

Já no caso de $G$ possuir dois vértices o quantitativo de reações que irá compor o código completo em Gamma seria composto por duas reações:

$$C = \{R_1, R_2\}$$

O que também faz valer o enunciado.

**Hipótese Indutiva:** o enunciado vale para uma quantidade $k$ de vértices. Desta forma teríamos:

$$C = \{R_1, R_2, ..., R_k\}$$

**Passo Indutivo:** Ao inserirmos mais um vértice à quantidade de $K$ vértice teríamos:

$C = \{R_1, R_2, ..., R_k\} U \{R_{k+1}\}$
Sendo $AUB = \{x \mid x \in A \; ou \; x \in B\}$

$\{R_1, R_2, ..., R_k\} U \{R_{k+1}\} = \{R_1, R_2, ..., R_k, R_{k+1}\} = \{R_1, R_2, ..., R_{k+1}\}$

Por fim, substituindo-se o elemento $K + 1$ por $n$, temos que a hipótese vale. □

### 3.3.2 De um código Gamma para um Grafo Dataflow

Após apresentarmos as análises e a prova da transformação de um grafo dataflow em um código Gamma, iremos iniciar as análises de um código Gamma visando extrair um grafo dataflow equivalente, conforme foi introduzido em [23].

#### 3.3.2.1 Análise Inicial

Utilizando raciocínio semelhante ao que apresentamos na discussão sobre a generalização das transformações de um grafo dataflow para um código Gamma, a intenção agora é mostrar que sempre poderemos extrair um grafo dataflow a partir de um código fonte escrito em Gamma. Assim, teremos como origem um código Gamma composto por várias reações onde cada reação dará origem a vértices de um grafo dataflow.

Veremos que um código Gamma extraído de um grafo dataflow, facilmente consegue ser convertido novamente em seu grafo dataflow original. Tal facilidade deve-se ao fato do determinismo na seleção dos dados a serem utilizados pelo grafo dataflow que acabará ficando refletido no código Gamma equivalente. Ou seja, este código Gamma extraído de um grafo carrega o determinismo na seleção dos elementos no multiconjunto.

Entretanto um código originalmente escrito em Gamma carrega consigo um não determinismo, inerente ao próprio paradigma Gamma. A seleção dos elementos a serem utilizados por uma reação não é previsível, fazendo com que tenhamos vários grafos equivalentes às instâncias de execução de um programa em Gamma. Desta forma, uma única reação em Gamma poderá dar origem a múltiplos subgrafos da solução final.

#### 3.3.2.2 Dataflow - Gamma - Dataflow

Analisando a Figura 3.22, verificamos um grafo dataflow composto por 3 vértices - $R_1$, $R_2$, $R_3$ - e seu respectivo código Gamma, composto exatamente por três reações.

Na Figura 3.22 utilizamos em nossa representação dos códigos Gamma, algumas variáveis auxiliares, como por exemplo: *Id1* e *Id2* (utilizados para referenciar-se aos elementos que representam os dados das tuplas) e *t1*, *t2* (utilizados para referenciar *tags* das tuplas). A primeira representação do código Gamma (código 1) visa

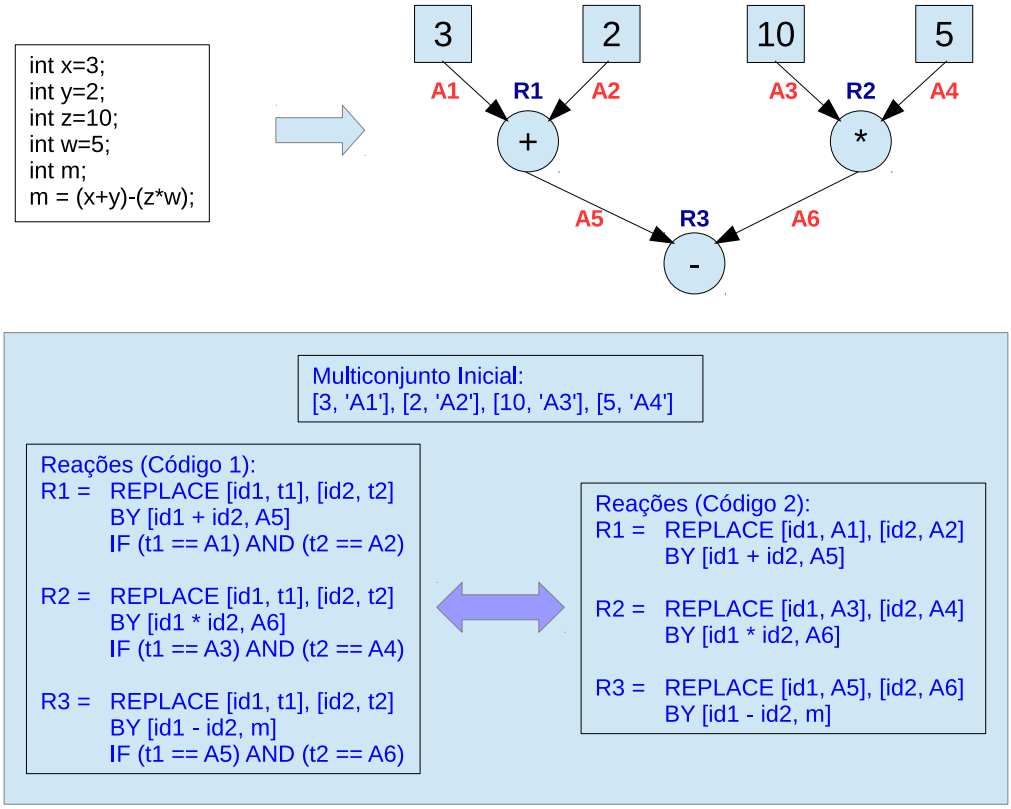


Figura 3.22: Exemplos de códigos em linguagem imperativa, dataflow e Gamma.

ser coerente ao código genérico apresentado na Figura 3.21 (Código de uma reação genérica em Gamma capaz de traduzir o comportamento de um vértice dataflow qualquer). Entretanto, é possível a realização de algumas simplificações neste código visando a facilitação no entendimento. Neste caso, sugerimos o "Código 2", onde as comparações da cláusula *IF* foram suprimidas e a identificação das *tags* foram incluídas na cláusula *REPLACE*, sem perda de corretude do código Gamma. Portanto, a Figura 3.22 representa um código escrito em linguagem imperativa, seu respectivo grafo dataflow e finalmente o código Gamma extraído deste grafo.

Perceba ainda na Figura 3.22 que cada elemento do multiconjunto inicial $M$ é representado por uma tupla $[x, y]$, onde $x$ representa o valor da aresta (dado) e $y$ representa seu identificador unívoco (*tag*). É importante notar que o multiconjunto inicial é extraído a partir das entradas de dados no grafo dataflow, ou seja, os elementos iniciais do multiconjunto serão obtidos pelos vértices "quadrados" e suas respectivas arestas de saída. Desta forma, o fluxo de dados de entrada não permite nenhum tipo de alteração, ou seja, para este exemplo, não seria possível o valor "10" ser operando de entrada para o vértice $R1$, por exemplo.

Dessa maneira, a escolha dos elementos que serão utilizados como operandos de um vértice de um grafo dataflow é determinística, ao passo que neste grafo, cada entrada de dados (representado no grafo da Figura 3.22 por vértices quadrados) é

operando de vértices específicos. Consequentemente, a geração do código Gamma equivalente "herda" este determinismo, uma vez que as reações somente reagirão caso elementos específicos sejam selecionados. Em outras palavras, analisando a Figura 3.22, e tomando como base a reação $R2$ do segundo código, verificamos que esta reação só reage quando os elementos cujas *tags* sejam iguais a $A3$ e $A4$, respectivamente, são selecionados. Desta forma, realizar a transformação de um código Gamma, que foi gerado a partir de um grafo dataflow, novamente em um grafo dataflow torna-se trivial. Cada reação dará origem a um único vértice e as interligações das arestas entre estes serão identificadas pelas *tags* manipuladas no código Gamma. Além disso, a partir do multiconjunto inicial, a entrada de dados no grafo (arestas de saída dos vértices quadrados) consegue ser extraída, conforme apresentamos na Figura 3.23.

Repare na Figura 3.23 que a partir do multiconjunto inicial podem ser extraídos os vértices iniciais e suas respectivas arestas de saída (o valor do vértice representa o campo "valor" e a *tag* da aresta representa o campo "*tag*" na tupla $[valor, tag]$). Neste exemplo, em tempo de execução, este multiconjunto irá sendo transformado até que um único elemento reste, sendo esta a solução do problema. Repare ainda que, a partir da análise de cada reação individualmente pode-se extrair as arestas de entrada e saída de cada vértice, aplicando raciocínio análogo ao que apresentamos na Seção 3.3.1, mas agora do ponto de vista de uma reação gerando um vértice. Ou seja, tendo uma reação $R$ qualquer, os elementos manipulados / selecionados pela cláusula *REPLACE* darão origem às arestas de entrada do vértice $R$, ao passo que os elementos criados pelas cláusulas *IF* corresponderão às arestas de saída deste mesmo vértice. Ao final, tendo todos os vértices extraídos do código Gamma, é possível criar o grafo em si, respeitando as dependências das arestas demonstradas neste mesmo grafo. Por exemplo, $R3$ tem duas arestas de entrada $A5$ e $A6$ que são arestas de saída de $R1$ e $R2$, respectivamente. Este fato proporciona a "conexão" destes três vértices através de suas arestas de entrada e saída. Por fim, as arestas de saída dos vértices representados por quadrados são arestas de entrada de outros vértices, o que leva a geração do grafo equivalente. Desta forma, um código Gamma gerado a partir da transformação de um grafo dataflow sempre poderá ser convertido novamente em seu grafo dataflow original.

#### 3.3.2.3 Provas - Gamma para Dataflow

Nesta seção iremos abordar as provas necessárias para analisar as conversões de um código Gamma para um grafo dataflow. Desta forma, tendo em vista a sintaxe de uma reação Gamma, iremos separar tais provas em duas partes conforme apresentado a seguir.

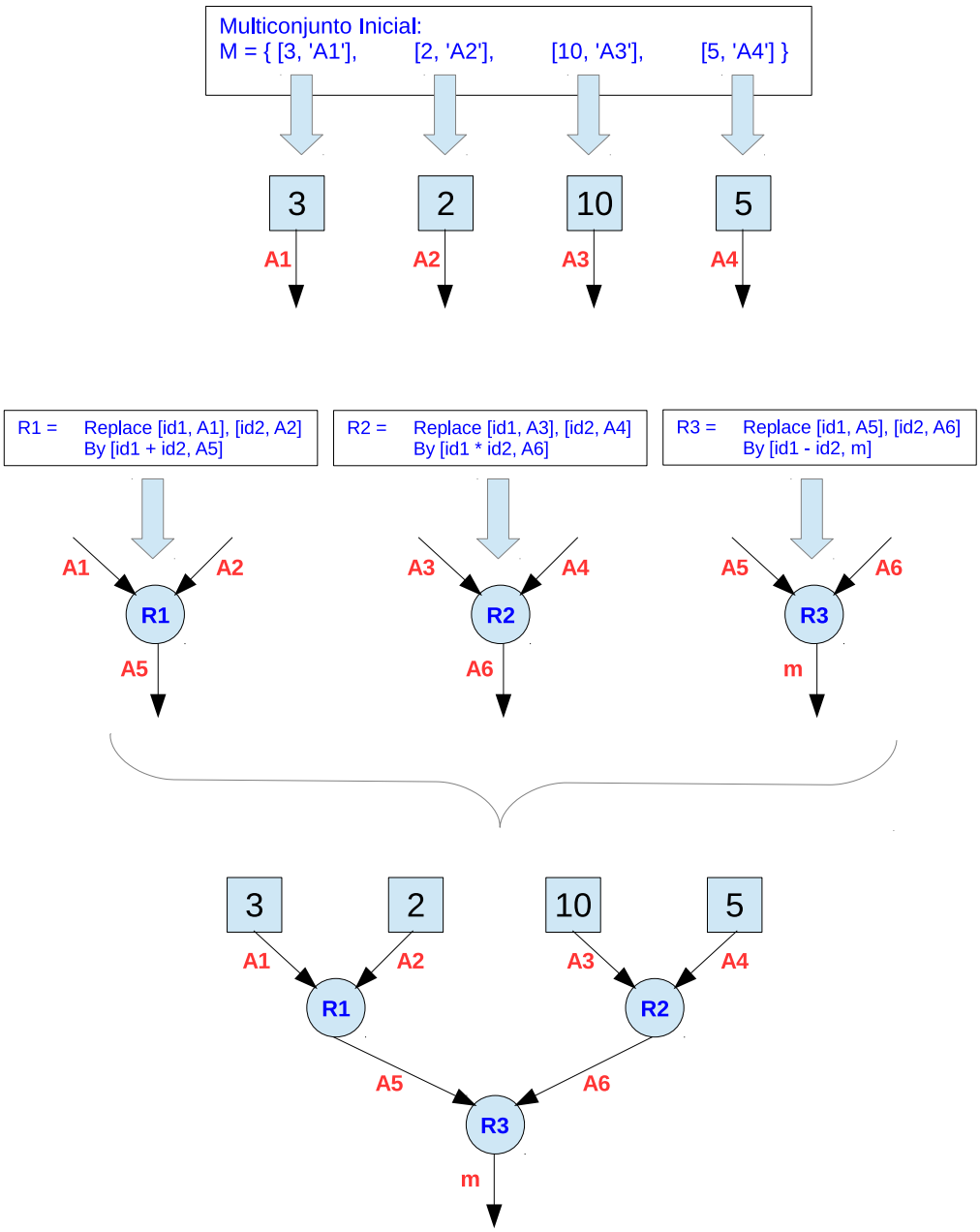


Figura 3.23: Exemplo de transformações entre elementos do multiconjunto e reações Gamma em vértices e arestas de um grafo dataflow.

– **Análise dos elementos manipulados pela cláusula *REPLACE***

Conforme discutido anteriormente, queremos mostrar que, a partir do código de uma reação em Gamma, é possível extrair as arestas de entrada e saída do vértice dataflow equivalente, uma vez que cada reação corresponderá a um vértice específico de um grafo dataflow. Entretanto, para facilitar a análise e a prova, inicialmente iremos extrair somente as arestas de entrada do vértice em questão, dividindo esta prova em duas etapas. Desta forma, o código Gamma que será analisado omitirá detalhes da cláusula *BY*, uma vez que as informações referentes às arestas de entrada serão obtidas através da análise das cláusulas *REPLACE* e *IF*.

**Lemma 4.** *Seja $R$ uma reação Gamma qualquer, $n$ a quantidade de elementos do multiconjunto selecionados pela cláusula REPLACE de $R$, $v$ o vértice do grafo dataflow $G$ equivalente a $R$ e $E_n$ o conjunto de arestas de entrada de $R$. Definimos o grafo dataflow $G(V, E)$ onde $V$ é o conjunto de vértices e $E$ o conjunto de arestas de $G$. Definimos ainda o conjunto de arestas de entrada de $v$ por: $E_n = \{e_1, e_2, ..., e_n\}$ onde cada aresta $e_x$ é definida por um par de vértices: $e_x = (v_x, v)$. O conjunto $In$, definido por $I_n = \{i_1, i_2, ..., i_n\}$, é o conjunto de rótulos atribuídos às arestas do conjunto $E_n$, de forma que $e_1$ tem rótulo $i_1$, $e_2$ tem rótulo $i_2$ e assim sucessivamente. Podemos extrair $E_n$ de $v$ a partir da reação $R$ conforme ilustrado na Figura 3.24. Onde cada variável auxiliar $(X, Y, ..., Z)$ manipulada pela cláusula REPLACE irá criar uma nova aresta em $E_n$.*

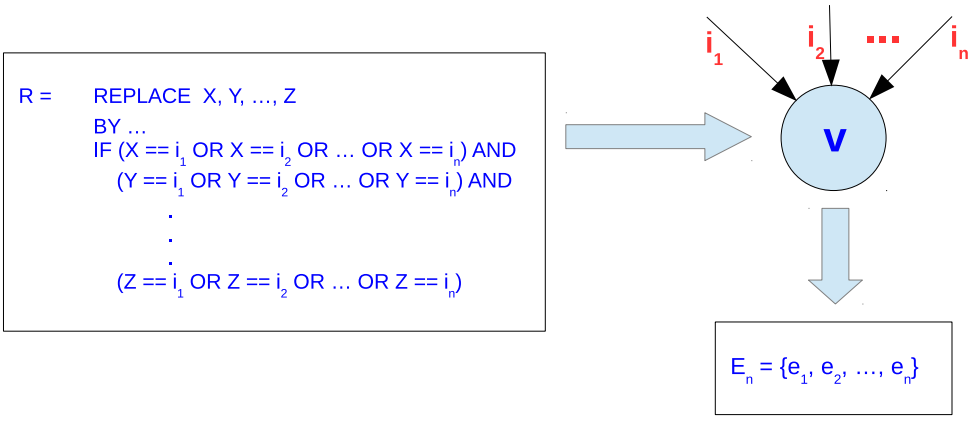


Figura 3.24: Extraindo arestas de entrada de $V$ a partir de uma reação $R$.

*Demonstração.* A prova será realizada pelo princípio da indução finita nos elementos manipulados pela cláusula *REPLACE* de $R$.

**Base:** Caso $n = 1$ ($R$ manipulando somente um elemento do multiconjunto na cláusula *REPLACE*), por definição, a quantidade de arestas de entrada extraídas de $R$ corresponde a:

$E = \{e_1\}$

O que corresponde ao seguinte código Gamma:

```
R1 = REPLACE X
BY ...
IF X == I1
```

O que faz valer o enunciado, pois como a quantidade de elementos manipulados pela cláusula *REPLACE* de $R$ é igual a 1, somente uma variável auxiliar seria utilizada por $R$ (neste caso $x$), o que daria origem a uma única aresta de entrada $e_1$ com rótulo $i_1$.

Já no caso de $R$ possuir dois elementos manipulados em sua cláusula *REPLACE* ($n = 2$), teríamos a seguinte quantidade de arestas de entrada:

$E = \{e_1, e_2\}$

Correspondendo ao seguinte código Gamma:

```
R1 = REPLACE X, Y
BY ...
IF (X == I1 OR X == I2) AND
   (Y == I1 OR Y == I2)
```

O que também vale, uma vez que, com duas variáveis auxiliares utilizadas na cláusula *REPLACE*, duas arestas de entrada seriam atribuídas a $v$.

**Hipótese Indutiva:** O enunciado vale para uma quantidade $K$ de elementos manipulados pela cláusula *REPLACE* de $R$. Desta forma teríamos o seguinte conjunto de arestas de entrada para $R$:

$E = \{e_1, e_2, ..., e_k\}$

Que seria equivalente à seguinte reação Gamma:

```
R1 = REPLACE X, Y, ..., Z
BY ...
IF (X == I1 OR X == I2 OR ... OR X == IK) AND
   (Y == I1 OR Y == I2 OR ... OR Y == IK) AND
   .
   .
   .
   (Z == I1 OR Z == I2 OR ... OR Z == IK)
```

**Passo Indutivo:** Ao inserirmos mais um elemento ($Z'$) a ser manipulado por $R$ teríamos o seguinte código correspondente:

```
R1 = REPLACE X, Y, ..., Z, Z'
BY ...
IF [(X == I1 OR X == I2 OR ... OR X == Ik) OR (X == Ik+1)] AND
   [(Y == I1 OR Y == I2 OR ... OR Y == Ik) OR (Y == Ik+1)] AND
   .
   .
   .
   [(Z == I1 OR Z == I2 OR ... OR Z == Ik) OR (Z == Ik+1)] AND
   [(Z' == I1 OR Z' == I2 OR ... OR Z' == Ik) OR (Z' == Ik+1)]
```

Conforme vimos nas provas anteriores, inserir mais um elemento a ser manipulado pela cláusula *REPLACE* de uma reação implica em podermos estender as comparações até este último elemento, no caso $Z'$, que corresponde exatamente ao elemento $k+1$. Desta forma teremos a seguinte situação:

```
R = REPLACE X,Y, ..., Z'
BY ...
IF (X == i1 OR X == i2 OR ... OR X == ik+1) AND
   (Y == i1 OR Y == i2 OR ... OR Y == ik+1) AND
   .
   .
   .
   (Z' == i1 OR Z' == i2 OR ... OR Z' == ik+1)
```

Ora, criar mais um elemento $Z'$ a ser manipulado por $R$ corresponde a incluir a aresta $e_{K+1}$ ao conjunto $E_k$ de arestas:

$E_k = \{e_1, e_2, ..., e_k\}$

$E_{k+1} = \{e_1, e_2, ..., e_k\}\ U\ \{e_{k+1}\}$

Sendo $A\ U\ B = \{x \mid x \in A\ ou\ x \in B\}$

$\{e_1, e_2, ..., e_k\}\ U\ \{e_{k+1}\} = \{e_1, e_2, ..., e_k, e_{k+1}\} = \{e_1, e_2, ..., e_{k+1}\}$

Assim, substituindo $k+1$ por $k$ temos que a hipótese de indução é verdadeira. □

**– Análise dos elementos manipulados pela cláusula *BY***

Agora queremos analisar um fragmento de um código de uma reação em Gamma a fim de extrair as arestas de saída do vértice dataflow equivalente. Desta maneira, os trechos de códigos utilizados não apresentação informações a respeito da cláusula *REPLACE*, já que esta se refere a informações sobre arestas de entrada. Portanto, nossa análise basear-se-á em análises sobre informações retiradas das cláusulas *BY* e *IF*.

**Lemma 5.** *Seja $R$ uma reação Gamma qualquer, $m$ a quantidade de elementos inseridos no multiconjunto pela cláusula $BY$ de $R$, $v$ o vértice do grafo dataflow $G$ equivalente a $R$ e $S_m$ o conjunto de arestas de saída de $R$. Definimos o grafo dataflow $G(V, E)$ onde $V$ é o conjunto de vértices e $E$ o conjunto de arestas de $G$. Definimos ainda o conjunto de arestas de saída de $v$ por: $S_m = \{s_1, s_2, ..., s_m\}$ onde cada aresta $s_x$ é definida por um par de vértices: $s_x = (v, v_x)$. O conjunto $O_m$, definido por $O_m = \{o_1, o_2, ..., o_m\}$, é o conjunto de rótulos atribuídos às arestas do conjunto $S_m$, de forma que $s_1$ tem rótulo $o_1$, $s_2$ tem rótulo $o_2$ e assim sucessivamente. Podemos extrair $S_m$ de $v$ a partir da reação $R$ conforme apresentado na Figura 3.25. Onde cada par BY / IF (BY $O_x$ .... IF CONDICAO $O_x$) dará origem a uma aresta em $S_m$.*

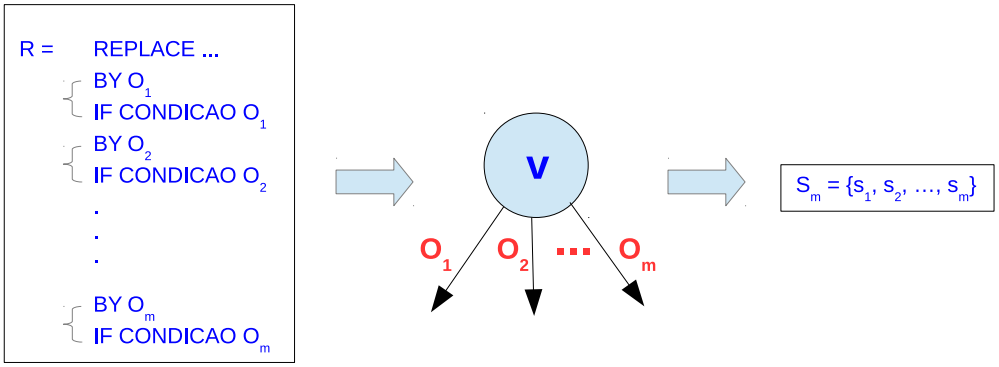


Figura 3.25: Extraindo arestas de saída de $V$ a partir de uma reação $R$.

*Demonstração.* A prova será realizada pelo princípio da indução finita nos elementos manipulados pela cláusula $BY$ de $R$.

**Base:** Caso $m = 1$ ($R$ manipulando somente um elemento do multiconjunto na cláusula $BY$), por definição, o código Gamma equivalente corresponde ao apresentado a seguir:

```
R = REPLACE ...
    BY O1
    IF CONDICAO O1
```

Correspondendo ao seguinte conjunto de arestas de saída de $R$:

$S = \{s_1\}$

O que faz valer o enunciado, pois como a quantidade de elementos manipulados pela cláusula $BY$ de $R$ é igual a 1, somente um elemento seria "devolvido" ao multiconjunto por $R$ (neste caso $o_1$), o que daria origem a uma única aresta de saída $s_1$ com rótulo $o_1$.

Já no caso de $R$ possuir dois elementos manipulados pela cláusula $BY$ ($m = 2$), teríamos o seguinte conjunto de arestas de saída:

$S = \{s_1, s_2\}$

Correspondendo ao seguinte código Gamma:

```
R = REPLACE ...
    BY O1
    IF CONDICAO O1

    BY O2
    IF CONDICAO O2
```

O que também vale, uma vez que, com dois elementos manipulados pelos pares de cláusulas $BY$ e $IF$, duas arestas de saída seriam atribuídas a $v$.

**Hipótese Indutiva:** O enunciado vale para uma quantidade $k$ de elementos manipulados pela cláusula $BY$ de $R$. Desta forma teríamos o seguinte conjunto de arestas de saída de $R$:

$S = \{s_1, s_2, ..., s_k\}$

Correspondendo ao seguinte fragmento de código Gamma:

```
R = REPLACE ...
    BY O1
    IF CONDICAO O1

    BY O2
    IF CONDICAO O2
    .
    .
    .
    BY Ok
    IF CONDICAO Ok
```

**Passo Indutivo:** Ao inserirmos mais um par de cláusulas $BY$ / $IF$ correspondendo ao k-ésimo par mais um, teríamos o código Gamma apresentado a seguir:

```
R = REPLACE ...
    BY O1
    IF CONDICAO O1
    BY O2
    IF CONDICAO O2
    .
    .
    .
```

```
    BY Ok
    IF CONDICAO Ok

    BY Ok+1
    IF CONDICAO Ok+1
```

Conforme vimos anteriormente, inserir mais um elemento a ser manipulado pela cláusula $BY$ de uma reação implica em podermos estender as comparações dos pares $BY/IF$ até este último elemento ($0_{k+1}$), resultando em:

```
R = REPLACE ...
    BY O1
    IF CONDICAO O1

    BY O2
    IF CONDICAO O2
    .
    .
    .
    BY Ok+1
    IF CONDICAO Ok+1
```

Utilizando raciocínio similar à prova anterior, criar mais um elemento a ser "devolvido" ao multiconjunto por $R$ através de sua cláusula $BY$ corresponde a incluir a aresta $s_{k+1}$ ao conjunto $S_k$ de arestas:

$S_k = \{s_1, s_2, ..., s_k\}$

$S_{k+1} = \{s_1, s_2, ..., s_k\}\ U\ \{s_{k+1}\}$

Sendo $A\ U\ B = \{x \mid x \in A\ ou\ x \in B\}$

$\{s_1, s_2, ..., s_k\}\ U\ \{s_{k+1}\} = \{s_1, s_2, ..., s_K, s_{k+1}\} = \{s_1, s_2, ..., s_{k+1}\}$

Assim, substituindo $k+1$ por $k$ temos que a hipótese de indução é verdadeira. □

#### 3.3.2.4 Considerações sobre o Não Determinismo de Gamma

Conforme mencionado anteriormente, o código Gamma extraído a partir de um grafo dataflow carrega o determinismo inerente ao Dataflow. Entretanto, o não determinismo de Gamma é uma poderosa ferramenta para a computação paralela, ao passo que uma única reação pode ser designada e, em um ambiente paralelo de execução, múltiplas instâncias desta mesma reação poderão rodar em paralelo sobre um mesmo multiconjunto. Tomemos como base um exemplo simples da escolha do menor elemento de um multiconjunto qualquer. Esta reação pode ser expressa da seguinte forma:

```
R = REPLACE X,Y
    BY X
    IF X < Y
```

Aqui, uma única reação é capaz de escolher o menor elemento de um multiconjunto qualquer, a partir da execução em paralelo de $R$. Repare que $R$ seleciona elementos dois a dois e exclui do multiconjunto o maior deles, caso o primeiro selecionado seja o menor. Note ainda que, no caso da escolha de elementos, onde o primeiro seja o maior, a reação simplesmente não reage e os elementos selecionados por esta não são alterados no multiconjunto. Pelo não determinismo de Gamma, é impossível prever a ordem pela qual os elementos do multiconjunto serão selecionados a reagir.

A Figura 3.26 apresenta possíveis instâncias da execução de $R$ sobre um multiconjunto específico. Repare que para o mesmo problema de escolha do menor elemento, dado um multiconjunto $M$ composto pelos seguintes elementos: $M = \{1, 5, 11\}$, temos algumas possibilidades de solução (instâncias de execução) dadas pelo não determinismo na seleção dos elementos que irão ser utilizados por $R$.

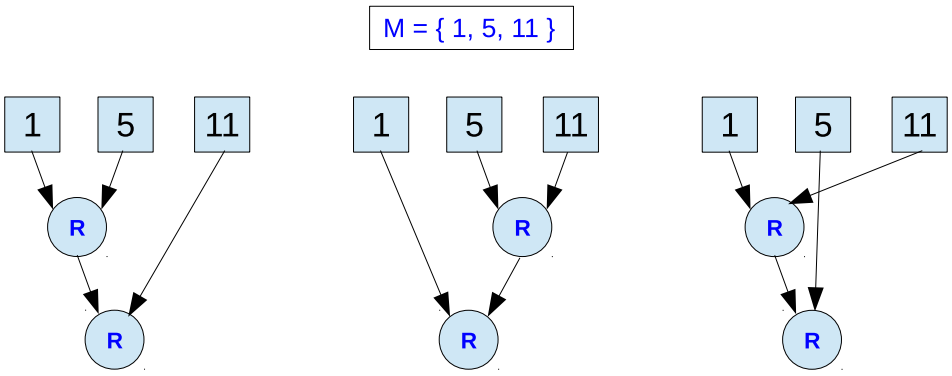


Figura 3.26: Possíveis instancias de execução da reação $R$ sob o multiconjunto M = {1, 5, 11}.

Repare que, quanto maior for o multiconjunto $M$, maiores serão as possibilidades de combinação dos elementos selecionados a reagir e, consequentemente, maior será a quantidade de instâncias de execução (grafos correspondentes) da solução. Note ainda que, no exemplo da Figura 3.26, temos somente uma única reação $R$.

Entretanto, como mostramos anteriormente, sempre será possível extrair vértices de um grafo dataflow e suas respectivas arestas de entrada e saída, a partir de uma reação em Gamma. Desta forma, ao transformar cada reação em um vértice, teremos um subgrafo da solução, que deverá ser replicado para cobrir todo o multiconjunto. Em outras palavras, tal subgrafo da solução deverá ser aplicado a todos os elementos, de uma maneira eficiente e que garanta a manipulação de todos os elementos

do multiconjunto. Esta tarefa corresponde à característica de não determinismo do modelo Gamma e deverá ser desempenhada por um componente chamado de escalonador, responsável pela seleção de todos os elementos do multiconjunto.

Desta forma, podemos afirmar que, ao convertermos cada reação de um código Gamma em um vértice dataflow, teremos um subgrafo da solução que, ao ser replicado de forma a cobrir todo o multiconjunto, irá fornecer um grafo dataflow equivalente, que será uma das possíveis instâncias de solução existente.

Outra possível solução para lidar com o não determinismo de Gamma consiste na introdução de vértices especiais (e/ou arestas) de maneira a fornecer recursos não determinísticos. Por exemplo, poderíamos propor um vértice especial que seleciona aleatoriamente um elemento no multiconjunto. Este vértice seria responsável por combinar todos os elementos do multiconjunto e fornecer dados para o grafo dataflow. Neste sentido, dado um subgrafo produzido pela transformação de um código Gamma em um grafo dataflow, nesta abordagem, não seria necessário replicar este subgrafo para cobrir todo o multiconjunto. Assim, tal subgrafo teria alguns vértices e arestas especiais responsáveis por fornecer elementos aleatoriamente, visando cobrir todo o multiconjunto.

## 3.4 Discussões

Gamma e Dataflow apresentam uma similaridade que foi inicialmente proposta em [23]. Na ocasião da publicação deste trabalho em referência, exercitamos a similaridade através de exemplos onde pudemos extrair algoritmos de conversão e os detalhes implícitos inerentes à equivalência de um vértice de um grafo dataflow com uma reação segundo o paradigma Gamma e vice e versa.

Dessa maneira, um vértice de um grafo dataflow pode ser convertido em uma reação Gamma, onde as arestas de entrada deste vértice guardam uma correspondência com os elementos manipulados pela cláusula *REPLACE* da reação Gamma equivalente, assim como as arestas de saída do referido vértice estão relacionadas à cláusula *BY*.

Ainda mencionando o estudo publicado em [23], foram apresentadas considerações à respeito de reduções, apresentadas aqui na seção 3.1.3, que mostram a possibilidade de alteração de granularidade tanto das reações Gamma quanto da operação expressa por um vértice de um grafo dataflow. Este estudo é particularmente interessante pelo potencial de Gamma a ser utilizado em ambientes distribuídos, o que será discutido nos capítulos seguintes.

Em [24] foi publicado uma extensão do trabalho apresentado em [23], onde foi apresentada a prova formal de equivalência entre os modelos computacionais Gamma e dataflow, incluindo algumas considerações sobre o não determinismo do modelo

Gamma, conforme apresentado aqui na seção 3.3.

O estudo da equivalência entre os modelos citados contribui para a versatilidade no desenvolvimento de programas paralelos, uma vez que oferece ao desenvolvedor a escolha de programar entre dois diferentes paradigmas computacionais que possuem mecanismos naturais e poderosos para explorar o paralelismo de aplicações. Da mesma forma o estudo permite explorar benefícios mútuos entre os paradigmas computacionais. Em outras palavras, um programa escrito em Gamma poderá se beneficiar de estudos já desenvolvidos para o modelo dataflow e vice e versa, como por exemplo, os estudos apresentados em [7] e [8].

# Capítulo 4

# Gamma - Explorando Benefícios

## 4.1 Introdução do Capítulo

Conforme apresentado no capítulo anterior, o estudo de equivalência entre os modelos computacionais Gamma e Dataflow proporcionou a oportunidade de explorar para um modelo computacional, os benefícios desenvolvidos para outro paradigma. Assim sendo, através do uso da equivalência, um programa Gamma poderá se beneficiar de estudos abordando execução especulativa e fora de ordem no contexto de um grafo dataflow [7], reuso aplicado a traços de instruções dataflow [8], além de outros trabalhos desenvolvidos.

Na mesma linha de raciocínio, programas desenvolvidos segundo o modelo Dataflow poderiam se beneficiar de Gamma. Este último possui potencial para ser utilizado com técnicas de computação aproximativa [5], conforme apresentado no Apêndice B, além de possuir um potencial de paralelismo inerente ao modelo, conforme será discutido durante o decorrer deste trabalho.

Desta forma, contribuindo para a expressividade do modelo Gamma, propusemos um modelo, que será descrito no presente Capítulo, onde fornecemos uma ferramenta de conversão tanto para um programa escrito em linguagem C, quanto para um programa escrito em dataflow, para seu correspondente código Gamma. Trata-se da primeira proposta de ferramenta computacional para realizar a conversão mencionada.

Além disso, frente à análise das implementações de Gamma existentes, foi implementado um novo ambiente de execução de programas Gamma, baseado em [22], que utiliza mapeamento ótimo de sistemas com restrições sobre a vizinhança, proposto em [63]. Trata-se da primeira implementação de um ambiente de execução para programas Gamma que permite a execução paralela de instâncias de reações, conforme veremos na seção 4.3.

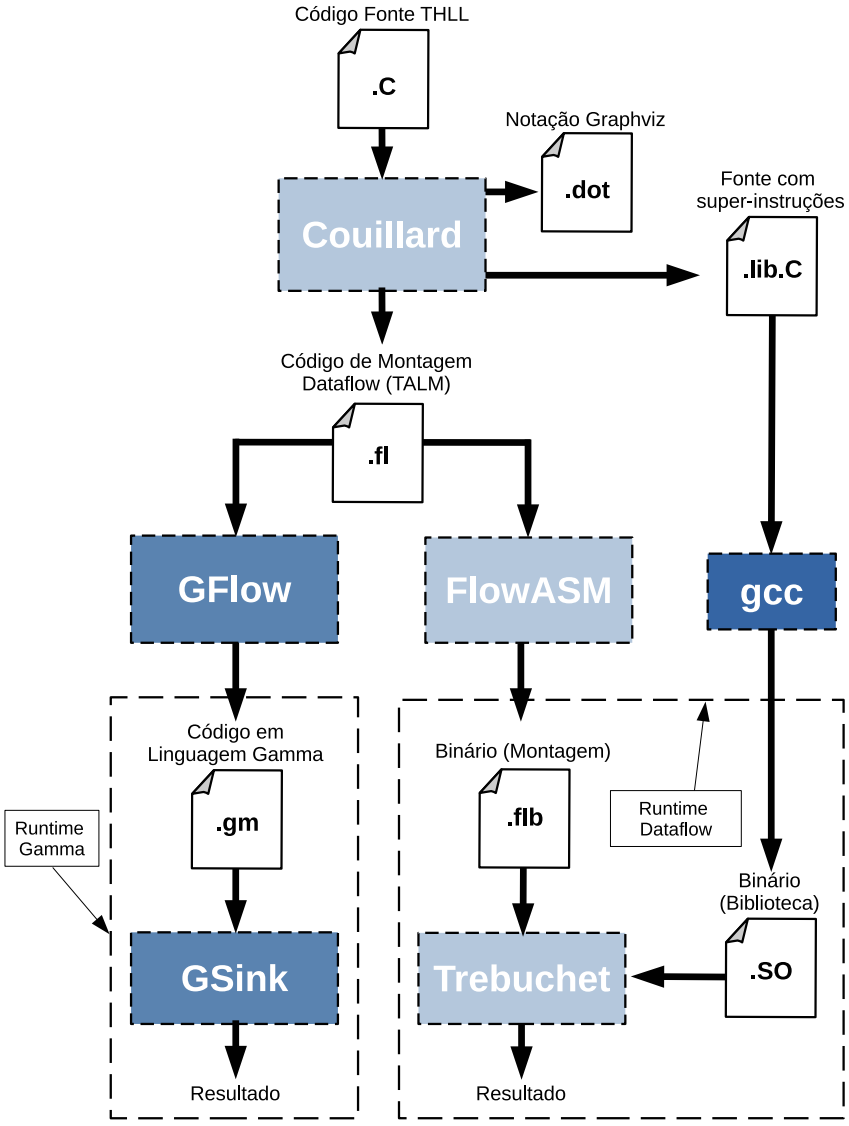


Figura 4.1: Workflow das ferramentas Gamma e dataflow.

A Figura 4.1 apresenta uma descrição sucinta e generalista de um fluxo de trabalho contendo as ferramentas propostas (***GFlow*** e ***GSink***) e suas interações com outras ferramentas previamente existentes e propostas em [64] e [61]. Nas seções subsequentes os processos e ferramentas envolvidas serão descritas em maiores detalhes. A partir de um código C anotado conforme a ***THLL*** ( *TALM* High-Level Language [61]), utiliza-se o ***Couillard*** [65], um compilador dataflow, para gerar o *assembly* do ***TALM*** (*TALM* é uma arquitetura e linguagem para *Multithreading* [61]), além de outros arquivos. Desta forma, o ***Couillard*** gera, um grafo dataflow (expresso em um aquivo **".fl"**) que corresponde à linguagem de montagem do ***TALM***. Caso seja desejável a execução do programa, inicialmente expresso em ***THLL***, em um ambiente de execução baseado em dataflow, um montador (***FlowASM***) é utilizado para gerar código binário para a ***Trebuchet*** (uma máquina virtual que utiliza a arquitetura de von Neuman para executar programas dataflow).

Entretanto, nossa proposta, o ***GFlow***, permite a conversão de programas escritos em linguagem C anotada, segundo o ***THLL***, em programas expressos pelo paradigma Gamma, através da implementação da conversão entre os modelos computacionais dataflow e Gamma, apresentada no Capítulo 3. Assim, o ***GFlow*** converte um grafo dataflow em um código Gamma expresso conforme a sintaxe proposta por Juarez Muylaert e descrita por DE MELLO JUNIOR [18] e DE ALMEIDA [30]. Por fim, conforme mencionado anteriormente, implementamos o ***GSink***, um novo ambiente para execução de programas Gamma que também será descrito no presente Capítulo.

## 4.2 GFlow

A presente seção pretende apresentar o *GFlow*, uma ferramenta de conversão dataflow - Gamma. Como utilizamos conhecimentos de projetos prévios desenvolvidos por ocasião da proposta do *TALM*, inicialmente apresentaremos informações a respeito destes estudos. Assim, iniciaremos a presente seção pela apresentação do *TALM*, passando por sua arquitetura, conjunto de instruções, sua linguagem de alto nível (*THLL*), o compilador *Couillard*, o montador *FlowASM*, finalizando com a máquina virtual *Trebuchet*. Após isso, apresentaremos o *GFlow* onde abordaremos sua descrição, benefícios e contribuições além de aspectos gerais sobre a implementação da ferramenta. No Capítulo 5 serão apresentadas uma série de experimentos visando demonstrar a corretude da conversão e implementação do conjunto de instruções do *TALM*.

### 4.2.1 TALM

A arquitetura de von Neumann é amplamente utilizada pelos processadores atuais. Em um esforço para explorar o paralelismo, várias técnicas são utilizadas, a exemplo de execução fora de ordem (baseada no algoritmo de Tomasulo [66]), onde o disparo de instruções continua sendo realizado de maneira sequencial, como preconiza o modelo sequencial vigente.

Nesse contexto, visando tirar proveito das oportunidades do modelo dataflow, sem perder a compatibilidade com arquiteturas sequenciais atuais, surge o *TALM* (*TALM is an Architecture and Language for Multi-threading*) [14]. O modelo propõe uma arquitetura base para um ambiente de execução baseado em fluxo de dados, com instruções de granularidade customizável, uma vez que permite a definição de super-instruções pelo desenvolvedor. O modelo foi idealizado para ser implementado como um sistema de execução que crie uma abstração de uma arquitetura dataflow sobre uma máquina von Neumann.

#### 4.2.1.1 Arquitetura do TALM

Conforme ilustrado na Figura 4.2, a arquitetura do *TALM* é composta de diversos Elementos de Processamento (EP) idênticos, interconectados por uma rede de comunicação. Estes são os elementos responsáveis pela execução de cada instrução neste modelo. Dessa forma, para a execução do programa dataflow expresso pelo *TALM*, o passo inicial é mapear as instruções entre os EP disponíveis.

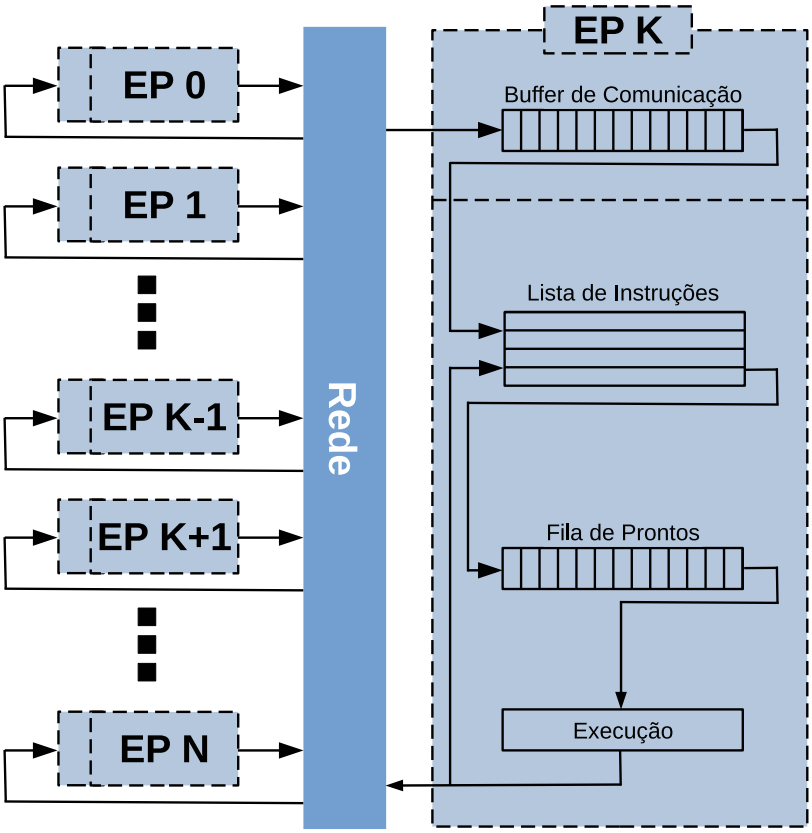


Figura 4.2: Arquitetura do TALM (Baseado em [61]).

Por ocasião da execução da aplicação, um EP inicialmente verifica em um *buffer* de comunicação, o recebimento de mensagens de outros EPs contendo operandos. Cada EP mantém uma lista de operandos para instâncias dinâmicas de instruções, de forma que quando todos os operandos de mesmo rótulo de iteração estiverem disponíveis, esta instância de instrução pode ser enviada à lista de instâncias prontas (Fila de Prontos) a fim de serem enviadas à execução. Maiores informações sobre rótulos de iteração de operandos podem ser verificados no Capítulo 2, Seção 2.2.

Máquinas dataflow não utilizam *Program Counter*. Assim, tendo em vista determinar o final da execução do programa, no modelo *TALM*, utilizou-se um algoritmo distribuído para detecção de terminação global da computação [64], [61].

#### 4.2.1.2 Conjunto de Instruções

O formato de instruções proposto pelo *TALM* está ilustrado na Figura 4.3. Os 32 primeiros *bits* são obrigatórios e dizem respeito ao código da Operação (*Opcode* - 22 *bits*), quantidade de operandos de entrada (#Origens) e quantidade de operandos de saída (#Resultados). Estes dois últimos campos, conforme a quantidade de *bits* (5) apresentados na figura (entre parênteses), suporta uma quantidade total de 32 operandos cada. O Campo *Imediato*, é opcional e, conforme a própria nomenclatura sugere, é utilizado somente por instruções com imediato. O modelo suporta ainda a execução de instruções com operandos vindos de origens distintas. Dessa forma, um operando é tido como disponível assim que a porta de entrada for preenchida por qualquer uma de suas possíveis origens. Por isso, um campo de 32 *bits* é utilizado visando indicar a quantidade de origens para cada porta de entrada. Para cada possível origem, 32 *bits* são utilizados para indicar o endereço da instrução, onde 27 *bits* indicam seu número e 5 *bits* indicam a porta de saída desta instrução.

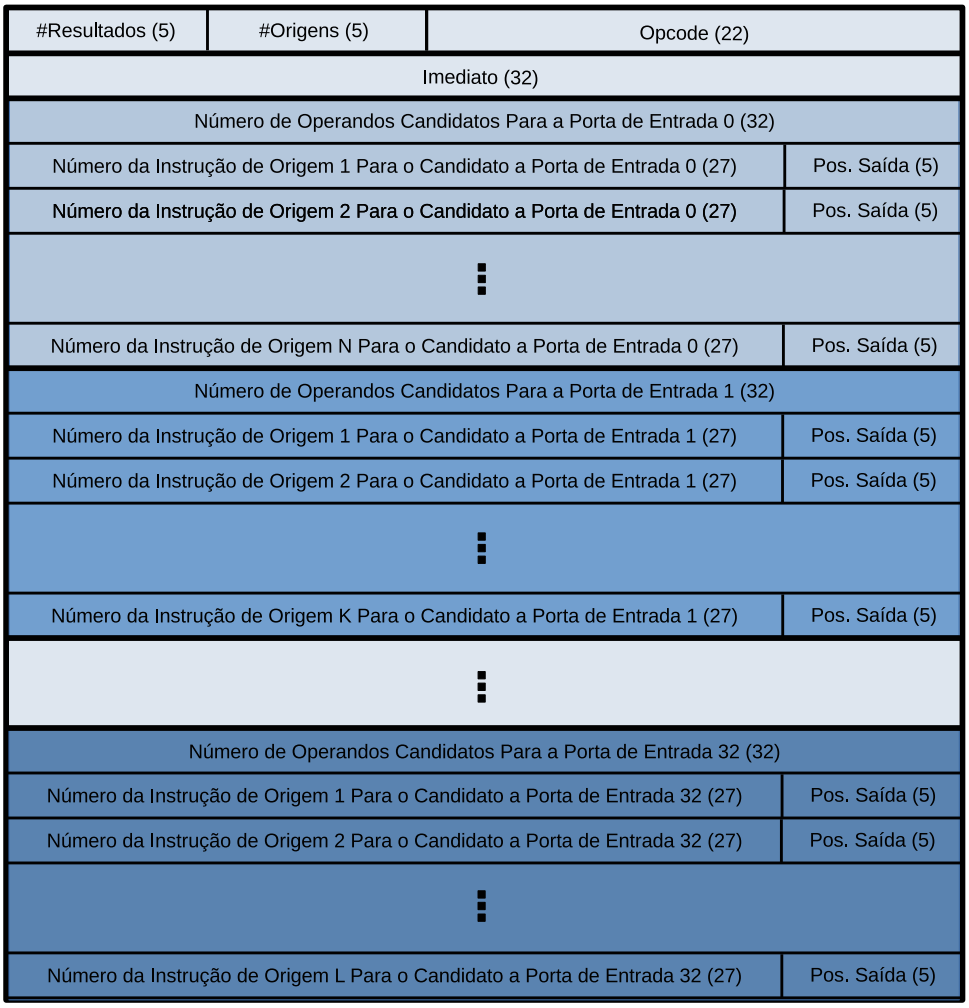


Figura 4.3: Formato das Instruções do TALM (Baseado em [61]).

A linguagem de montagem do *TALM* fornece um conjunto de instruções nativas que estão listadas na Tabela 4.1. Neste conjunto básico de instruções encontram-se instruções lógicas e aritméticas, suas variações pela utilização do operando imediato, instruções responsáveis pelo controle de desvios (*Steer*), instruções usadas no controle de laços (*Inctag*) além de instruções utilizadas no controle de chamadas de funções e definição de super-instruções. Uma descrição da utilização de instruções do tipo *Steer* e *Inctag* pode ser encontrada no Capítulo 2.

Ainda à respeito da linguagem de montagem do *TALM*, a sintaxe das instruções lógicas e aritméticas, sem utilização de operando imediato é dada por:

```
<mnemônico> <nome>, <op1>, <op2>
```

Onde ***mnemônico*** indica a operação a ser realizada, ***nome*** refere-se à identificação da instância da instrução, usada para referenciar o operando de saída. Os campos ***op1*** e ***op2*** indicam os operandos da instrução.

A sintaxe das instruções lógicas e aritméticas com utilização do operando imediato difere somente pelo fato do segundo operando possuir operando imediato:

```
<mnemônico> <nome>, <op>, <imediato>
```

Conforme mencionamos anteriormente, uma porta de entrada pode receber um operando proveniente de diversas origens. Para descrever na linguagem de montagem estes múltiplos candidatos para um mesmo operando, é utilizada a sintaxe a seguir:

```
[Candidato_0, Candidato_1, ..., Candidato_N]
```

Já uma instrução do tipo *Steer* possui a seguinte sintaxe:

```
steer <nome>, <seletor>, <valor>
```

Onde o campo ***valor*** indica o operando com o valor que será encaminhado para uma das duas saídas existentes na referida instrução: *<**nome**>**.t***, caso ***seletor*** possuir valor *TRUE* ou *<**nome**>**.f***, caso contrário.

Uma instrução do tipo *Inctag* incrementa o rótulo de iteração do operando de entrada descrito no campo *operando* e apresenta a seguinte sintaxe na linguagem de montagem:

```
inctag <nome>, <operando>
```

Em [61], encontra-se uma descrição detalhada de cada instrução implementada na linguagem de montagem do TALM, incluindo instruções utilizadas no controle de chamadas de funções e instruções para definição de super-instruções.

Tabela 4.1: Resumo do Conjunto de Instruções do TALM (Baseado em [61]).

| **Mnemônico** | **Função** | **#In** | **#Out** | **Imm** | **Tipo** |
|---|---|---|---|---|---|
| add | + | 2 | 1 | Não | Int |
| sub | - | 2 | 1 | Não | Int |
| div | / | 2 | 1 | Não | Int |
| mult | * | 2 | 1 | Não | Int |
| mod | resto da divisão | 2 | 1 | Não | Int |
| and | E lógico | 2 | 1 | Não | Int |
| or | OU lógico | 2 | 1 | Não | Int |
| lthan | < | 2 | 1 | Não | Int |
| gthan | > | 2 | 1 | Não | Int |
| leq | <= | 2 | 1 | Não | Int |
| geq | >= | 2 | 1 | Não | Int |
| addi | + | 2 | 1 | Sim | Int |
| subi | - | 2 | 1 | Sim | Int |
| divi | / | 2 | 1 | Sim | Int |
| multi | * | 2 | 1 | Sim | Int |
| modi | resto da divisão | 2 | 1 | Sim | Int |
| andi | E lógico | 2 | 1 | Sim | Int |
| ori | OU lógico | 2 | 1 | Sim | Int |
| lthani | < | 2 | 1 | Sim | Int |
| gthani | > | 2 | 1 | Sim | Int |
| leqi | <= | 2 | 1 | Sim | Int |
| geqi | >= | 2 | 1 | Sim | Int |
| fadd | + | 2 | 1 | Não | Float |
| fsub | - | 2 | 1 | Não | Float |
| fdiv | / | 2 | 1 | Não | Float |
| fmult | * | 2 | 1 | Não | Float |
| fmod | resto da divisão | 2 | 1 | Não | Float |
| fand | E lógico | 2 | 1 | Não | Float |
| for | OU lógico | 2 | 1 | Não | Float |
| flthan | < | 2 | 1 | Não | Float |
| fgthan | > | 2 | 1 | Não | Float |
| fleq | <= | 2 | 1 | Não | Float |
| fgeq | >= | 2 | 1 | Não | Float |
| faddi | + | 2 | 1 | Sim | Float |

Continua na próxima página.

Continuação da página anterior.

| Mnemônico | Função | #In | #Out | Imm | Tipo |
|---|---|---|---|---|---|
| fsubi | - | 2 | 1 | Sim | Float |
| fdivi | / | 2 | 1 | Sim | Float |
| fmulti | * | 2 | 1 | Sim | Float |
| fmodi | resto da divisão | 2 | 1 | Sim | Float |
| fandi | E lógico | 2 | 1 | Sim | Float |
| fori | OU lógico | 2 | 1 | Sim | Float |
| flthani | < | 2 | 1 | Sim | Float |
| fgthani | > | 2 | 1 | Sim | Float |
| fleqi | <= | 2 | 1 | Sim | Float |
| fgeqi | >= | 2 | 1 | Sim | Float |
| inctag | Incrementa o rótulo de iteração | 1 | 1 | Não | N/A |
| steer | desvio | 2 | 1 | Não | N/A |
| callsnd | chamada de função | 2 | 1 | Sim | N/A |
| retsnd | chamada de função | 2 | 1 | Sim | N/A |
| ret | retorno de função | 2 | 1 | Não | N/A |
| super | definida pelo usuário | até 32 | até 32 | Não | N/A |
| superi | definida pelo usuário | até 32 | até 32 | Sim | N/A |
| specsuper | super com especulação | até 32 | até 32 | Não | N/A |
| specsuperi | superi com especulação | até 32 | até 32 | Sim | N/A |

Tabela 4.1 Resumo do Conjunto de Instruções do TALM (Baseado em [61]).

#### 4.2.1.3 THLL

Tendo em vista facilitar a criação de programas paralelos destinados ao *TALM*, foi proposta uma linguagem de alto nível, que estende a linguagem C: o *THLL (TALM High Level Language)* [61]. Desta maneira, o *THLL* é compilado para a linguagem de montagem do *TALM* através do *Couillard*, que será abordado a seguir.

O *THLL* fornece mecanismos para definição de super-instruções (definidas entre as diretivas #**BEGINSUPER** e #**ENDSUPER**, conforme apresentado no Código 4.1), trechos de códigos que descrevem o comportamento de alguma tarefa e que são executados no modelo de von Neumann. O desenvolvedor descreve as super-instruções e seus argumentos de entrada e saída. Tais construções, as super-instruções, possuem parâmetros de configuração que permitem a criação de somente uma ou múltiplas instâncias no grafo dataflow. A definição da granularidade das super-instruções é fundamental para a eficiência de um programa paralelo

implementado através do *TALM*, uma vez que permite balancear o custo-benefício entre o *overhead* de comunicação e a exploração do paralelismo da aplicação.

```
super parallel input(b::mytid) output (c)
#BEGINSUPER //Processing Code
    int i;
    int tid = treb_get_tid();
    int n_tasks = treb_get_n_tasks();
    int task_size = size / n_tasks;
    int begin = tid * task_size;
    int end = begin + task_size;
    for(i=begin; i<end; i++){
        C[i] = A[i] + B[i];
    }
#ENDSUPER
```

Código 4.1: Exemplo de sintaxe de uma super-instrução no TALM [61].

Além das super-instruções, a linguagem também fornece a possibilidade de definir blocos de códigos que não serão compilados pelo *Couillard*, através do par de anotações #**BEGINBLOCK** e #**ENDBLOCK** (ilustrado pelo Código 4.2). Tais anotações normalmente são utilizadas para reunir o *Header* (arquivos e bibliotecas incluídos no código), definições de funções auxiliares e declarações de variáveis globais.

```
#BEGINBLOCK     //Includes, Functions and Globals
    #include <stdio.h>
    #include <stdlib.h>
    #define size 10000
    FILE * fa;
    FILE * fb;
    FILE * fc;
    int A[size];
    int B[size];
    int C[size];
#ENDBLOCK
```

Código 4.2: Exemplo de sintaxe de um bloco que não será compilado para o TALM [61].

Por fim, o controle do programa, assim como estruturas de laços de repetição e execução condicional podem ser descritos normalmente através da linguagem C.

#### 4.2.1.4 Couillard

Uma vez tendo desenvolvido uma aplicação em *THLL*, a mesma precisa ser compilada para ser executada na *Trebuchet*. Assim, os autores do *TALM* implementaram o *Couillard*, um compilador responsável por transformar o código de alto nível descrito em *THLL* em um grafo dataflow descrito pela linguagem de montagem do *TALM*, além de uma biblioteca contendo a descrição das super-instruções.

O *Couillard* foi implementado em *Python* e produz um código em linguagem C correspondente à cada super-instrução, do código *THLL* origem, que será posteriormente compilado como um objeto compartilhado (em uma biblioteca dinamicamente vinculada) para a arquitetura alvo, carregada pela *Trebuchet*. Todo código que não estiver definido no escopo das super-instruções será compilado diretamente para a linguagem de montagem do *TALM*, de forma a representar as instruções descritas na Tabela 4.1.

O *Front-End* do compilador *Couillard* realiza as funções de análise léxica e sintática utilizando através da utilização da biblioteca PLY(Python Lex-Yacc) [67], produzindo uma AST (Auxiliary Sintax Tree) que será utilizada para gerar uma representação na forma de um grafo dataflow.

Já o *Back-End* do compilador desempenha três funções distintas. Ele é responsável por gerar a linguagem de montagem do *TALM*, que será expressa em um arquivo com extensão **".fl"** que, conforme mencionado anteriormente, corresponde ao código da aplicação *THLL* originária que não encontra-se definido dentro de blocos de super-instruções. A segunda atribuição do *Back-End* é gerar o código C correspondente às super-instruções que serão compiladas por um compilador C padrão como uma biblioteca de ligação dinâmica. Este código referente às super-instruções estará descrito em um arquivo de extensão **".lib.c"**. Por fim, a última atribuição do compilador é gerar um arquivo com extensão **".dot"**, contendo uma descrição do grafo em notação Graphviz [68], fornecendo uma visão mais clara do grafo gerado, para ser utilizado para fins acadêmicos.

#### 4.2.1.5 FlowASM

Conforme mencionado anteriormente, o compilador *Couillard* gera um arquivo em notação Graphviz, um código C, referente às super-instruções e um arquivo contendo instruções em linguagem de montagem, referente a todo e qualquer código do arquivo C inicial, que não estiver anotado como *blocos* ou *super-instruções* pela *THLL*. O Código 4.3 representa um código simples, em linguagem C, somente com anotações de blocos e código fonte C, segundo a *THLL*, sem nenhuma definição de super-instrução.

```
#BEGINBLOCK
   #include <stdio.h>
#ENDBLOCK

int main() {
  int i, a, n, resultado;
  i = 0;
  a = 2;
  n = 5;
    resultado = 0;
  while (i<n) {
    i = i + 1;
    a = a + i;
        resultado = i * a;
  }
}
```

Código 4.3: Exemplo de um Código C, com anotação de bloco pela *THLL*.

Já o Código 4.4 abaixo, apresenta a lista de instruções, em linguagem de montagem do *TALM*, convertido pelo *Couillard*:

```
1  const flowInstConst139658106973936, 0
2  const flowInstConst139658106971704, 2
3  const flowInstConst139658106971128, 5
4  const flowInstConst139658106971488, 0
5  inctag flowInstIncTag139658107024664, [flowInstConst139658106973936,
       flowInstBinopI139658107024592]
6  inctag flowInstIncTag139658107023512, [flowInstConst139658106971128,
       flowInstSteer139658106577696.t]
7  lthan flowInstBinop139658106973792, flowInstIncTag139658107024664,
       flowInstIncTag139658107023512
8  steer flowInstSteer139658107025240, flowInstBinop139658106973792,
       flowInstIncTag139658107024664
9  addi flowInstBinopI139658107024592, flowInstSteer139658107025240.t, 1
10 inctag flowInstIncTag139658107025672, [flowInstConst139658106971704,
       flowInstBinop139658107025384]
11 steer flowInstSteer139658107025888, flowInstBinop139658106973792,
       flowInstIncTag139658107025672
12 add flowInstBinop139658107025384, flowInstSteer139658107025888.t,
       flowInstBinopI139658107024592
13 mult flowInstBinop139658107026320, flowInstBinopI139658107024592,
       flowInstBinop139658107025384
14 inctag flowInstIncTag139658107026824, [flowInstConst139658106971488,
       flowInstBinop139658107026320]
15 steer flowInstSteer139658107026968, flowInstBinop139658106973792,
       flowInstIncTag139658107026824
16 steer flowInstSteer139658106577696, flowInstBinop139658106973792,
       flowInstIncTag139658107023512
```

Código 4.4: Código em linguagem de montagem, gerado através do Código 4.3.

Assim, o *FlowASM* é a implementação de um montador para o *TALM*, escrito em linguagem Python, que é responsável por converter um arquivo em linguagem de montagem (arquivo extensão ".fl", conforme exemplo apresentado no Código 4.4) em código binário a ser executado pela *Trebuchet*. O *FlowASM* identifica e trata expressões regulares da linguagem de montagem do *TALM*, identificando constantes, nome de operandos, nome de instruções e operandos de funções. Estes componentes da gramática do *TALM* serão convertidos para o respectivo código binário e fornecidos futuramente à *Trebuchet* através de um arquivo com extensão **".flb"**.

#### 4.2.1.6 Trebuchet

A máquina virtual, que implementa o *TALM*, fornecida pelos mesmos autores é denominada *Trebuchet*. Ela é responsável pela execução da aplicação inicialmente expressa segundo a sintaxe da *THLL*. O processo completo, para submissão de uma aplicação sequencial para o *TALM* está ilustrado na Figura 4.4. Primeiramente, o desenvolvedor deve realizar anotações de super-instruções e blocos, no código C inicial, de acordo com a sintaxe da *THLL*. Este código é compilado pelo *Couillard*, que fornecerá a linguagem de montagem (arquivo extensão **".fl"**), o fonte com as super-instruções (**".lib.c"**) além do arquivo em notação Graphviz (**".dot"** - não apresentado na Figura 4.4). O fonte contendo as super-instruções é compilado por um compilador C padrão (gcc), como uma biblioteca de ligação dinâmica. Já a linguagem de montagem gerada pelo *Couillard* será submetida ao *FlowASM* que irá gerar o código binário (arquivo com extensão **".flb"**) a ser executado pela *Trebuchet*. Dessa forma, a *Trebuchet* recebe como entradas: o código binário correspondente ao arquivo expresso em linguagem de montagem, a biblioteca dinâmica referente ao código das super-instruções e um arquivo de alocação de instruções.

O modelo *TALM* baseia-se num esquema de troca de mensagens para comunicação entre os diversos Elementos de Processamento (EP) existentes. Assim, a comunicação para troca de operandos entre instruções que pertençam a um mesmo EP ocorre através do acesso direto às estruturas destes operandos. Por outro lado, quando instruções pertençam a EPs distintos, a troca de mensagens é realizada através da implementação de rotinas que dependem da arquitetura da máquina alvo.

Em sua implementação atual, a *Trebuchet* é fornecida para máquinas multicore (*CMPs - Chip Multiprocessors*) com memória compartilhada entre os núcleos de processamento. Dessa forma, seus elementos de processamento são implementados através de *threads* que são mapeadas para processadores reais. Assim, a troca de mensagens foi implementada através de regiões de memória compartilhada.

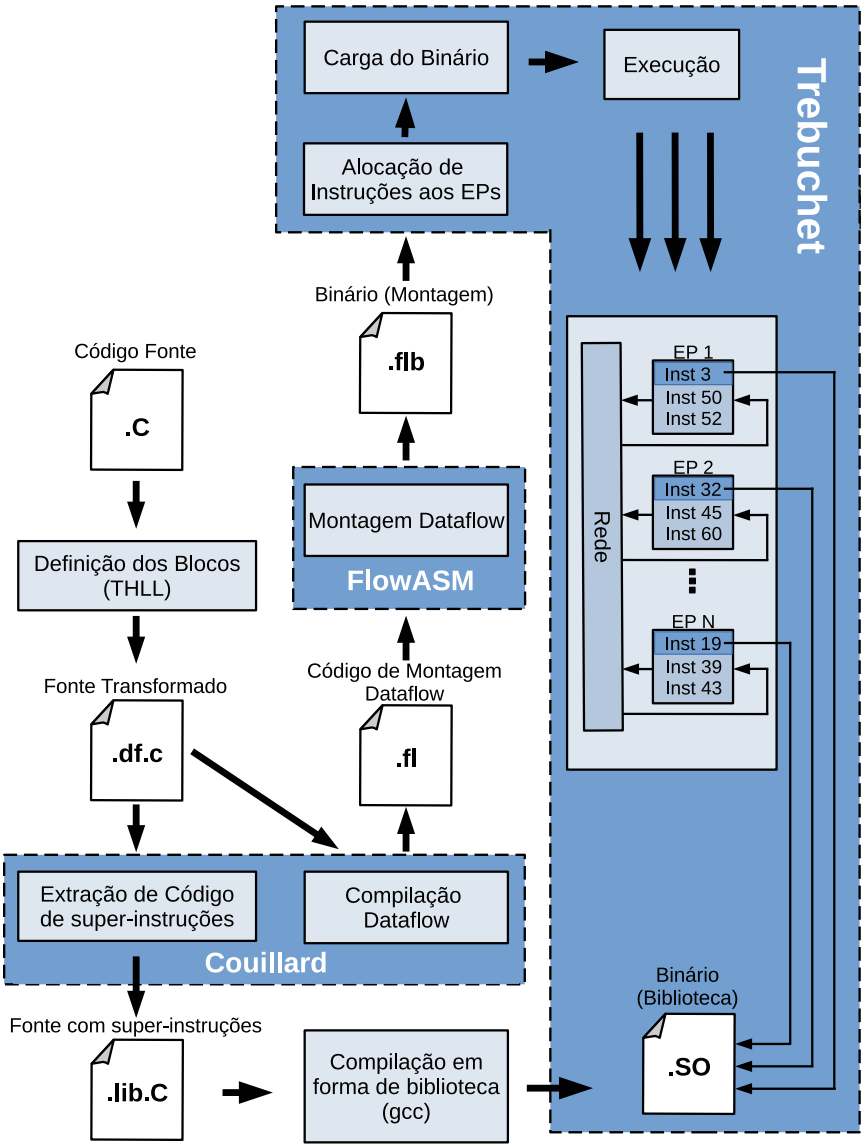


Figura 4.4: Fluxo de trabalho da Trebuchet (Baseado em [61]).

### 4.2.2 GFlow: Um Conversor Dataflow - Gamma

O estudo da equivalência descrita no Capítulo 3, demonstrou a exequibilidade da conversão entre os modelos computacionais envolvidos. Isso é particularmente interessante ao passo que permite estender as contribuições já realizadas para ambos modelos. Ou seja, um código Gamma poderia se beneficiar de diversos estudos já propostos para o modelo de fluxo de dados. Entretanto, também faz sentido pensar que o modelo dataflow poderia se beneficiar da conversão para Gamma, uma vez que este último tem potencial para ser utilizado com técnicas de computação aproximativa (Apêndice B), em domínios de aplicação específicos como processamento de imagens [27, 32, 33], fusão de dados aplicados ao meio militar naval [17], além de ser adequado para um ambiente de computação heterogêneo, como o exemplo da *Fluid Computing* [5].

Assim, após termos abordado os conceitos inerentes a proposta do *TALM* e detalhes de sua implementação, chega o momento de explicitarmos a nossa proposta de implementação da equivalência dataflow - Gamma: o *GFlow*.

#### 4.2.2.1 Descrição

O *GFlow* consiste na primeira implementação da equivalência entre os modelos computacionais dataflow e Gamma. A Figura 4.5 descreve de maneira genérica o contexto e as entradas e saídas do *GFlow*.

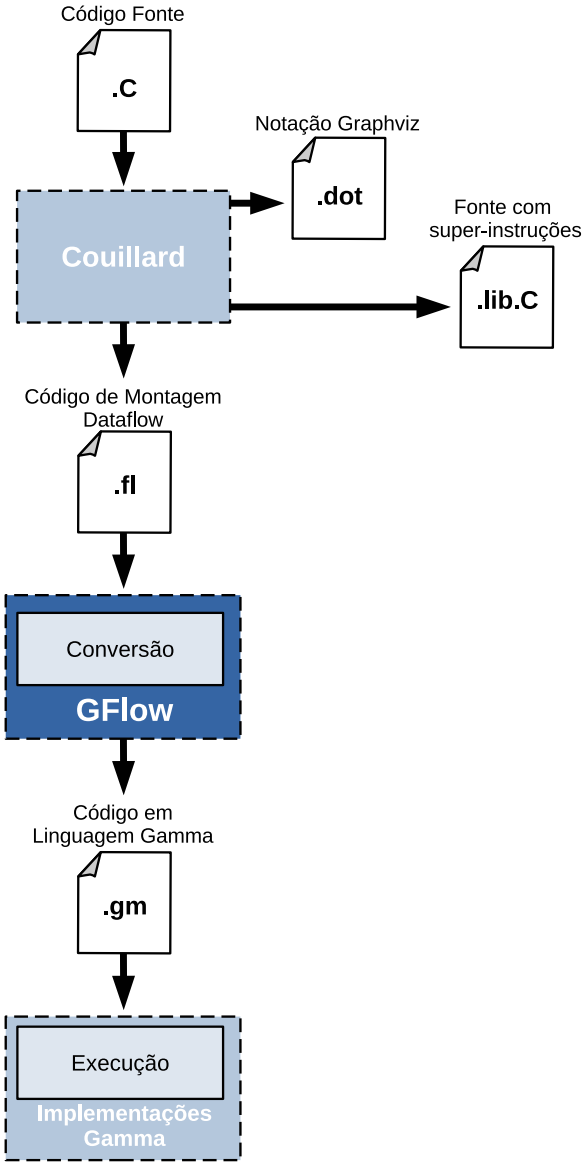


Figura 4.5: GFlow - Visão Geral.

O *GFlow* recebe como entrada um arquivo expresso em linguagem de montagem do TALM, cuja extensão é **".fl"**, conforme descrito na sub-seção 4.2.1.2. Assim, neste

arquivo encontram-se instruções lógicas, aritméticas, suas variações pela utilização de operando imediato, instruções utilizadas para controle de desvios (*Steer*), instruções usadas para controle de laços (*Inctag*) e funções utilizadas para chamadas de funções e super-instruções. Uma lista contendo todas as instruções suportadas pela linguagem de montagem do *TALM* está disponibilizada na Tabela 4.1. É importante notar que esta linguagem de montagem do *TALM* corresponde a um grafo dataflow, que no caso específico da Figura 4.5, expressa o grafo dataflow correspondente ao arquivo C de entrada.

Tendo em vista o preconizado por ocasião do estudo da equivalência (Capítulo 3), o *GFlow* converte instruções da linguagem de montagem do *TALM* para reações Gamma. Para isso são considerados os elementos que irão compor as cláusulas existentes na reação Gamma (Cláusulas *REPLACE*, *IF* e *BY*) e o multiconjunto inicial, que será criado mediante informações das constantes definidas na linguagem de montagem do *TALM*. Dessa maneira, para cada instrução que conste do arquivo ".fl" de entrada, será criada a reação (ou reações) equivalente(s) no código Gamma.

Após o procedimento de conversão, um arquivo de extensão **".gm"** é fornecido, correspondendo ao código Gamma equivalente ao arquivo **".fl"** de entrada. Tal arquivo, que expressa o código Gamma, possui a sintaxe Gamma fornecida por Juarez Muylaert [18, 30].

Confome mencionamos anteriormente, por ocasião da descrição do *TALM*, o arquivo que contém a linguagem de montagem é gerado através da utilização do *Couillard*, que gera também um arquivo para visualização do grafo dataflow em notação *Graphviz* e um arquivo contendo código C correspondente ao código das super-instruções. Estes dois últimos arquivos, não são utilizados pelo *GFlow*.

Maiores informações a respeito de detalhes da conversão entre instruções TALM (que correspondem a um grafo dataflow) para reações Gamma, além de informações sobre a implementação do *GFlow* serão fornecidas no decorrer da presente seção.

#### 4.2.2.2 Conjunto de Instruções do TALM implementados no GFlow

O *GFlow* implementa a conversão de todas as instruções da linguagem de montagem do *TALM* descritas na Tabela 4.1, com exceção das instruções que constam na Tabela 4.2. Isso deve-se ao fato de que estas instruções correspondem àquelas utilizadas em chamadas de funções e super-instruções.

Tabela 4.2: Instruções do *TALM* não convertidas pelo *GFlow*.

| **Mnemônico** | **Função** | **#In** | **#Out** | **Imm** | **Tipo** |
|---|---|---|---|---|---|
| callsnd | chamada de função | 2 | 1 | Sim | N/A |
| retsnd | chamada de função | 2 | 1 | Sim | N/A |

| Mnemônico | Função | #In | #Out | Imm | Tipo |
|---|---|---|---|---|---|
| ret | retorno de função | 2 | 1 | Não | N/A |
| super | definida pelo usuário | até 32 | até 32 | Não | N/A |
| superi | definida pelo usuário | até 32 | até 32 | Sim | N/A |
| specsuper | super com especulação | até 32 | até 32 | Não | N/A |
| specsuperi | superi com especulação | até 32 | até 32 | Sim | N/A |

É importante mencionar que as implementações de Gamma utilizadas, não fornecem suporte à execução de funções como parte da computação executada como ação de uma reação. Em outras palavras, a computação executada pelas reações possui granularidade fina e, um estudo para permitir aumento da granularidade das operações (ações) executadas por uma reação é foco de trabalhos futuros, não fazendo parte do escopo dessa tese. Maiores informações sobre a granularidade das operações realizadas no modelo Gamma serão apresentadas no decorrer desta tese. Assim, pelos argumentos aqui expostos, o *GFlow* não provê suporte à conversão de instruções utilizadas em chamadas de funções e super-instruções.

#### 4.2.2.3 Benefícios e Contribuições

Fornecer uma ferramenta de conversão entre os modelos dataflow e Gamma permite materializar os benefícios elencados pelo estudo da equivalência. Em outras palavras, permite maior versatilidade no desenvolvimento de aplicações paralelas, uma vez que fornece ao desenvolvedor pouco familiarizado com a sintaxe não tão difundida do modelo, a possibilidade de executar sua aplicação em ambiente Gamma. Isso pode ser particularmente interessante para alguns domínios de aplicação como processamento de imagens, fusão de dados para rastreamento de contatos entre outros (conforme referências elencadas no início da Seção 4.2.2).

Dessa maneira, a proposta contribui para o paradigma Gamma, por incentivar uma maior utilização do modelo, através da possibilidade de conversão, o que indiretamente pode vir a aumentar o interesse e a visibilidade do modelo, contribuindo para um ganho de expressividade de Gamma.

Por outro lado a proposta também contribui para o modelo de fluxo de dados, uma vez que permite com que este modelo tire proveito de estudos e pesquisas desenvolvidas para o paradigma Gamma.

Vale ressaltar que a facilidade de expressão de problemas, do ponto de vista de Gamma é por si só um beneficio [17] e, fornecer uma ferramenta de conversão não obriga nem incentiva o fato de programas serem desenvolvidos em C (com anotações *THLL*) ou diretamente em dataflow em detrimento de serem desenvolvidos em

Gamma. Entretanto, a possibilidade de conversão explora uma quantidade maior de possibilidades, tanto para execução de aplicações em ambientes paralelos diferentes, tanto para explorar benefícios de ambos modelos.

Por fim, é importante frisar que, até o momento do desenvolvimento desta pesquisa, não foi encontrada na literatura nenhuma ferramenta de conversão entre os modelos computacionais Dataflow e Gamma, sendo esta a primeira proposta de implementação da referida conversão.

### 4.2.3 Considerações sobre a Implementação

Antes de abordarmos características sobre a implementação do *GFlow* se faz necessária uma revisão da sintaxe de Gamma utilizada tanto como produto da conversão do *GFlow* quanto como arquivo de entrada do *GSink*. Tal sintaxe de Gamma foi proposta por Juarez Muylaert e maiores detalhes podem ser encontrados em [18, 30].

Com base nesta sintaxe, um código Gamma, expresso em um arquivo **".gm"** é composto da seguinte maneira:

```
<lista de reações> <multiconjunto>
where
<definição da reação 1>
<definição da reação 2>
    .
    .
    .
<definição da reação N>
```

Onde:

- *<**lista de reações**>* - contém o nome de cada reação existente na aplicação separado por operadores sequenciais (";" - operador que determina a execução sequencial entre reações) ou paralelos ("|"). Um exemplo de uma lista de reações para uma aplicação composta por 3 reações $R1$, $R2$ e $R3$ seria: ($R1$ | $R2$); $R3$. Maiores detalhes sobre composição de operadores pode ser visto na fundamentação teórica desta tese (Capítulo 2).

- *<**multiconjunto**>* - é composto pela descrição de todos os elementos iniciais do multiconjunto.

- ***where*** - palavra reservada, que indica o início das especificações das reações.

- *<**definição da reação**>* - definição de cada reação existente na aplicação. Contem o nome da reação, cláusula *REPLACE* (responsável por selecionar os

elementos para reagir), cláusula *BY* (resultado da ação) e cláusula *IF* (expressa a condição de reação).

Um exemplo de um código Gamma de acordo com a sintaxe acima, contendo duas reações genéricas onde uma soma elementos do multiconjunto e a outra subtrai elementos dois a dois pode ser descrita da seguinte maneira:

```
R1 | R2 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

R1 = REPLACE X,Y
     BY X + Y
     IF X < Y

R2 = REPLACE X,Y
     BY X - Y
     IF X > Y
```

#### 4.2.3.1 Aspectos Gerais

O *GFlow* (https://gitlab.com/rui.rmj/gflow) é uma ferramenta de conversão entre modelos computacionais escrita em linguagem Python. Ela aproveita a implementação do Front-End do *FlowASM*, apresentado anteriormente, onde as expressões regulares da linguagem de montagem do *TALM* são analisadas para posterior geração de código binário para a execução na *Trebuchet*, realizado pelo Back-End do *FlowASM*.

Dessa maneira, aproveitando a análise das expressões regulares, cada instrução em linguagem de montagem do *TALM* dará origem a uma ou mais reações (como veremos posteriormente, algumas instruções dão origem a mais de uma reação) no respectivo código Gamma.

O processo de conversão executado pelo GFlow, pode ser descrito de maneira didática em três etapas (loops) distintas (ilustradas na Figura 4.6):

***Primeira Etapa:*** Identificação e armazenamento de informações a respeito de constantes e registradores de saída (nome que identifica o operando de saída de cada instrução). Nesta etapa também é gerada a identificação de cada reação que será utilizada tanto no escopo de cada uma, quanto para a listagem de reações. O padrão de nomes atribuído a cada reação é: "*R_*" sucedido do nome do registrador de saída da instrução *TALM* correspondente. Ainda durante esta primeira etapa é armazenado o nome de cada instrução com um identificador sequencial unívoco, que futuramente será utilizado para fornecimento de identificadores. As informações citadas neste parágrafo são armazenadas em dicionários [1] distintos, a saber:

[1] Dicionários são estruturas em Python que representam coleções de itens, compostos por chaves para indexar valores.

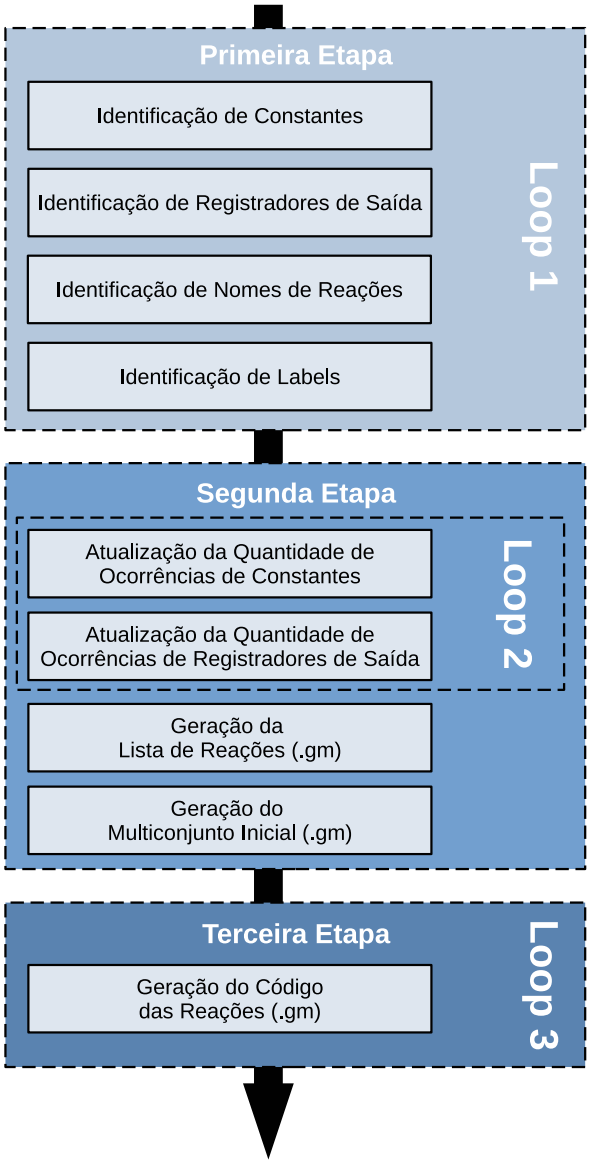


Figura 4.6: GFlow - Etapas do Processo de Conversão.

- *square_nodes* - armazena constantes e quantidade de ocorrência das mesmas no código;
- *square_nodes_values* - contém as mesmas constantes e seus respectivos valores;
- *output_registers* - guarda o nome dos registradores de saída para todas as instruções com exceção das constantes; e
- *labels* - armazena o nome de cada instrução ou constante vinculadas a um identificador sequencial unívoco.

***Segunda Etapa:*** Atualização da quantidade de ocorrências de constantes (cômputo da quantidade de vezes que cada constante é utilizada como operando

para cada instrução). Observe que aqui cada operando pode possuir múltiplos candidatos (Seção 4.2.1.2). Ainda nesta etapa também calcula-se a quantidade de ocorrência de cada registrador de saída, uma vez que um registrador de saída de uma instrução poderá ser operando de outras. Dessa maneira os dicionários *square_nodes* e *output_registers* encontram-se atualizados com suas quantidades de ocorrências no código. No final desta etapa já se possui condições de inserir no código Gamma de saída, a listagem de reações e o multiconjunto inicial.

***Terceira Etapa:*** Geração do código das reações e inserção das mesmas no código Gamma de saída. Nesta etapa, para cada tipo de instrução, são extraídas as cláusulas que compõem uma reação Gamma. Dessa forma, os elementos que serão criados pela cláusula *BY* de uma reação estão diretamente relacionados a quantidade de vezes que o correspondente registrador de saída da instrução *TALM* for utilizado como operando de outras instruções. A operação a ser realizada na ação da reação está diretamente relacionada ao mnemônico da instrução *TALM*. Já as cláusulas *REPLACE* e *IF* estão relacionadas a existência de múltiplos candidatos como operandos das instruções. Informações sobre a implementação destas cláusulas serão fornecidas nas seções seguintes.

#### 4.2.3.2 O Multiconjunto Inicial

Um componente necessário que deve ser fornecido por ocasião da escrita de um código Gamma é o multiconjunto inicial. Nele irão constar todos os elementos iniciais que permitirão a seleção de elementos para execução pelas reações. É importante notar que, conforme o próprio nome desta seção indica, este é o conjunto inicial de elementos, ao passo que a medida em que reações vão sendo executadas, elementos vão sendo modificados, excluídos e incluídos ao multiconjunto.

Na linguagem de montagem do *TALM*, além do conjunto de instruções listados na Tabela 4.1, existe a descrição de constantes, cuja sintaxe é definida por:

```
<mnemônico> <nome>, <valor>
```

Onde:

- <***mnemônico***> pode ser do tipo *CONST* ou *TCONST*;
- <***nome***> é uma *string* que determina o nome do registrador de saída para esta constante; e
- <***valor***> inteiro ou ponto flutuante que corresponde ao valor da constante.

Como podemos observar no exemplo do Código 4.4, temos quatro constantes, além do conjunto de instruções e, por exemplo, a primeira instrução do

tipo *Inctag* cujo registrador de saída é ***flowInstIncTag139658107024664***, utiliza a constante cujo valor é igual a zero (registrador de saída igual a ***flowInstConst139658106973936***) como um dos seus dois candidatos a operando.

Assim, é a partir das constantes existentes no código do *TALM* e utilizadas como operando de alguma instrução é que são extraídos os elementos iniciais do multiconjunto. Em outras palavras, o multiconjunto inicial será composto por elementos extraídos das constantes fornecidas pelo programa *TALM*. Entretanto é preciso atentar para o fato de que mais de uma instrução *TALM* poderá utilizar o mesmo registrador de saída atribuído a uma constante como um de seus operandos. Em termos práticos, isso equivale a dizer que um mesmo valor pode ser utilizado por mais de uma instrução, o que faz bastante sentido. Por este motivo, foi necessário replicar a quantidade de elementos iniciais do multiconjunto por quantas forem as instruções que as utilizem. Isso é computado por ocasião da execução da segunda etapa do processamento descrita anteriormente. Note que, esta característica de replicar elementos no multiconjunto pela quantidade de vezes que forem utilizados por outras instruções também deverá ser respeitada por ocasião da implementação da cláusula *BY*, que produz os elementos de uma reação. Uma ilustração que exemplifica a replicação dos elementos iniciais do multiconjunto é apresentada Figura 4.7. Aqui uma constante cujo registrador de saída é igual a *flowInstConst02* é utilizada por duas instruções, fato este que torna necessária a replicação desta constante no multiconjunto equivalente.

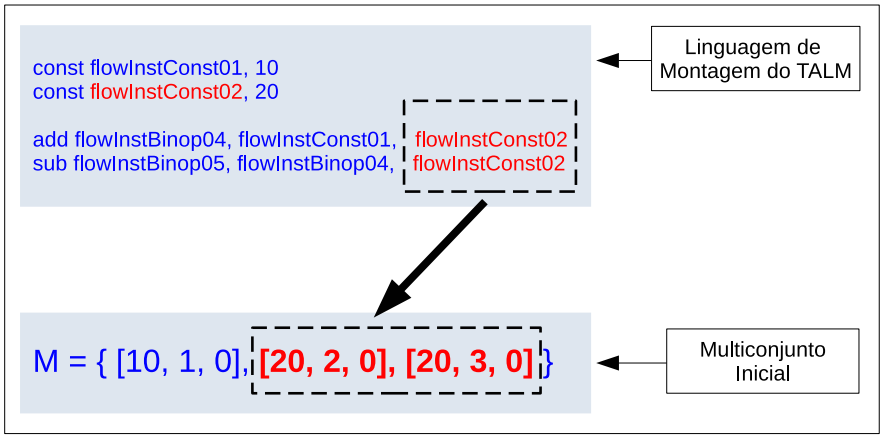


Figura 4.7: GFlow - Replicação de elementos no multiconjunto inicial.

Visando identificar univocamente cada elemento do multiconjunto (inclusive para realizar a distinção entre cópias de um mesmo elemento), foi necessário gerar um identificador sequencial para cada elemento do multiconjunto, além de identificar a qual iteração este dado pertence. Desta forma, os dados no multiconjunto são representados por tuplas de três elementos, definidos por:

```
[ valor, identificador, tag ]
```

Onde:

- ***valor*** corresponde ao valor do elemento do multiconjunto em questão, que pode ser gerado através de constantes (multiconjunto inicial), ou fruto da execução de alguma reação;
- **identificador** diz respeito ao identificador unívoco de cada elemento do multiconjunto; e
- **tag** significa o rótulo de iteração, incrementado pela instrução *Inctag* para diferenciar dados de iterações distintas.

#### 4.2.3.3 Geração das Reações

Agora iremos abordar alguns detalhes a respeito da geração de reações a partir das instruções *TALM*. Para tanto iremos agrupar as reações em conjuntos que possuem o mesmo padrão de processamento: operações aritméticas, operações aritméticas com operando imediato, operações lógicas, operações lógicas com operando imediato, instruções *Steer* e instruções *Inctag*. O processamento destas conversões possuem semelhanças, portanto, iremos abordar de maneira um pouco mais detalhada o primeiro grupo e, a partir daí, focaremos nas diferenças para os demais.

**Operações aritméticas:** os tipos de instruções agrupados nesta categoria possuem o mesmo comportamento no que diz respeito ao preenchimento das cláusulas necessárias para a montagem da reação gamma. Dessa maneira, o nome da reação é obtido através da concatenação de $R\_$ com o nome do registrador de saída da instrução. A operação a ser realizada pela reação pode ser extraída tendo como origem o mnemônico da instrução. Já a cláusula $BY$ poderá criar mais de um elemento do multiconjunto de saída. Isso vai ser definido pela quantidade de vezes que o registrador de saída da instrução corrente for utilizado como operando de outra reação. Desta maneira a segunda etapa do processo de conversão apresentada na Figura 4.6, é responsável, dentre outras coisas, pela atualização da quantidade de ocorrências de registradores de saída de instruções como operando de outras instruções. As cláusulas *REPLACE* e *IF* são compostas por informações dos operandos das instruções, uma vez que estes são os elementos de interesse que cada instrução necessita para disparar sua execução. Entretanto, há de se prestar atenção a existência de múltiplos candidatos para os operandos de cada instrução. Caso exista mais de uma possibilidade de candidato, a cláusula *IF* (combinada à Cláusula *REPLACE*)

deverá expressar a possibilidade de existência de mais de um candidato. O tratamento de múltiplos candidatos será apresentado na próxima seção. Tendo extraido as informações acima descritas (nome da reação, operação e cláusulas *REPLACE*, *IF* e *BY*), já é possível montar o código Gamma e escrever no respectivo arquivo **".gm"** (tarefa executada na terceira etapa do processo de conversão - Figura 4.6).

É importante mencionar que, tanto para replicação de elementos criados pelas reações quanto para replicação de cópias de elementos do multiconjunto inicial, são definidos *buffers* de identificadores indexados pela quantidade de *labels* [2] da aplicação. Ou seja, se uma aplicação em linguagem de montagem do *TALM* possui um conjunto de definição de constantes e instruções totalizando 10, o segundo *buffer* de identificadores será de 10 a 19 (uma vez que o primeiro conjunto compreende o intervalo de 0 a 9), o terceiro de 20 a 29 e assim sucessivamente. Estes *buffers* são utilizados para identificar elementos do multiconjunto que precisam ser replicados, de forma que uma reação $R$ que necessite duplicar o resultado de sua operação (cláusula $BY$) irá atribuir um identificador do primeiro *buffer* para o primeiro elemento criado e outro identificador do segundo *buffer* para o outro elemento. A Figura 4.8 ilustra a situação de 3 *buffers* para uma quantidade de 10 *labels* (entre instruções e contantes). Na Figura, existem 3 ocorrências de utilização da constante cujo registrador de saída é igual a *"a"*, duas ocorrências da constante *"d"*, duas ocorrências da instrução *add* com registrador de saída *"y"* e três ocorrências da instrução *"mult"*, registrador de saída *"d"*.

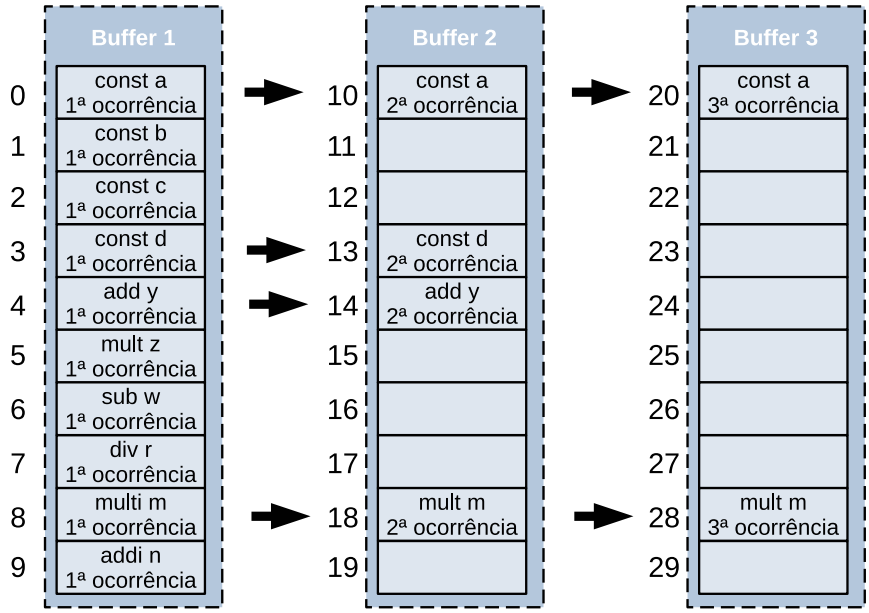


Figura 4.8: GFlow - *Buffers* de identificação de constantes e instruções.

[2]Labels aqui refere-se ao dicionario mencionado anteriormente para armazenar o registrador de saída de cada instrução e de cada constante, indexado por um identificador sequencial único.

**Operações aritméticas com operando imediato:** possui processamento de conversão idêntico ao apresentado anteriormente, exceto pelo fato de um de seus operandos possuir um valor (imediato). Dessa maneira, somente um elemento será selecionado pela cláusula *REPLACE* da reação Gamma equivalente, uma vez que o valor imediato já consta da operação da reação. Por exemplo, considere a instrução de soma com imediato abaixo:

```
addi flowInstBinop0931, flowInstConst3245, 25
```

Tal instrução realiza a soma do valor expresso na constante cujo registrador de saída é *flowInstConst3245* com o imediato de valor 25. A reação equivalente em Gamma irá somente selecionar o elemento no multiconjunto cujo id corresponda a *flowInstConst3245*. Uma vez selecionado este elemento, a operação expressa pela reação já realiza a soma com o valor 25. Considerando que o identificador sequencial de *flowInstConst3245* é 10 e o valor desta constante é 2, produzindo o elemento $[2, 10, 0]$, um fragmento do código Gamma equivalente seria dado por:

```
R_flowInstBinop0931 = REPLACE [X,10,0]
    BY [X + 25, 8, 0]
    IF true
```

Onde o valor "8" que consta na tupla descrita na cláusula $BY$ seria o identificador do elemento criado por esta reação.

**Operações lógicas:** Este grupo de instruções reúne aquelas instruções cuja conversão para Gamma dará origem a duas reações. Isto deve-se ao fato de que foi necessário gerar a reação *antagônica* à operação. Por exemplo, para a operação lógica de "menor que" duas reações foram criadas: a própria reação que exprime a comparação "menor que" e a reação que exprime a comparação "maior". Os motivos que levaram a esta necessidade serão expostos na seção seguinte. É importante mencionar que para este conjunto de instruções, que compreende as instruções *LTHAN*, *GTHAN*, *LEQ* e *GEQ*, o resultado da aplicação da ação da reação produz sempre uma tupla (ou tuplas) contendo valores Booleanos. Ou seja, o campo "valor" da tupla será sempre igual a zero ou um.

**Operações lógicas com operando imediato:** possuem comportamento semelhante ao apresentado no parágrafo anterior, exceto pela existência de um operando imediato. As observações a respeito de operandos imediatos realizadas quando abordamos as "Operações aritméticas com operando imediato" também aplicam-se aqui.

**Instruções Steer:** a conversão deste tipo de instrução também gera duas reações. Mas, neste caso, o motivo desta duplicação de reações é a necessidade de gerar uma tupla especifica quando o *seletor* possui valor zero e outra tupla quando o mesmo possui valor um. Lembrando que uma instrução do tipo *steer* replica em sua saída o valor do segundo operando caso o primeiro (operando *seletor*) seja verdadeiro. Estas tuplas necessitam ser diferentes uma vez que algumas instruções podem utilizar o registrador de saída "true" de um *steer* enquanto outras instruções utilizem o registrador de saída "false" como operando.

**Instruções Inctag:** Por fim, o último tipo de instrução corresponde às *Inctags*, responsáveis por incrementar o rótulo de iteração. Uma vez que o último campo da tupla que representa os elementos do multiconjunto é o rótulo de iteração (tag), a reação correspondente realiza o incremento deste campo. Para isso, seleciona o elemento no multiconjunto, através da cláusula *REPLACE*. Este elemento corresponde ao único operando da instrução, que pode admitir múltiplos candidatos. O elemento que será devolvido ao multiconjunto terá seu rótulo de iteração incrementado, na própria cláusula *BY*, conforme o fragmento de código Gamma abaixo:

```
R_flowInstIncTag1425 = REPLACE [X, 21, tag]
    BY [X, 32, tag+1]
    IF true
```

O código acima corresponde a uma instrução *Inctag* que possui como operando, um registrador de saída (de uma instrução ou constante) que possui identificador igual a 21. Ainda pela análise do fragmento de código acima podemos verificar que o identificador da instrução *inctag* cujo registrador de saída é igual a $R\_flowInstIncTag1425$ é igual a 32.

#### 4.2.3.4 Adequações Necessárias

Tendo em vista características da implementação de Gamma utilizada, e da linguagem de montagem do *TALM*, a presente seção abordará algumas adequações que foram necessárias visando a correta conversão entre os modelos.

**Operandos com Múltiplos Candidatos:** Como vimos anteriormente na Seção 4.2.1.2, podem existir múltiplos candidatos para determinado operando de dada instrução. Como a cláusula *REPLACE* de uma reação é responsável por selecionar quantos e quais elementos irão ser submetidos à condição de reação, por ocasião da conversão, esta cláusula reflete a existência de múltiplos candidatos como operando de uma instrução. O Código abaixo apresenta uma instrução de multiplicação onde o primeiro operando possui dois possíveis candidatos:

```
mult flowInstConstBinop12, [flowInstConst26,flowInstConst22], flowInstConst20
```

Visando facilitar o entendimento, vamos substituir os nomes dos registradores de saída, tanto da instrução quando os utilizados nos operandos, por inteiros, da seguinte maneira:

```
mult 12, [26,22], 20
```

O código Gamma equivalente a esta instrução TALM de multiplicação é definido por:

```
R_flowInstConstBinop12 = replace [x, w, tag], [y, 20, tag1]
    by [x * y, 12, tag]
    if (tag == tag1) and (w == 26 or w == 22)
```

Repare que na cláusula *REPLACE* duas tuplas estão sendo selecionadas, cujos valores serão multiplicados como consta no primeiro campo da tupla de resultado, na cláusula *BY*. A segunda tupla, necessariamente deve possuir identificador com valor "20", conforme o código da instrução *TALM*. Já a primeira tupla utiliza uma variável auxiliar "w" como campo identificador (segundo campo da tupla). Esta variável auxiliar é utilizada para permitir vários candidatos, conforme descrito na cláusula *IF*, quando "w" pode assumir os valores "26" ou "22". Ainda na cláusula *IF*, é realizada a verificação de "tags" visando permitir que somente elementos com mesmo rótulo de iteração possam vir a ser selecionados para reagir.

**Duplicação de Reações**: a sintaxe de Gamma das implementações utilizadas não provê suporte a uma cláusula *ELSE*. Em outras palavras só é fornecida a cláusula *IF* e, para produzir elementos equivalentes à situação onde o teste condicional do *IF* for igual a falso, deve-se utilizar outra reação. Por isso, tanto para as instruções lógicas quanto para os *Steer* são utilizadas duas reações por ocasião da conversão.

Para as instruções lógicas, se faz necessário produzir uma tupla cujo valor (primeiro campo) seja igual a um, caso o teste condicional da instrução seja verdadeiro e, uma tupla com valor zero, caso contrário. Dessa forma, para cada instrução lógica do *TALM* (LTHAN, GTHAN, LEQ e GEQ), duas reações são fornecidas. A primeira delas expressa a comparação lógica determinada pela instrução e, caso verdade, um elemento cuja tupla possua um campo valor igual a um é produzido. A segunda reação fornecida realiza a comparação "antagônica" à primeira e produz uma tupla com campo valor igual a zero, caso a condição de reação seja verdadeira. Por exemplo, no caso de uma instrução *LTHAN*, a primeira reação equivalente faz o teste condicional "menor" enquanto que a segunda reação realiza o teste "maior ou igual". Repare que em ambos os casos o campo "identificador" das tuplas fornecidas possuem o mesmo valor. A Figura 4.9 ilustra a conversão de uma instrução

*LTHAN*, na qual os operandos *flowInstConst06* e *flowInstConst07* darão origem a identificadores "6" e "7" respectivamente, conforme indicado nas tuplas selecionadas na cláusula *REPLACE*, que a propósito, são as mesmas em ambas reações. A reação mais acima implementa a funcionalidade "<", enquanto que a outra implementa a funcionalidade ">=". Repare ainda que há diferenças nas tuplas produzidas pelas reações, enquanto uma produz o valor "1" a subsequente produz o valor "0", ambos com mesmo identificador (10).

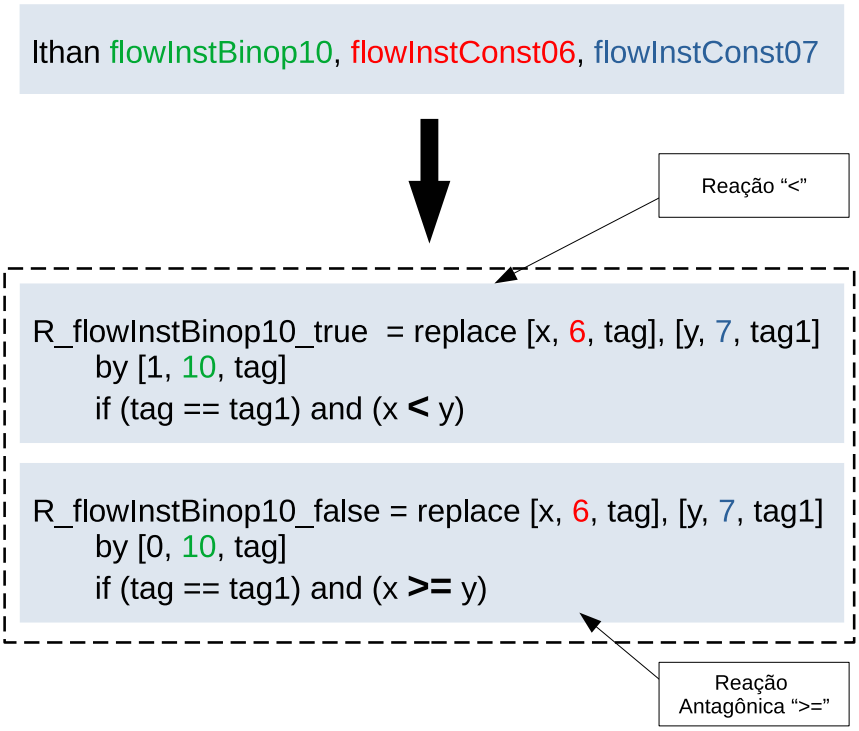


Figura 4.9: GFlow - Exemplo de conversão de instrução lógica.

No caso de instruções *Steer* também se faz necessário fornecer duas reações a cada instrução convertida. A primeira delas avalia o operando "seletor" (primeiro operando) e, caso positivo, fornece uma tupla contendo o valor do segundo operando e contendo um identificador específico. A segunda reação também avalia o operando "seletor", entretanto só reage (produz o elemento no multiconjunto) se tal seletor for igual a zero e, neste caso, o identificador da tupla gerada é diferente do identificador da tupla gerada pela primeira reação. É importante diferenciar os identificadores das tuplas geradas por estas duas reações uma vez que as instruções no *TALM* utilizam o registrador de saída dos *Steer* diferenciando a saída "true" da saída "false" dessas instruções. A Figura 4.10 ilustra a conversão de uma instrução do tipo *Steer*. Repare que os identificadores das tuplas geradas por cada reação são diferentes. Além disso, a primeira reação verifica se o valor do operando "seletor" (operando cujo identificador é igual a "6") é verdadeiro, enquanto que a segunda verifica se o mesmo é falso.

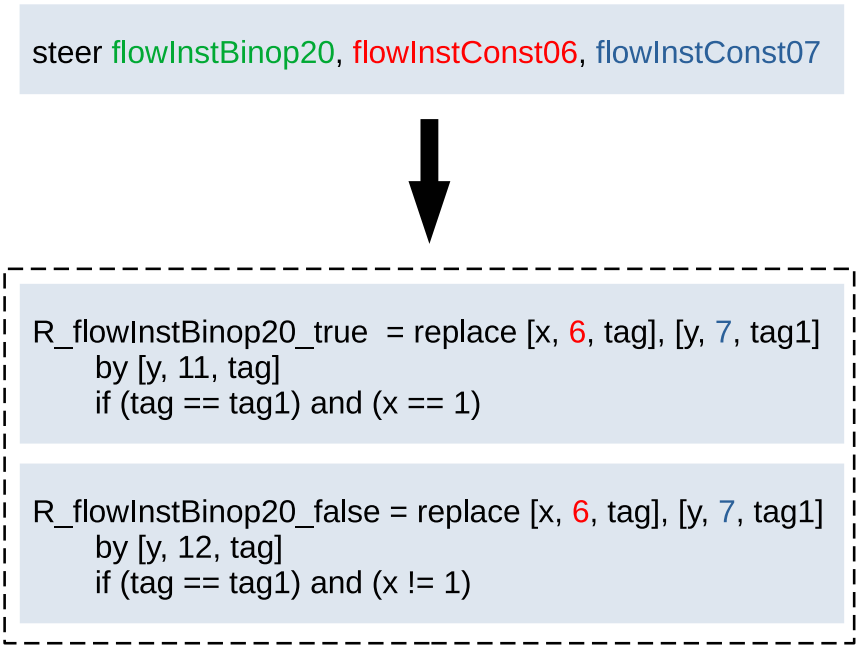


Figura 4.10: GFlow - Exemplo de conversão de instrução do tipo *Steer*.

**Replicação de Elementos:** a segunda etapa do processo de conversão (Figura 4.6) é responsável por atualizar a quantidade de ocorrências de registradores de saída. Ou seja, é verificada em toda a aplicação a quantidade de vezes que determinado registrador de saída é utilizado como operando de alguma instrução. Isso é particularmente importante, uma vez que demonstra quantos elementos deverão ser fornecidos por determinada reação. Assim, quando é necessário replicar elementos de saída de uma reação a cláusula $BY$ apresenta esta quantidade de elementos, todos identificados em seus respectivos *buffer* de identificação, conforme mencionamos anteriormente.

Da mesma forma, ainda durante a segunda etapa da conversão, também existe a necessidade de replicação dos elementos iniciais do multiconjunto, conforme discutimos anteriormente na Seção 4.2.3.2.

**Identificadores inteiros:** por vezes se faz necessário identificar determinada tupla, como por exemplo, o elemento gerado pela instrução cujo registrador de saída é igual a *flowInstBinop65*. Neste caso, a tupla poderia ser formada da seguinte forma:

```
[23, flowInstBinop65, 0]
```

Onde esta tupla identificaria a execução da instrução cujo registrador de saída é igual a *flowInstBinop65*, contendo o resultado igual a 23 e pertencente a iteração

com rótulo igual a zero.

Entretanto, não é possível referenciar-se a "flowInstBinop65" como identificador de tupla, uma vez que as implementações de Gamma utilizadas não provêm suporte a *strings*. Dessa maneira, para cada registrador de saída de instruções e para cada constante do programa, é atribuído um identificador inteiro unívoco, armazenado no dicionário *labels*.

#### 4.2.3.5 Exemplo de Conversão

Agora iremos apresentar um exemplo simples de conversão, focado em detalhes que apresentamos anteriormente. Outros exemplos abordando cada categoria de instrução, além da demonstração de conversão utilizando o *Couillard* para fornecer a linguagem de montagem do *TALM* a partir de um código escrito em C, serão apresentados no Capítulo 5.

Considere o Código 4.5 abaixo, escrito em linguagem de montagem do *TALM*:

```
1 const a, 5
2 const b, 8
3 const c, 3
4 add adicao, a, b
5 sub subtracao, adicao, b
6 lthan menorque, desvio.t, adicao
7 steer desvio, c, adicao
```

Código 4.5: Exemplo de código em linguagem de montagem do *TALM*.

Visando facilitar o entendimento, os registradores de saída foram substituídos por *strings* simples, ao invés das *strings* comumente atribuídas pelo *Couillard*, conforme em muitos exemplos anteriores. Note que esta substituição não impede a conversão pelo *GFlow*.

O código Gamma equivalente, após conversão pelo *GFlow* é dado por:

```
1 R_adicao | R_subtracao | R_menorque_true | R_menorque_false |
    R_desvio_true | R_desvio_false {[5, 0, 0], [3, 2, 0], [8, 9,
    0], [8, 1, 0]}
2
3 where
4
5 R_adicao = replace [x, 0, tag], [y, 9, tag1]
6 by [x + y, 3, tag], [x + y, 19, tag], [x + y, 11, tag]
7 if (tag == tag1)
8
9 R_subtracao = replace [x, 19, tag], [y, 1, tag1]
10 by [x - y, 4, tag]
11 if (tag == tag1)
12
```

```
13 R_menorque_true = replace [x, 6, tag], [y, 11, tag1]
14 by [1, 5, tag]
15 if (tag == tag1) and (x  <  y)
16
17 R_menorque_false = replace [x, 6, tag], [y, 11, tag1]
18 by [0, 5, tag]
19 if (tag == tag1) and (x  >=  y)
20
21 R_desvio_true = replace [x, 2, tag], [y, 3, tag1]
22 by [y, 6, tag]
23 if (tag == tag1) and (x == 1)
24
25 R_desvio_false = replace [x, 2, tag], [y, 3, tag1]
26 by [y, 7, tag]
27 if (tag == tag1) and (x != 1)
28
29 /* Qtde Labels =  8  */
30
31 /* Square_nodes_values            = {[5, a], [3, c], [8, b]}  */
32
33 /* Square_nodes (Ocorrencias)     = {[2, a], [2, c], [3, b]}  */
34
35 /* Output_registers (Ocorrencias) = {[1, desvio.t], [0, desvio.f],
     [3, adicao], [0, menorque], [0, subtracao]}  */
36
37 /* Labels                         = {[0, a], [2, c], [1, b], [3,
    adicao], [6, desvio.t], [5, menorque], [7, desvio.f], [4,
    subtracao]}  */
38
39 /* Ocorrencias                    = {[1, a], [1, desvio.t], [1, c
    ], [2, b], [3, adicao]}  */
```

Código 4.6: Exemplo de código Gamma convertido pelo *GFlow*.

Note que inicialmente a lista de reações é impressa, onde cada instrução é separada pelo operador paralelo "|". Nesta listagem de instruções, apesar da quantidade de 4 instruções no código em linguagem de montagem inicial, existem 6 reações. Isso deve-se ao fato de duas instruções (*LTHAN* e *STEER*) necessitarem de duplicação. Note também que no multiconjunto inicial existe a duplicação da tupla referente à constante cujo valor é 8. Isso ocorre pois esta constante está sendo utilizada por duas instruções distintas (*ADD* e *SUB*). Após a palavra reservada "where", cada instrução é definida. Ao final do código, alguns comentários são impressos visando prestar maiores informações ao desenvolvedor, que podem ser úteis em momento de *Debug* e desenvolvimento, como por exemplo quantidade de *labels* e informações a respeito dos dicionários utilizados.

## 4.3 GSink

Tendo finalizado a apresentação e discussão sobre nossa proposta de ferramenta para conversão entre modelos computacionais, realizada na Seção anterior, a presente Seção é dedicada a nossa proposta de ambiente de execução para programas Gamma: o *GSink*. Como parte da motivação em propor o *GSink* surge de características verificadas em outras implementações do paradigma Gamma (mais notadamente aquelas fornecidas por Juarez Muylaert), inicialmente apresentaremos alguns trabalhos relacionados no que diz respeito à implementação de Gamma. Após isso, apresentaremos o *GSink*, onde abordaremos sua descrição, benefícios e contribuições além de uma série de considerações sobre a implementação da ferramenta. Da mesma maneira como procedemos com o *GFlow*, no Capítulo 5 apresentaremos experimentos com objetivo de discutir a corretude da execução de programas Gamma além de realizar uma análise do potencial de nossa proposta para executar reações com granularidade grossa.

### 4.3.1 Trabalhos Relacionados - Implementações de Gamma

Devido a seu potencial natural como linguagem de programação para concorrência, várias implementações de Gamma foram propostas, como alguns projetos desenvolvidos na década de 90, como [69] e [70], entre outros.

Na implementação fornecida por CREVEUIL [69], o principal problema encontrado foi a seleção de elementos do multiconjunto que satisfaçam a condição de reação, uma vez que a execução da ação expressa pela reação Gamma pode ser considerada como uma operação local, independente do multiconjunto remanescente. O algoritmo utilizado para analisar todos os pares de elementos é baseado em *odd-even transposition sort* [71] e sua condição de terminação aguarda $n$ passos sem que nenhuma reação ocorra. Esta implementação foi realizada em uma *Connection Machine* e foi utilizada para testar exclusivamente reações binárias, que sempre produzem dois elementos para cada ação. Além disso, essa implementação não abordou programas Gamma compostos por mais de uma reação.

Conforme apresentado em [62], uma implementação sequencial de Gamma foi proposta por Juarez Muylaert e Simon Gay. Trata-se de uma implementação que utiliza somente um processador para executar programas escritos em Gamma. Desta forma, a noção de paralelismo é obtida através da execução de reações em um mesmo processador utilizando algum mecanismo para garantir a mudança entre estas reações neste processador. Os mesmos autores também propuseram uma implementação paralela do formalismo Gamma. Neste caso, as reações são executadas em um ambiente paralelo através da utilização de MPI - *Message Passing Interface*, um protocolo de comunicação entre múltiplos processadores. Outra implementação

paralela de Gamma foi proposta em 2015, agora com suporte à GPU - *Graphics Processing Unit* em um ambiente paralelo [72]. Com relação ao domínio de aplicação onde o paradigma Gamma foi aplicado, podemos citar processamento de imagens [73], fusão de dados para acompanhamento de contatos [17], entre outros.

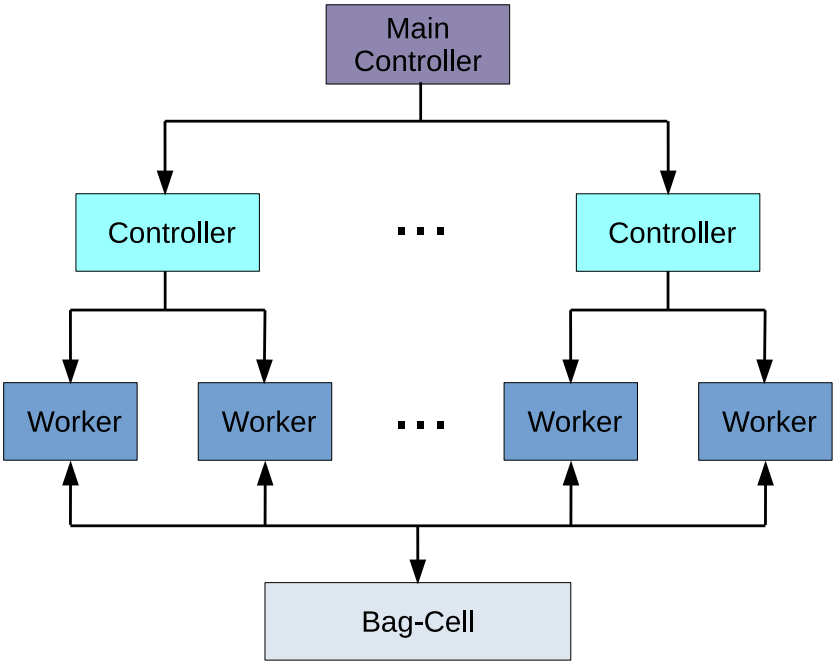


Figura 4.11: Esquema da Implementação de Gamma de Muylaert [62].

A Figura 4.11 apresenta um esquema proposto por Juarez Muylaert para implementar o paradigma Gamma em um ambiente distribuído. Dado um programa escrito em Gamma, quatro tipos de células de processamento são criadas: *Bag-Cell*, *Main Controller*, *Controller* e *Worker*. Os *Workers* são responsáveis por executar as reações Gamma sobre o multiconjunto, testando as condições de reação e executando as ações, no caso onde tais condições sejam verdadeiras. Desta forma, teremos tantos *Workers* quantas forem as reações no programa Gamma. Cada *Controller* conduz a execução de duas outras células (*Workers* ou *Controllers*), representando operadores que permitem a execução sequencial e paralela de reações Gamma. A célula denominada *Main Controller* controla a execução de todas as outras células, indicando que há trabalho a ser feito. Além disso, esta célula controla o término do programa, alcançado quando nenhumma condição de reação é satisfeita. Por fim, a *Bag-Cell* controla todo o gerenciamento de utilização do multiconjunto, conduzindo, enviando e recebendo elementos do multiconjunto para os *Workers*. Quando um *Worker* deseja processar o multiconjunto, é necessário primeiro solicitá-lo à *Bag-Cell*. A *Bag-Cell* enviará o multiconjunto somente se nenhuma outra célula o estiver processando. Ao final, o *Worker* devolve o multiconjunto à *Bag-Cell*.

É importante notar que a abordagem descrita acima possui um gerenciamento centralizado do multiconjunto e não permite a execução de múltiplas reações ao mesmo tempo. Por exemplo, se o programa for composto por duas diferentes reações, $R1$ e $R2$, nesta abordagem não é possível $R1$ processar parte dos elementos do multiconjunto ao mesmo instante em que $R2$ processa outros elementos. Mesmo que um programa seja composto por uma única reação, esta implementação não permite múltiplas instâncias desta única reação. A implementação proposta por DE ALMEIDA *et al.* [72] aperfeiçoou a implementação de Muylaert pela utilização de processamento pela *GPU* na computação realizada pelos *Workers*. Entretanto, o mesmo esquema de gerenciamento centralizado do multiconjunto apresentado em Figura 4.11 foi utilizado.

### 4.3.2 Um ambiente de execução baseado em um grafo acíclico

A proposta da *Fluid Computing* [5, 74] (Apêndice C) sugere utilizar-se de Gamma para prover uma plataforma computacional onde o modelo de execução seria baseado em dataflow, através da utilização de um protocolo de comunicação baseado em interesses (RadNet). A intenção é investir em Gamma como um modelo computacional que permeie os recursos computacionais disponíveis em um ambiente de Computação Heterogênea [75–77]. Entretanto, algumas características da atual implementação do protocolo postergaram a proposta inicial.

Por outro lado, as implementações de Gamma existentes apresentaram alguns gargalos computacionais importantes, como por exemplo o gerenciamento centralizado do multiconjunto que não permite que a execução de instâncias de uma mesma reação (nem de instâncias de reações distintas) ocorra de maneira paralela, conforme abordamos na Seção 4.3.1.

Em 2005, Gabriel Paillard propôs um novo modelo de escalonador distribuído para Gamma [22]. A proposta de escalonamento era bastante promissora e ataca gargalos computacionais das implementações anteriores.

Diante da postergação da implementação da *Fluid Computing* como um ambiente de execução baseado em dataflow, dos gargalos computacionais observados nas implementações de Gamma atuais e da proposta de escalonamento de [22] surge a motivação em investir em um ambiente de execução que contribuísse para sobrepor os obstáculos verificados nas implementações de Gamma existentes. Além disso, a própria contribuição fornecida com a proposta do *GFlow* como ferramenta de conversão, fomenta o investimento em um ambiente de execução Gamma, uma vez que trás consigo um investimento em utilização e expressividade no modelo Gamma.

#### 4.3.2.1 Descrição

O *GSink* consiste na primeira implementação de um ambiente de execução de programas Gamma que permite a execução de instâncias de uma mesma reação ou de várias reações de maneira paralela. A Figura 4.12 descreve de maneira simplificada o contexto de entradas e saídas do *GSink*.

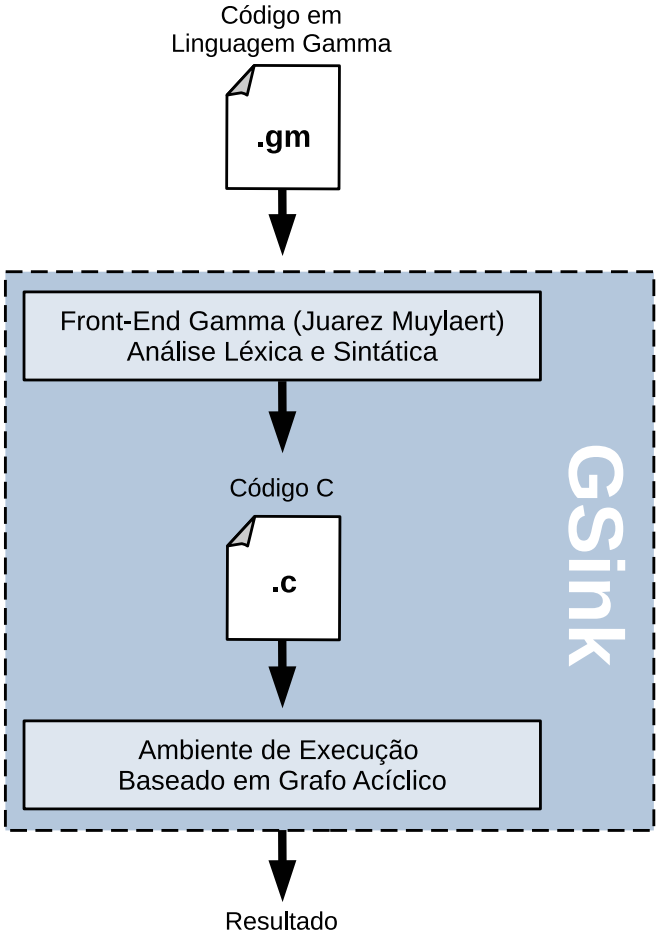


Figura 4.12: GSink - Visão Geral.

O *GSink* recebe como entrada um código escrito em linguagem Gamma (com extensão **".gm"**), utilizando a sintaxe fornecida por Juarez Muylaert, cuja descrição pode ser encontrada em [18, 30]. Assim, o arquivo de entrada descreve a listagem de reações, a definição do multiconjunto inicial e a definição de cada reação que compõe o código Gamma em questão, conforme apresentamos na Seção 4.2.3.2. Para as tarefas afetas à análise léxica e sintática, utilizamos o *Front-End* fornecido por Juarez Muylaert por ocasião das implementações *Gamma-Sequencial* e *Gamma-MPI*. Dessa forma, através da extração de informações da árvore sintática produto do *Front-End* mencionado, montamos um código em linguagem C para ser submetido à execução no ambiente que implementamos no *GSink*.

O Algoritmo usado para o escalonamento da execução foi baseado em [22], onde é utilizado mapeamento ótimo de sistemas com restrições sobre a vizinhança [63], que por sua vez baseia-se na técnica de *SER (Scheduling by Edge Reversal)* [78] e *MCC (Minimum Clique Covering)*. Nesta abordagem, recursos compartilhados, são representados por um grafo orientado finito. O *SER* é inicializado a partir de uma orientação acíclica sobre o grafo $G$, significando que sempre existirá pelo menos um vértice do tipo *Sink* [3]. O algoritmo utiliza reversão de arestas de *sinks*, o que acarreta que sempre teremos novos *sinks* a cada orientação acíclica de $G$. Através deste mecanismo de reversão de arestas de *sinks*, será encontrado um período $p$, onde as orientações acíclicas passam a se repetir.

Em nossa implementação, o Grafo $G$ representa o compartilhamento de elementos do multiconjunto Gamma, onde os vértices representam os elementos do multiconjunto candidatos a compor uma possível instância de execução de alguma reação e as arestas representam o compartilhamento destes recursos entre os nós de $G$. Nossa proposta difere de [22], essencialmente, dentre outras características, na medida em que não utilizamos um grafo complementar para identificação dos *sinks* e aplicação do *SER*. Nosso interesse é descobrir quais instâncias de reações podem vir a executar de maneira paralela, ao passo que em [22] o interesse era identificar em quais elementos de processamento tais instâncias de execução poderiam ser distribuídas.

#### 4.3.2.2 Etapas do Processo de Escalonamento

A Figura 4.13 apresenta uma visão esquemática dos passos executados pelo algoritmo de escalonamento de execução de reações do *GSink*. As etapas de processamento são distribuídas em ondas (*"waves"*) onde em cada uma delas, um grafo é gerado contendo vértices que correspondem às combinações de elementos do multiconjunto que vão sendo consumidos (excluídos do grafo) à medida em que os *sinks* vão sendo executados. Quando não existirem mais vértices a serem consumidos, um novo grafo é gerado a partir dos elementos do multiconjunto atual (contendo as alterações realizadas pela execução dos *sinks*), dando início a uma nova onda. A condição de parada do algoritmo é quando se alcança um estado onde não se possa mais combinar os elementos restantes do multiconjunto para criar vértices (uma vez que, como veremos mais a frente, é armazenado um histórico de vértices já fornecidos ao Grafo). Portanto, podemos descrever de maneira genérica os passos executados por uma *wave* da seguinte forma:

**Passo 1** - Criação de *ReAgentes* - Para cada reação Gamma que compõe o código fonte, é criado um agente de reação (*ReAgente*), cuja função é gerenciar a criação

[3] Sink é um nó de um grafo que não possui arestas de saída, ou seja, toda aresta existente está direcionada a seu favor.

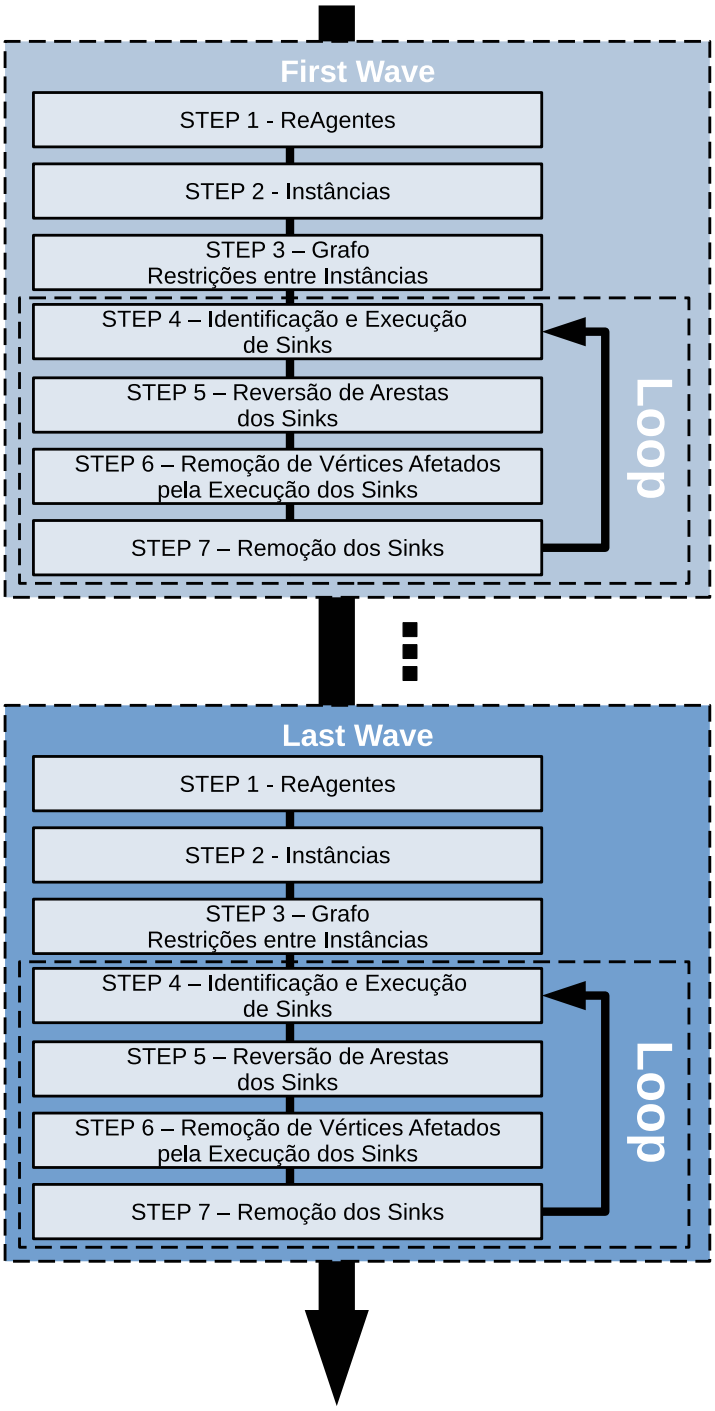


Figura 4.13: GSink - Esquema Geral de Processamento.

e fornecimento de instâncias para execução. Desta forma cada *ReAgente* possui uma cópia do multiconjunto, o histórico de todas as instâncias já fornecidas e uma relação de instâncias a serem fornecidas.

**Passo 2** - Instâncias - Neste passo, cada *ReAgente* gera instâncias de execução da reação ao qual estão relacionadas. Uma instância pode ser definida como uma

combinação de elementos do multiconjunto que possa ser utilizada como candidata à execução por uma reação. Dessa maneira, se a reação $R$ seleciona elementos inteiros dois a dois para tentar reagir, tendo em vista alguma condição de reação, podemos sugerir as seguintes instâncias, como exemplo: $(3, 4), (5, 2), (231, 0), (3423, 55)$. Ou seja, neste passo, cada *ReAgente* gera combinações de elementos (combinação sem repetição) de forma a utilizar todo elemento do multiconjunto para criar instâncias que serão candidatas a execução desta reação. Vale ressaltar que a geração de instâncias leva em consideração as instâncias já fornecidas por ocasião de outras *"waves"* conforme veremos posteriormente.

**Passo 3** - Geração do Grafo - Tendo sido gerado, no passo anterior, um conjunto de instâncias de execução para cada reação, no passo 3 estas instâncias serão inseridas como vértices do grafo $G$. Entretanto, conforme mencionamos anteriormente, é necessário gerar uma orientação acíclica do grafo para futura identificação de *sinks* como preconiza o *SER*. A maneira utilizada para garantir a orientação acíclica do grafo foi, a medida em que um vértice (instância de execução de alguma reação) for inserido no grafo, verifica-se se o mesmo compartilha recursos (elementos do multiconjunto que compõem a referida instância de execução) com outro vértice pré-existente. Caso exista tal compartilhamento, é criada uma aresta entre estes dois vértices, entretanto, o sentido será sempre a favor do vértice mais antigo no grafo. Em outras palavras, ao inserirmos um vértice no grafo este só terá arestas de saída (caso tenha alguma aresta). Vale ressaltar que neste passo 3 instâncias de todas as reações serão inseridas como vértices do grafo $G$. Isso é uma característica importante de nossa proposta pois permite coexistência natural de instâncias de reações distintas.

**Passo 4** - Identificação e Execução dos *Sinks* - Durante esta etapa, são identificados aqueles vértices que não possuem arestas de saída (*Sinks*). É importante notar que como as arestas representam compartilhamento de recursos entre vértices (restrições entre instâncias), *sinks* de uma mesma orientação acíclica necessariamente não compartilham recursos e, portanto, podem executar de maneira paralela. Assim, no passo 4, todos os *sinks* tentam executar suas instâncias de reações.

**Passo 5** - Reversão de Arestas dos *Sinks* - Após executarem, os *sinks* revertem suas arestas, ou seja, cada aresta de entrada que por ventura possuam será transformada em aresta de saída, o que possibilitará a futura identificação de outros *sinks*.

**Passo 6** - Remoção de Vértices afetados pela execução dos *Sinks* - No passo 4, os *sinks* tentam executar suas instâncias de reação. Note que é uma tentativa de

execução, uma vez que o teste condicional da reação será aplicado neste momento. Dessa maneira, a referida instância pode não vir a executar. Tomemos como exemplo a seguinte reação Gamma:

```
R = replace x, y
    by x
    if x > y
```

A reação $R$ seleciona elementos dois a dois e devolve ao multiconjunto o primeiro elemento, caso este seja maior que o segundo. Ou seja, se os elementos selecionados como par x,y forem, por exemplo, $(5, 9)$, a reação não ocorre e nenhuma alteração no multiconjunto é realizada. Assim, caso um *sink* não execute, nenhuma alteração no multiconjunto será realizada. Entretanto, caso o *sink* consiga executar sua instância de reação, o multiconjunto é alterado e, os demais vértices que compartilham recursos com este *sink* devem ser removidos do grafo $G$. Isso deve-se ao fato de que tais recursos compartilhados podem ter sido modificados pela execução dos *sinks*. Portanto, o passo 6 remove os vértices que compartilham recursos com os *sinks* que conseguiram executar suas instâncias de reações.

**Passo 7** - Remoção dos *Sinks* - Após a remoção dos nós afetados pela execução dos *sinks*, os próprios *sinks* são excluídos do grafo. Neste passo, tanto os *sinks* que executaram quanto os que não executaram são retirados do grafo. Os que conseguiram reagir modificaram os elementos que fazem parte de sua instância de execução e, consequentemente, alteraram o multiconjunto. Já os que não reagiram irão disponibilizar seus elementos (que não foram alterados por sua execução) para uma futura geração de instâncias.

Após a execução do passo 7, retornar-se-a ao passo 4, procurando identificar e executar novos *sinks*, caso ainda existam vértices no grafo $G$. Isso caracteriza o *"loop"* identificado na Figura 4.13. Caso não existam vértices a serem identificados no passo 4, uma nova *wave* é iniciada, com a identificação de *ReAgentes* a partir do multiconjunto atualizado, por ocasião da execução de todos os *sinks* que executaram na *wave* anterior. Note que, conforme mencionamos no passo 2, a criação de instâncias leva em consideração o histórico de instâncias já fornecidas às outras *waves*. Assim, caso não existam mais elementos no multiconjunto a serem selecionados para compor instâncias, seja pela quantidade de elementos restantes, seja pelo histórico de instâncias fornecidas, a computação termina.

#### 4.3.2.3 Exemplo das Etapas do Processo de Escalonamento

Para exemplificar os passos descritos na Seção anterior, considere o exemplo de execução do código Gamma descrito abaixo (Código 4.7):

```
1  P1 | P2 {1, 2, 3, 4, 5, 6, 7, 8}
2
3  where
4
5  P1 = replace x, y
6      by x + y
7      if (x > 4) and (y > 4)
8
9  P2 = replace x, y
10     by x - y
11     if (x <= 4) and (y <= 4)
```

Código 4.7: Exemplo das Etapas do Algoritmo de Escalonamento do *GSink*.

O código é composto por duas reações binárias onde a primeira (P1) realiza a soma dos elementos selecionados desde que estes sejam maiores que 4, ao passo que a segunda reação, P2, devolve ao multiconjunto a subtração do primeiro elemento pelo segundo, uma vez que estes sejam menores ou iguais a 4.

O primeiro passo do algoritmo de escalonamento cria os *ReAgentes*, cada um com sua cópia do multiconjunto:

**Passo 1 - Reagentes**:
$R1 - \{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8\}$
$R2 - \{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8\}$

O passo seguinte é responsável por criar as instâncias de execução para cada *ReAgente*, de modo a utilizar todos os elementos do multiconjunto:

**Passo 2 - Instâncias**:
$R1 - \{(7,\ 8),\ (6,\ 1),\ (2,\ 3),\ (5,\ 4)\}$
$R2 - \{(1,\ 2),\ (3,\ 7),\ (4,\ 5),\ (6,\ 8)\}$

Tendo sido geradas as instâncias para cada *ReAgente*, o grafo pode ser criado, que irá seguir a seguinte ordem de inclusão de instâncias: (7, 8), (1, 2), (6, 1) (3, 7), (2, 3), (4, 5), (5, 4), (6, 8). Onde as instâncias em vermelho referem-se àquelas da segunda reação (P2 - *ReAgente* R2). A Figura 4.14 ilustra a geração do Grafo, já contendo uma orientação acíclica conforme mencionado anteriormente.

**Passo 3 - Grafo**:
Note que na Figura 4.14, o grafo contém instâncias de todas as reações que fazem parte do código Gamma, onde os vértices em vermelho correspondem àqueles formados por instâncias de execução da segunda reação. O passo seguinte (Passo 4) destina-se a identificação e execução dos *sinks*.

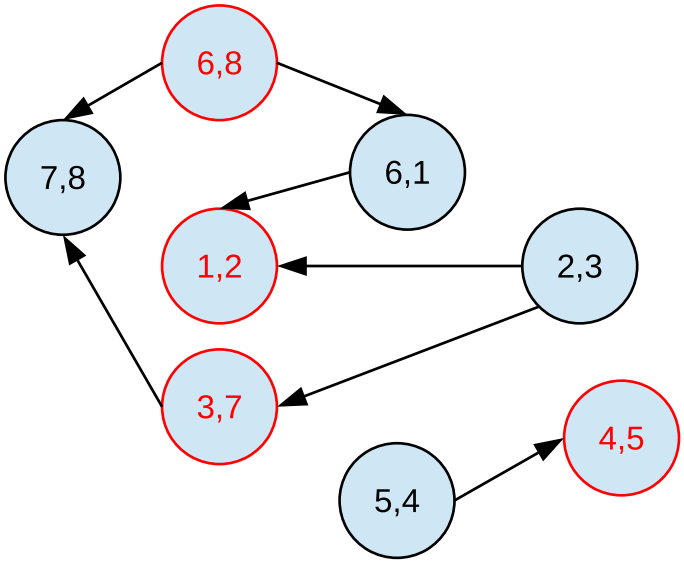


Figura 4.14: GSink - Exemplo - Passo 3.

**Passo 4 - Identificação e Execução dos *Sinks***:

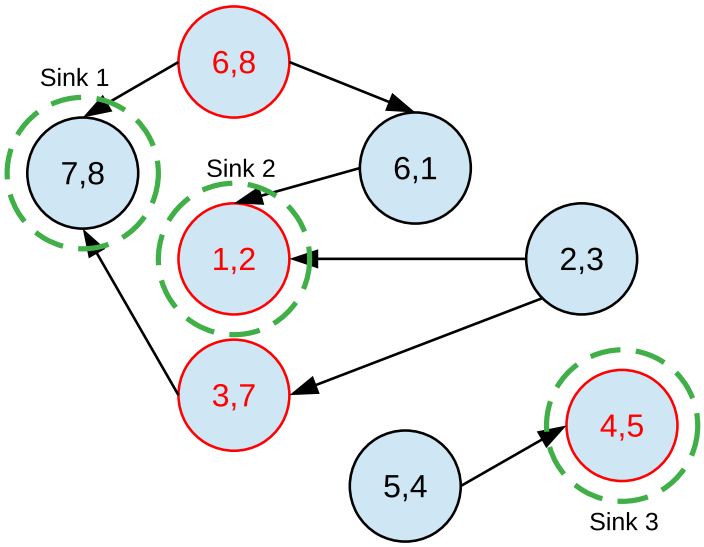


Figura 4.15: GSink - Exemplo - Passo 4.

A Figura 4.15 ilustra a identificação dos *sinks*. O primeiro *sink*, referente à reação P1, consegue reagir, já que seus elementos satisfazem a condição de reação de P1 ($(x > 4)$ *and* $(y > 4)$). Desta maneira, a execução do *Sink 1* retira os elementos 7 e

8 do multiconjunto e insere o elemento 15. O segundo *sink*, cuja instância pertence a P2, também consegue reagir, uma vez que $((x <= 4)\ and\ (y <= 4))$. Dessa maneira, os elementos 1 e 2 são retirados do multiconjunto e o elemento -1 é inserido. Por fim, o *Sink 3* não consegue reagir, pois a instância $(4, 5)$ não satisfaz a condição de reação de P2 e, portanto, nenhuma alteração no multiconjunto é imposta pelo terceiro *sink*. Desta forma pela tentativa de execução dos sinks em questão, temos que o multiconjunto passa a ser composto pelos seguintes elementos: $\{-1,\ 3,\ 4,\ 5,\ 6,\ 15\}$.

**Passo 5 - Reversão das Arestas dos *Sinks***:

Tendo identificado e executado os *sinks* no passo anterior, a Figura 4.16 apresenta, na cor verde, as arestas que foram revertidas de cada um dos três *sinks* abordados no passo 4.

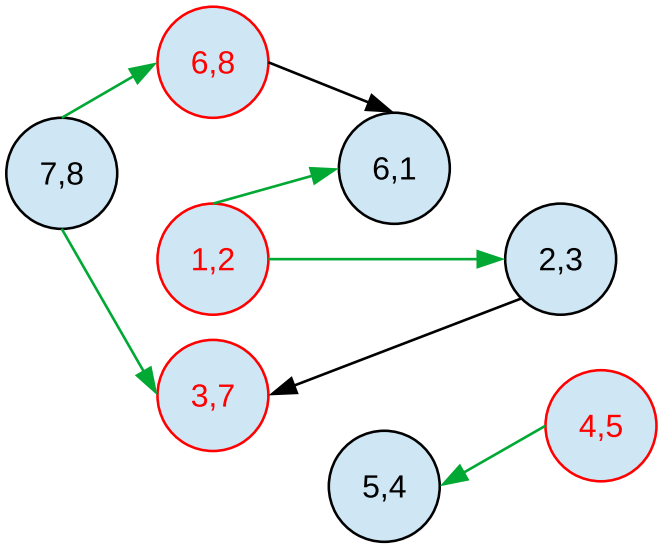


Figura 4.16: GSink - Exemplo - Passo 5.

**Passo 6 - Remoção de Vértices afetados pela execução dos *Sinks***:

A Figura 4.17 apresenta o grafo alterado com a remoção de todos os vértices que foram afetados pela execução de algum *sink*. Dessa maneira, verifique que, como o *Sink 1* (instância $(7, 8)$ da reação P1) reagiu, os vértices $(6, 8)$ e $(3, 7)$ foram afetados por tal execução, e por isso foram excluídos, uma vez que os elementos 7 e 8 do multiconjunto foram modificados (excluídos e substituídos por 15). Da mesma forma, o *Sink 2* reagiu, alterando através da execução de P2, os elementos 1 e 2, o que fez com que os vértices $(6, 1)$ e $(2, 3)$ também fossem excluídos. Por fim, o terceiro *sink* (instância $(4, 5)$ de P2) não reagiu. Por isso, o vértice $(5, 4)$ que compartilha recursos com este vértice, não precisou ser excluído, uma vez que seus elementos não foram modificados.

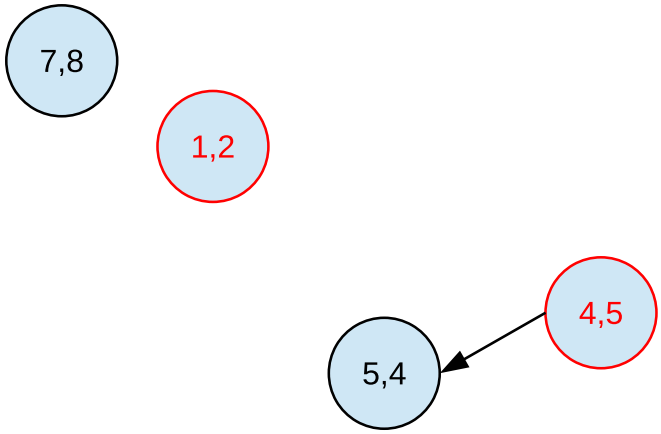


Figura 4.17: GSink - Exemplo - Passo 6.

**Passo 7 - Remoção dos *Sinks***:
Por fim, os *sinks* são removidos. Dessa forma, os vértices $(7, 8)$, $(1, 2)$ e $(4, 5)$ são removidos do grafo. Note que mesmo os *sinks* que não executaram são removidos nesta etapa, visando disponibilizar seus elementos (cuja combinação não foi útil para determinada reação) para uma próxima geração de instâncias, em outra *wave*. Dessa forma, após a remoção dos *sinks* o único vértice que continua no grafo é $(5, 4)$ referente a instância de P1.

Finalizado o passo 7, como ainda existem vértices no grafo (neste caso somente um vértice), voltamos ao passo 4 para uma nova identificação e execução de *sinks*. Neste caso específico, teremos um único *sink* que não irá reagir, pois não satisfaz a condição de reação de P1. Assim, ao seguirmos os passos de 4 a 7, este vértice não executará e por isso não irá impor nenhuma alteração ao multiconjunto. Dessa maneira, como não haverá mais vértices no grafo, uma nova *wave* será iniciada, onde o passo 1 será composto da seguinte maneira:

**(Segunda Wave) Passo 1 - Reagentes**:
$R1 - \{-1,\ 3,\ 4,\ 5,\ 6,\ 15\}$
$R2 - \{-1,\ 3,\ 4,\ 5,\ 6,\ 15\}$

Assim, a segunda *wave* irá conduzir os passos apresentados anteriormente até que o grafo gerado no passo 3 não possua mais vértices, pelas exclusões realizadas. Ocorrerão quantas *waves* forem necessárias até que se encontre um estado de terminação global. Tal estado é alcançado quando não existirem mais elementos a serem combinados para gerar instâncias (existência de um único elemento no multiconjunto em um programa com reações binárias, por exemplo) ou então, até que todas as combinações possíveis já tenham sido fornecidas como instâncias de execução.

#### 4.3.2.4 Benefícios e Contribuições

Conforme vimos anteriormente, as implementações de Gamma existentes apresentaram diversas questões que merecem investimento de pesquisa, a fim de que se tenha um ambiente robusto para a execução de programas Gamma. Por exemplo, as implementações *Gamma-Sequencial*, *Gamma-MPI* e *Gamma-GPU* não permitem a execução paralela de reações distintas nem de instâncias de uma mesma reação. Neste contexto, o *GSink*, contribui diretamente para o modelo computacional em questão, por permitir a coexistência e execução paralela de instâncias de várias reações, através da utilização de uma proposta de escalonamento baseada em [22].

De acordo com o que apresentaremos nas Seções seguintes, a implementação da execução dos *Sinks* utilizou *threads*. Tal abordagem permite a inserção de elementos de não determinismo no modelo, ao passo que não se pode controlar a ordem da execução destas estruturas, de acordo com o que veremos por ocasião da discussão de experimentos no Capítulo 5. Isso permite trazer uma característica implícita de Gamma que é um modelo não determinístico de execução, o que faz com que alguns autores o considerem um modelo com comportamento caótico [22].

Da mesma maneira que o *GFlow*, o *GSink* também contribui para a expressividade de Gamma, ao passo que além de expor o potencial do paradigma, esta pesquisa contribui com uma série de direcionamentos futuros. Tais indicações podem vir a tornam o modelo cada vez mais robusto e atraente para diversos domínios de aplicação.

Vale ressaltar que, até o momento do desenvolvimento deste trabalho, não foi encontrada nenhuma implementação de Gamma que permitisse a execução paralela de instâncias de várias reações ou de uma mesma reação. Desta forma, o GSink é a primeira proposta de implementação do paradigma Gamma com esta característica.

### 4.3.3 Considerações sobre a implementação

Após a exposição inicial da descrição da solução, esta Seção dedica-se a apresentar alguns detalhes mais específicos da implementação de nossa proposta de escalonamento para execução de instâncias de reações, o *GSink*.

Assim, iniciaremos pela apresentação de aspectos gerais a respeito das estruturas e do ambiente fornecido. Posteriormente, serão abordadas informações à respeito da implementação do conceito de *ReAgente*, suas estruturas e funcionalidades. Após isso, a implementação do grafo de instâncias de execução de reações será apresentado seguido do mecanismo de execução dos *sinks*. Por fim, apresentaremos a geração de código C a partir das alterações no *Front-End* das implementações anteriores. O Capítulo 5 apresentará os experimentos executados para o *GSink*.

#### 4.3.3.1 Aspectos Gerais

O *GSink* (https://gitlab.com/rui.rmj/gsink) é um ambiente de execução para programas Gamma baseado em [22], implementado em linguagem C. O projeto é composto de um *Front-End* que realiza análise léxica e sintática (Lex e Yacc). Posteriormente, a Seção 4.3.3.5 apresentará detalhes sobre o *Front-End*. Analisando a árvore sintática produzida, um código C é fornecido para execução no *Runtime* do *GSink*, contendo funções específicas que possibilitam o gerenciamento e execução das instâncias de reações. Em outras palavras, o *Front-End* é utilizado para gerar o código C que irá conter todas as funcionalidades necessárias para execução do algoritmo de escalonamento de execução do *GSink*.

Algumas estruturas de dados utilizadas merecem destaque, e serão mais detalhadas no decorrer das seções seguintes, sejam elas:

- **reg_vertex *vertices** - lista de listas. Consiste em uma lista encadeada que contém todos os vértices do grafo e, para cada vértice, uma lista de aresta de saída, onde é armazenado o id de cada vértice ao qual se possui alguma aresta de saída.
- **reg_multiset *multiset** - lista encadeada que contém os elementos do multiconjunto.
- **reg_I *instances_Ix** - lista encadeada que armazena as instâncias geradas por um determinado *ReAgente*.
- **reg_I *usedinstances_Ix** - lista encadeada que contém o histórico de instâncias já fornecidas por um dado *ReAgente*.
- **reg_multiset *multiset_Mx** - lista encadeada que contém uma cópia do multiconjunto para um *ReAgente* específico.
- **reg_sink *sinks** - lista encadeada que contém os *sinks* de uma determinada orientação acíclica.

O *Runtime* do *GSink* é composto por arquivos de cabeçalho (".h"), e arquivos com definições de funções (".c") disponibilizadas para permitir o escalonamento segundo nossa proposta. Abaixo estão listados os principais arquivos que compõem o ambiente de execução:

- **const.h** - Definição de Constantes.
- **fnc.h** - Declaração de funções.
- **types.h** - Definição de tipos e estruturas.

- **graph.c** - Definição de funções para manipulação de vértices e arestas do grafo, incluindo funções utilizadas para orientação acíclica do grafo (por ocasião da inclusão de vértices), entre outras.

- **misc.c** - Contém mecanismo de teste para funções implementadas. Apresenta um menu com possibilidade de teste das funções. Utilizada para fins de *Debug*.

- **multiset.c** - Definição de funções para manipulação dos elementos da lista que corresponde ao multiconjunto.

- **ReAgent.c** - Definição de funções para manipulação da lista de instâncias e da lista que corresponde à cópia do Multiconjunto gerada por cada *ReAgente*. Também engloba funções utilizadas na geração de instâncias.

- **sinks.c** - Definição de funções implementadas para manipular a lista de *sinks*, incluindo funcionalidades para identificação de *sinks*, reversão de arestas de *sinks*, entre outras.

- **main.c** - Contém o código principal com a utilização das funções implementadas anteriormente e que implementa o gerenciamento do grafo de instâncias e a execução do escalonamento da execução.

O código 4.8 abaixo apresenta o arquivo *fnc.h* contendo a declaração das funções disponibilizadas para o ambiente de execução do *GSink*:

```
1  reg_vertex * CreateVertex (void);
2  reg_vertex * AddVertex (int *id_vertex, reg_data data, int reaction);
3  reg_vertex * SearchVertex (reg_data data, int reaction);
4  void DeleteVertex(reg_data data, int reaction);
5  void DeleteVertexAllEdges(reg_data data, int reaction);
6  void DeleteVerticesAffected(reg_sink *ps);
7  reg_edge * CreateEdge (void);
8  reg_edge * AddEdge (reg_data data_v, int reaction_v, int id_edge);
9  reg_edge * SearchEdge (reg_data data_v, int reaction_v, int id_edge);
10 void CreateOutputEdges (reg_data data_v, int reaction_v);
11 void DeleteEdge (reg_data data_v, int reaction_v, int id_edge);
12 void DeleteAllEdges (reg_data data_v, int reaction_v);
13 void ListingGraph(void);
14 void IdentifySinks (void);
15 reg_sink * CreateSink (void);
16 reg_sink * AddSink (reg_data data, int id, int reaction);
17 void ListingSinks(void);
18 void DeleteAllSinks (void);
19 void EdgesReversalSink (reg_sink *ps);
20 reg_multiset * CreateMultisetElement (void);
21 reg_multiset * AddMultisetElement (int *id_element, int element);
22 void ListingMultiset (void);
23 reg_multiset * SearchMultisetElement (int element);
24 void DeleteMultisetElement (int element);
25 void DeleteMultisetElementId (int id);
26 void DeleteAllMultiset (void);
27 reg_I * CreateInstanceIx (void);
```

```
28 reg_I * AddInstanceIx (reg_I **instances_Ix, reg_I element_Ix);
29 reg_I * SearchInstanceIx (reg_I **instances_Ix, reg_I element_Ix);
30 reg_I * SearchInstanceIxId (reg_I **instances_Ix, reg_I element_Ix);
31 void DeleteInstanceIx (reg_I **instances_Ix, reg_I element_Ix);
32 void DeleteInstanceIxId (reg_I **instances_Ix, reg_I element_Ix);
33 void DeleteAllInstanceIx (reg_I **instances_Ix);
34 void ListingInstancesIx (reg_I *instances_Ix);
35 reg_multiset * CreateElementMx (void);
36 reg_multiset *AddElementMx (reg_multiset **multiset_Mx, reg_multiset element_Mx);
37 reg_multiset * SearchElementIdMx (reg_multiset **multiset_Mx, int id);
38 reg_multiset * SearchElementValueMx (reg_multiset **multiset_Mx, int
       element_value);
39 void DeleteElementIdMx(reg_multiset **multiset_Mx, int id);
40 void DeleteElementValueMx(reg_multiset **multiset_Mx, int element_value);
41 void DeleteAllElementsMx (reg_multiset **multiset_Mx);
42 void CopyMultisetToMx (reg_multiset **multiset_Mx);
43 void ListingMx (reg_multiset *multiset_Mx);
44 void GeneratesInstancesRx (reg_multiset **multiset_Mx, reg_I **instances_Ix,
       reg_I **usedinstances_Ix);
45 void * ThreadFunction (void *arg);
46 int R1Function(reg_sink *ps);
47 int R2Function(reg_sink *ps);
```

Código 4.8: fnc.h - Declaração das funções do *Runtime* do *GSink*.

O algoritmo 3 descreve as etapas genéricas de processamento do ambiente de execução do *GSink*.

Inicialmente, após definições e inicializações de variáveis, é criado o Multiconjunto inicial, através da utilização de funções disponibilizadas para inserção de elementos no multiconjunto (lista encadeada que representa o multiconjunto). Em seguida, para cada *ReAgente* (lembrando que existe um *ReAgente* para cada reação) é criada uma lista de instâncias, onde uma cópia do multiconjunto é criada para cada *ReAgente* e, a partir disso, combinações válidas de elementos são geradas para formar instâncias de execução. Tais combinações são armazenadas em uma lista de instâncias.

Na linha 3 do Algoritmo 3, existe um loop que será executado enquanto existirem instâncias. Em outras palavras, enquanto for possível gerar instâncias de execução de reações, uma nova *wave* será inicializada neste ponto. O *loop* seguinte executa, para cada *ReAgente*, a inserção de cada instância da lista de instâncias como vértice do grafo $G$. Note que no momento da inserção de uma instância como vértice de $G$ e verificado se este compartilha recursos com demais vértices pré-existentes em $G$. Caso isso seja verdade, é criada uma aresta cujo sentido será sempre a favor do vértice já existente no grafo. Conforme já mencionamos, isso faz com que todo vértice inserido só crie arestas de saída, garantindo a orientação acíclica necessária ao *SER* [78]. Ainda por ocasião da inserção de uma instância como vértice de $G$, também é gerada uma entrada no histórico de instâncias fornecidas (reg_I *usedinstances_Ix), visando controlar a geração de futuras instâncias, como veremos posteriormente.

Após a criação do grafo $G$, os *sinks* desta primeira orientação acíclica são identi-

**Algorithm 3:** GSink - Escalonamento de execução de instâncias de reação.

```
1  Criação do Multiconjunto Inicial;
2  Geração Inicial de Instâncias (para cada ReAgente);
3  while existirem instâncias do
4      foreach ReAgente do
5          foreach instância da lista de instâncias do
6              Insere instância como um vértice em G;
7              Cria aresta para os nós mais antigos (caso necessite);
8              Adiciona instância utilizada em Histórico;
9          end
10     end
11     Identifica Sinks em G;
12     while existirem vértices em G do
13         foreach Sink do
14             Tenta executar reação;
15             if reação reagiu then
16                 altera elementos no multiconjunto;
17             end
18             Reverte arestas do Sink;
19             if reação reagiu then
20                 Remove nós afetados pela execução do Sink;
21             end
22             Remove o Sink;
23         end
24         Identifica novos Sinks;
25     end
26     Geração de novas Instâncias;
27 end
```

ficados. Assim, na linha 12, outro *loop* inicia-se, onde os *sinks* vão sendo executados conforme apresentamos na Seção 4.3.2.2. Ou seja, enquanto existirem vértices em $G$, sempre existirá pelo menos um *sink* [78], portanto, para cada *sink* é disparada uma *thread* responsável pela execução no mesmo. Assim, ocorre a tentativa de execução da instância da referida reação (lembrando que aqui coexistem instâncias de várias reações). Caso a condição de reação seja satisfeita, a ação é aplicada, ou seja, o multiconjunto é alterado. Em seguida o *sink* reverte suas arestas e verifica se precisa remover vértices que porventura tenham sido afetados pela sua execução. Por fim o próprio *sink* é removido do grafo e ocorre uma tentativa de identificação de novos *sinks* para início de uma nova orientação acíclica.

Caso não existam mais vértices em $G$, ocorre uma tentativa de uma nova geração de instâncias, na linha 26 do Algoritmo 3. Caso seja possível gerar novas instâncias, uma nova *wave* inicia-se. Caso contrário o algoritmo termina. A Figura 4.18 ilustra as etapas descritas acima.

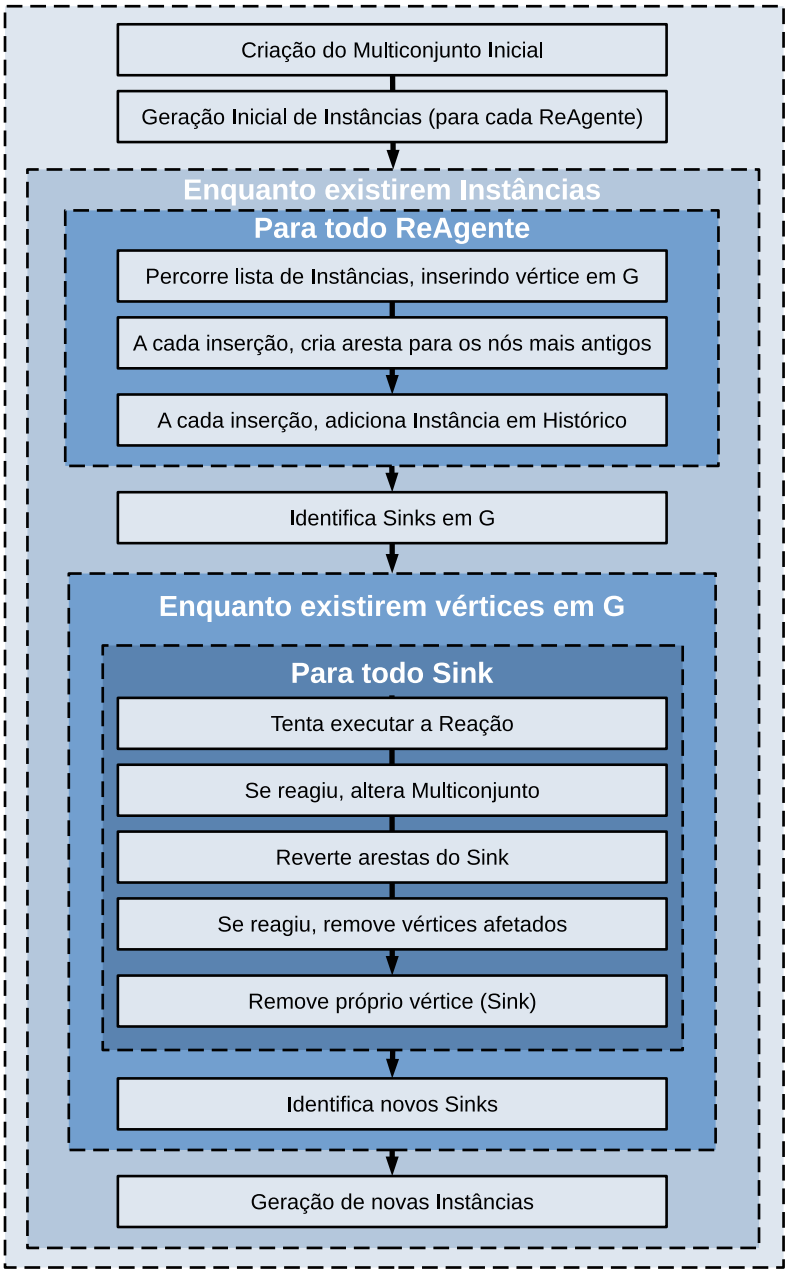


Figura 4.18: GSink - Escalonamento de execução de instâncias de reação.

#### 4.3.3.2 ReAgentes

Conforme proposto em [22], tendo em vista a gerência de criação e fornecimento de instâncias de cada reação que compõe o código Gamma, são usados agentes de reações (*ReAgentes*). No *GSink* implementamos o conceito de *ReAgentes* através de algumas estruturas de dados, citadas anteriormente, responsáveis por: armazenamento das instâncias criadas para cada *ReAgente* (*reg_I *instances_Ix*), armazenamento do histórico das instâncias já fornecidas (*reg_I *usedinstances_Ix*) como vértices para o Grafo $G$ por determinado *ReAgente* e armazenamento de uma cópia de trabalho do multiconjunto (*reg_multiset *multiset_Mx*), para cada *ReAgente*.

Dentre as funcionalidades desempenhadas por um *ReAgente* destaca-se a geração de instâncias. Para tanto, inicialmente efetua-se, no âmbito de cada *ReAgente*, uma cópia do multiconjunto geral. Diante desta cópia, cada *ReAgente* vai gerar sua própria lista de instâncias, onde posteriormente cada instância passará a ser um vértice do grafo $G$. A lista encadeada que representa a cópia do multiconjunto é percorrida por dois ponteiros, *first* e *second* de forma que o primeiro é fixado até que o segundo percorra toda a lista. A cada posicionamento dos ponteiros ocorre uma tentativa de geração de instâncias, (*first*, *second*) e, antes de mover o segundo ponteiro, a tentativa (*second*, *first*) também é realizada. Se a instância puder ser gerada, tais elementos são excluídos da cópia do multiconjunto e a instância é inserida na lista de instâncias. Caso nenhuma das duas tentativas sejam válidas, o segundo ponteiro é movimentado e o processo se repete. Se o segundo ponteiro percorre a lista inteira, o primeiro ponteiro é atualizado para a próxima posição da lista e todo o processo é repetido novamente. Note que, a cada tentativa de geração de instâncias, é verificado se a mesma já não foi fornecida anteriormente, através da consulta à lista que representa o histórico de instâncias. Assim, o fator impeditivo de geração de uma instância é a mesma já ter sido ofertada como instância anteriormente.

É importante mencionar que, a geração de instâncias leva em consideração o identificador de cada elemento do multiconjunto e não seu valor, além da ordem destes elementos. Ao passo em que a definição do multiconjunto permite a repetição de elementos, foi necessário armazenar, além do valor, um identificador inteiro unívoco de cada elemento. Repare que além do multiconjunto inicial permitir a repetição de elementos, as alterações dos elementos em tempo de execução da aplicação, podem vir a inserir no multiconjunto cópias de elementos pré-existentes. Assim, a geração das instâncias leva em consideração os identificadores ao invés dos valores dos elementos. A Figura 4.19 ilustra o processo de geração de instâncias por cada *ReAgente*.

#### 4.3.3.3 O Grafo

As instâncias geradas pelos *ReAgentes* serão inseridas como vértices de um grafo (denominado de Grafo $G$), por meio do qual futuramente serão executadas as reações Gamma referentes à tais instâncias. Assim, os vértices deste grafo representam as instâncias de execução das reações, enquanto que as arestas representam o compartilhamento de recursos (elementos do multiconjunto) entre os vértices. O grafo foi implementado como uma lista de adjacência, onde uma lista encadeada contém os vértices do grafo e, cada elemento desta lista aponta para uma outra lista encadeada, contendo a identificação de outros nós, representando as arestas de saída deste vértice. A Figura 4.20 ilustra o Grafo $G$ com seus vértices e arestas.

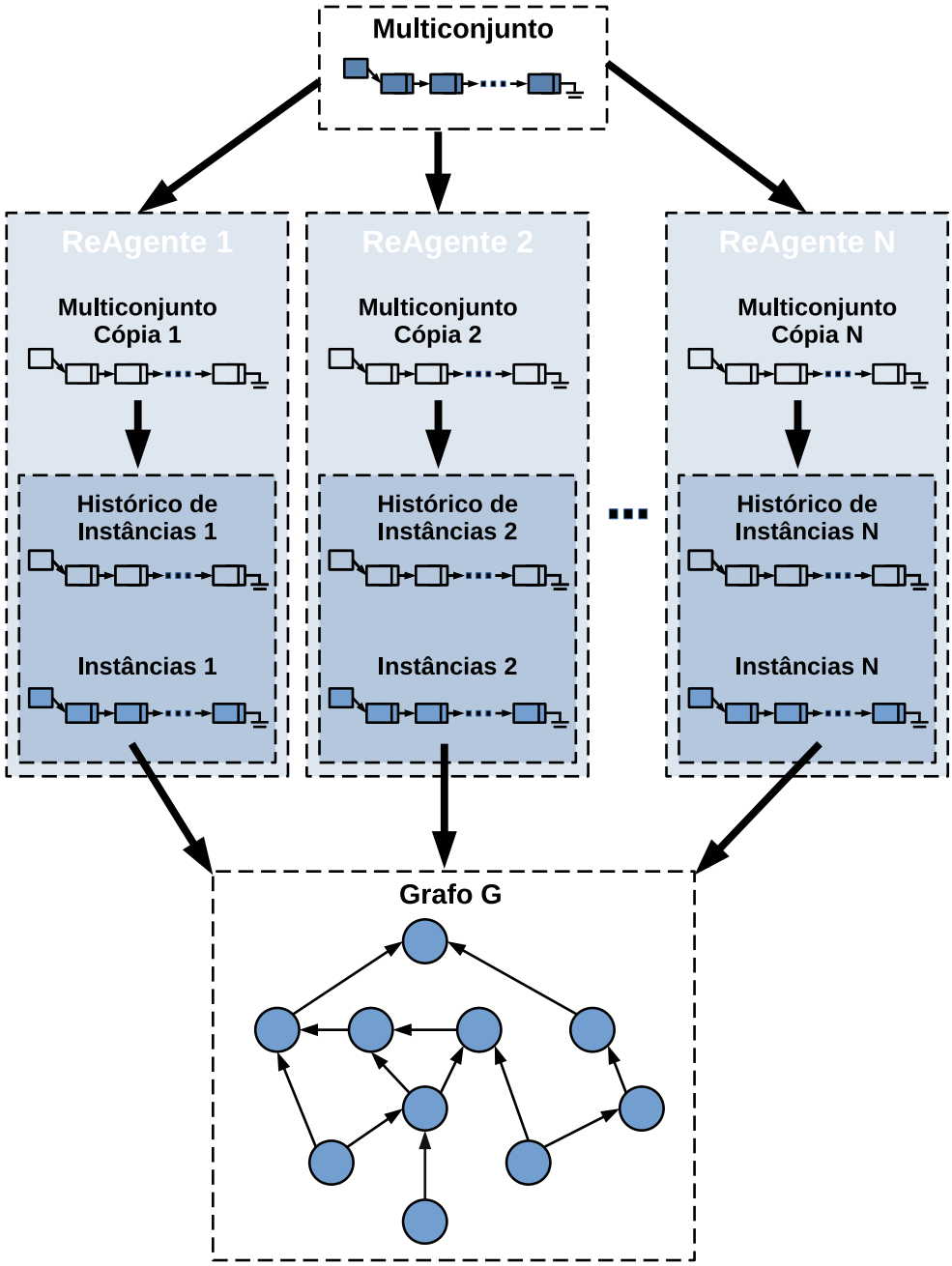


Figura 4.19: GSink - Geração de Instâncias por *ReAgente*.

Na Figura 4.20 temos um exemplo de um grafo $G$ à direita, e sua representação como lista de adjacências à esquerda. Os vértices estão contidos em uma lista encadeada onde cada elemento possui um ponteiro para outra lista, indicando os índices dos vértices aos quais tem-se arestas de saída. Em nossa implementação, cada vértice (do tipo *reg_ vertex*), armazena informações a respeito dos elementos que formam a instância de execução (incluindo um *Id* de cada elemento), assim como um identificador do vértice, um identificador da reação ao qual pertence e um ponteiro para a lista de arestas. Já a lista de arestas (*reg_ edge*) armazena o

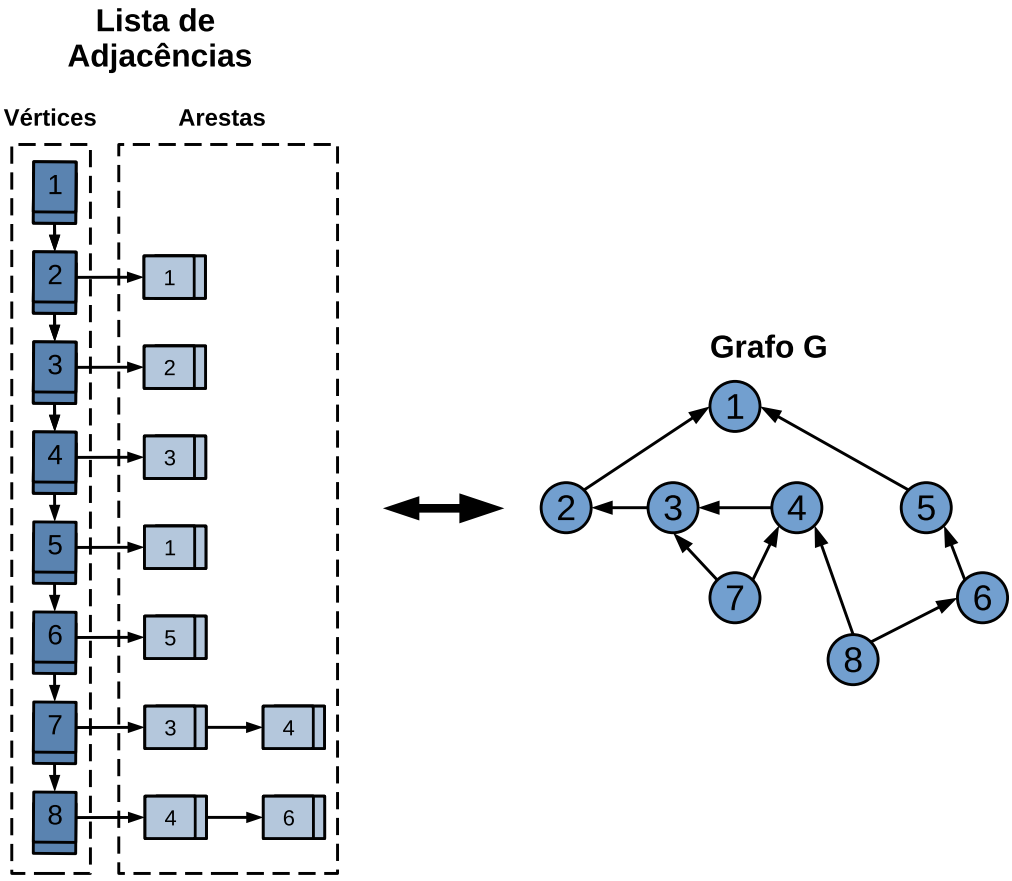


Figura 4.20: GSink - Representação do Grafo como Lista de Adjacências.

identificador do vértice e um ponteiro para o próximo elemento.

Note ainda que na Figura 4.20 só existe um vértice que não possui nenhuma aresta de saída, o vértice cujo identificador é igual a 1. Este seria o único *sink* desta primeira orientação acíclica.

Conforme mencionamos anteriormente, a medida em que os vértices vão sendo incluídos no grafo, verifica-se se existe compartilhamento de recursos com outros vértices. Caso positivo, é criada uma aresta cujo sentido será sempre a favor do vértice pré-existente, visando a garantia da orientação acíclica do grafo.

#### 4.3.3.4 Sinks

Os *sinks* representam os vértices do grafo cujas arestas estão todas direcionadas a seu favor, ou seja, tal tipo de vértice não possui arestas de saída. Em nosso grafo de instâncias de execução de reações, cada vértice está relacionado a uma instância específica de alguma reação do programa Gamma. Como as arestas representam o compartilhamento de recursos entre estes vértices, é correto afirmar que os *sinks* de uma mesma orientação acíclica não compartilham recursos entre si. Caso compartilhassem recursos, existiria pelo menos uma aresta entre estes nós e, como este é um

grafo dirigido, tal aresta teria um sentido, fazendo com que um destes dois vértices possuísse uma aresta de saída, não caracterizando-o como um *sink*. Assim, é correto afirmar que *sinks* de uma mesma orientação acíclica não compartilham recursos e, portanto, podem ser executados em paralelo.

Em nossa implementação, utilizamos uma lista encadeada (*reg_ sink *sinks*) para armazenar os *sinks* identificados em alguma orientação acíclica. Tal identificação ocorre nas linhas 11 e 24 do Algoritmo 3. Sobre cada *sink* identificado, armazenamos informações a respeito dos elementos que formam a respectiva instância de execução (valores e *id* destes elementos), identificador do vértice, identificador da reação ao qual a referida instância pertence, além de um ponteiro para o próximo *sink* da lista.

A identificação dos *Sinks* ocorre através da verificação, na lista de adjacências que representa o grafo de instâncias de reações, de qual vértice não possui arestas de saída. Ou seja, em uma lista de adjacência o vértice cujo ponteiro para lista de arestas contiver valor *NULL*, será considerado um *sink*. Uma vez identificado um *sink*, o mesmo será inserido na lista de *sinks*. Dessa maneira a etapa de identificação de sinks preenche uma lista encadeada com todos os *sinks* da orientação acíclica atual.

Assim, uma vez identificados os *sinks* de determinada orientação acíclica, os mesmos podem executar. O mecanismo de execução de instâncias de reações dispara uma *thread* para cada *sink*. Inicialmente cada *sink* tenta realizar a operação (reação), através do teste da condição de reação. Caso o teste condicional seja verdadeiro, o cálculo computacional é realizado e o multiconjunto deve ser alterado para refletir tais modificações. Neste momento, é definida uma região crítica, onde as *threads* modificam o multiconjunto e liberam a região crítica, implementada a partir de semáforos. É importante mencionar que, para cada reação existente no código, o *Front-End* implementa uma função específica, para ser executada pela *trhread* que irá operar tal instância de determinada reação. Maiores informações serão apresentadas na Seção seguinte.

Após a tentativa de execução de cada *sink*, somente aqueles que conseguiram reagir modificam o multiconjunto. Reagindo ou não, cada *sink* necessita reverter suas arestas, remover nós que por ventura tenham sido afetados pela modificação dos elementos que compõem a instância deste *sink* e, por fim, remover o vértice do grafo que corresponde a este *sink*. Tais tarefas visam atualizar o grafo e gerar uma nova orientação acíclica para posterior identificação de *sinks*. Dessa maneira, uma nova região crítica é definida, agora para permitir as atualizações necessárias no grafo, realizadas por cada *sink* que está procedendo com seu mecanismo de execução. Repare que é necessário o estabelecimento de tais regiões críticas, uma vez que várias *threads* tentam acessar e modificar ponteiros de estruturas implementadas com alocação dinâmica de memória. A não utilização deste recurso para evitar

condições de corrida pode levar a perda de referência de ponteiros, ocasionando erros no programa em questão. A Figura 4.21 ilustra de maneira genérica a identificação de *sinks* e a execução dos mesmos tendo em vista as regiões críticas estabelecidas.

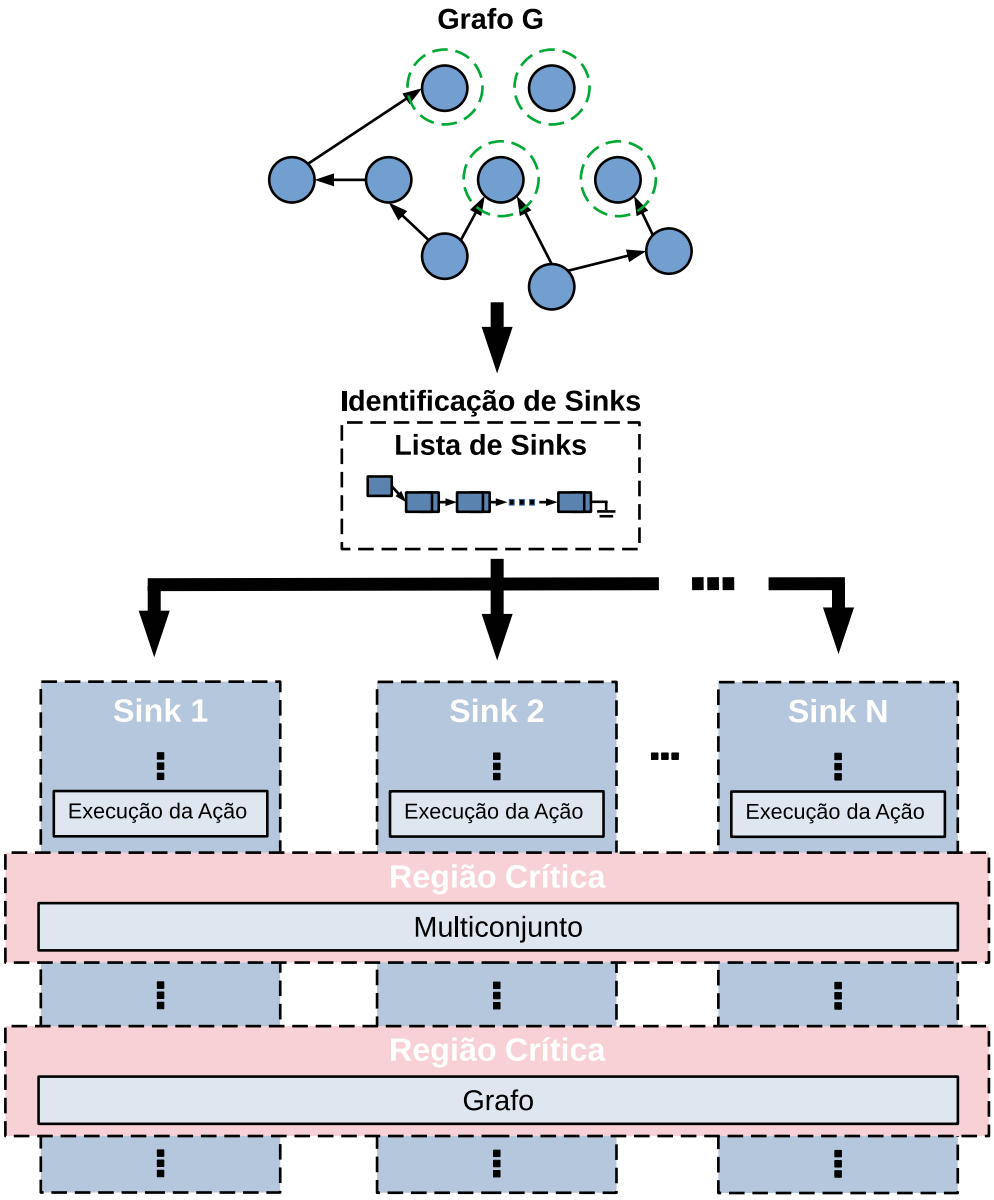


Figura 4.21: GSink - Execução dos *Sinks*.

Conforme mencionamos anteriormente, a identificação dos *sinks* na lista de adjacências se dá através da detecção dos vértices cujos ponteiros para lista de arestas de saída esteja apontando para *NULL*. Assim, o procedimento para reversão de arestas busca, para um vértice $s$ que é *sink*, se o mesmo é utilizado como aresta por outro vértice $v$. Caso positivo, exclui-se a aresta de saída com identificador de $s$ em $v$ e cria-se uma aresta para $s$, com o identificador de $v$.

#### 4.3.3.5 *Front-End* da Implementação *Gamma-Sequencial*

A implementação Gamma-Sequencial é composta de um *Front-End*, responsável pela análise léxica, sintática e geração da árvore sintática e de um *Runtime*, onde o código C (*"run.c"*) gerado pelo *Front-End* será executado de maneira sequencial, cujo escalonamento discutimos na Seção 4.3.1 e no Capítulo 2.

O *GSink* utiliza a análise léxica e sintática da implementação *Gamma-Sequencial* que tem como entrada um arquivo *".gm"*. Dessa maneira, foram implementadas algumas funções visando fornecer o arquivo "main.c", a ser executado em nosso ambiente de execução do *GSink*.

O código fonte abaixo (Código 4.9) apresenta a declaração das funções utilizadas no *Front-End* da implementação *Gamma-Sequencial*, com destaque para as funções fornecidas por ocasião da implementação do *GSink* (após comentário da linha 47).

```
1  void PrintNumNode ();
2  void PrintBoolNode ();
3  void PrintIdentNode ();
4  void PrintExpression ();
5  void PrintListDefinitions ();
6  void PrintBody ();
7  void PrintBasicBody ();
8  void PrintCombinatorBody ();
9  void PrintPattern ();
10 void PrintAction ();
11 void PrintBag ();
12 void PrintTuple ();
13 char *Alloc ();
14 void Free ();
15 RT_BAG *CreateInitialBagNode ();
16 void DeleteRTBagNode ();
17 RT_BAG *CreatRTBagNode ();
18 RT_BAG *AddRTBagNode ();
19 RT_BAG *AddRTTupleBagNode ();
20 RT_BAG *SearchRTBagInteger ();
21 void UnlockBagNodes ();
22 RT_BAG *SearchTuple ();
23 int TuplesMatch ();
24 RT_TUPLE *GenTupleSkeleton ();
25 NAMELIST *NamesInExpression ();
26 void CodePrintNumNode ();
27 void CodePrintBoolNode ();
28 void CodePrintIdentNode ();
29 void CodePrintExpression ();
30 void CodePrintListDefinitions ();
31 void CodePrintBody ();
32 void CodePrintBasicBody ();
```

```
33 void CodePrintCombinatorBody ();
34 void CodePrintPattern ();
35 void CodePrintAction ();
36 void CodePrintBag ();
37 void CodePrintTuple ();
38 void CodePrintProgram ();
39 void WriteHeader ();
40 void CodeInitialBag ();
41 void GenBasicExpression ();
42 int GetTupleSize ();
43 void GenTuple ();
44 void GenCodeExpression ();
45 void GenCodeReaction ();
46 void GenCodeBagIntSearch ();
47 //New Scheduler Functions (NSGB - New Scheduler Graph-Based)
48 void NSGBWriteHeader ();
49 void NSGBWriteGlobalVariables ();
50 void NSGBGenInitialMultiset ();
51 void NSGBLocalVarsInicialization ();
52 void NSGBInstancesGeneration ();
53 void NSGBMainLoop ();
54 void NSGBGenDeclarationsReactions ();
55 void NSGBGenCodeReactions ();
56 void NSGBCodePrintNumNode ();
57 void NSGBCodePrintBoolNode ();
58 void NSGBCodePrintIdentNode ();
59 void NSGBCodePrintPattern ();
60 void NSGBCodePrintExpression ();
```

Código 4.9: Declaração das funções do *Front-End* da implementação *Gamma-Sequencial*.

Assim, fornecemos funções responsáveis pela a montagem do *"main.c"*. Tais funções garantem a correta montagem de todo o código, contendo o *Header*, com todas as inclusões de bibliotecas necessárias, inclusão de informações acerca de definição e inicialização de variáveis. O multiconjunto inicial é extraído através da estrutura de "BAG" (struct bag), que armazena os elementos do multiconjunto.

Através da consulta à arvore sintática, armazenada na estrutura "PROGRAM" (struct program) são extraídas informações como quantidade de reações do programa. Isso é especialmente importante para conhecer a quantidade de *Re-Agentes* a serem gerados no *"main.c"*. Além da quantidade de reações, também extraímos informações a respeito de cada reação, como por exemplo condição de reação (teste condicional utilizado para definir a aplicação da ação), ação (operação a ser realizada com os elementos do multiconjunto), nomes de variáveis, entre outros.

Dessa maneira, através da alteração do *Front-End* de *Gamma-Sequencial*, produzimos o código fonte para ser executado no *Runtime* do *GSink*, extraindo informações da árvore sintática fornecida por este. Tal alteração no *Front-End* permite a correta montagem do *"main.c"*, incluindo a correta geração das funções que implementam a execução das *threads*, responsáveis pela execução das instâncias de reações no *GSink*. Os experimentos do Capítulo 5 apresentaram informações a respeito dos códigos gerados.

## 4.4 Discussões

Este Capítulo apresentou informações a respeito de duas implementações fornecidas por ocasião desta pesquisa. A primeira delas, o *GFlow*, consiste em uma ferramenta de conversão dataflow - Gamma implementada em *Python*. A segunda implementação fornecida trata-se de um novo ambiente de execução de programas Gamma, escrito em linguagem C, o *GSink*.

A partir dos conhecimentos obtidos com o estudo da equivalência entre os modelos computacionais Gamma e dataflow, apresentados no Capítulo 3, surge o *GFlow*, a primeira proposta de ferramenta para conversão entre os modelos computacionais em questão. O *GFlow* permite explorar benefícios desenvolvidos para ambos os modelos, o que além de aumentar a expressividade do modelo Gamma, permite uma maior versatilidade no desenvolvimento de programas. Por ocasião da implementação do *GFlow*, não foi implementado suporte a conversão de super-instruções, uma vez que as implementações de Gamma utilizadas como ambientes de execução não permitem reações de granularidade grossa.

Tendo em vista as características visualizadas nas implementações anteriores do modelo Gamma, surgiu a motivação em fornecer uma implementação inicial de um ambiente de execução para programas Gamma que permita a execução e coexistência de instâncias de várias reações, o *GSink*. Consiste na primeira implementação de um ambiente de execução com estas funcionalidades. O investimento em desenvolver tal ambiente de execução também contribui para o aumento da expressividade de Gamma, uma vez que este estudo apresentará direcionamentos futuros para melhoria de desempenho e identificará o potencial de Gamma em diversos aspectos. Por ocasião do desenvolvimento do *GSink*, tendo em vista o foco na implementação do gerenciamento de execução através de um grafo de instâncias, não foi implementado suporte a tuplas e utilizamos somente reações binárias.

Vale ressaltar que ambas implementações contribuem para um direcionamento de um modelo a ser futuramente utilizado como modelo computacional que permeia a rede pela utilização dos recursos disponíveis. Nossa proposta inicial de arquitetura utilizava um ambiente de execução para códigos Gamma baseado em um grafo da-

taflow (Apêndice C) e que utilizava protocolo de comunicação baseado em interesse. Entretanto, algumas características da atual implementação do protocolo de comunicação identificado, a *Radnet*, postergaram essa proposta e nos atentaram para o fato de que investir em Gamma poderia ser interessante. Assim, A ferramenta de conversão de modelos, o *GFlow*, e nosso ambiente de execução, o *GSink*, convergem para investimentos diretos no modelo Gamma, o que identifica um horizonte promissor para a utilização futura deste modelo no contexto da *Fluid Computing* e em áreas de interesse como Computação Heterogênea.

# Capítulo 5

# Experimentos e Resultados

No Capítulo anterior, foram apresentadas as características das soluções propostas nesta tese, seja para conversão entre os modelos computacionais, seja como ambiente de execução de programas Gamma. O presente Capítulo tem por objetivo apresentar os experimentos desenvolvidos além de expor os resultados obtidos e a discussão destes resultados. Dessa forma, apresentaremos alguns exemplos de aplicação visando discutir questões de implementação necessárias tanto à corretude quanto ao potencial de extração de desempenho.

Inicialmente apresentaremos os experimentos relacionados ao *GFlow*, onde o foco dos casos de teste pretende verificar a corretude da conversão proposta entre os modelos computacionais envolvidos. Para tanto, duas categorias de experimentos foram executados, conforme será apresentado na Seção 5.1.

Posteriormente, na Seção 5.2, serão apresentados experimentos para nossa proposta de ambiente de execução para aplicações Gamma, o *GSink*. Tais experimentos visam avaliar a corretude das respostas da execução das aplicações, comparadas a outras implementações do paradigma Gamma. Além disso, também demonstramos o potencial de extração de desempenho aliado à proposta de escalonamento utilizada pelo *GSink*, através de alguns experimentos.

Para os casos de teste realizados, a quantidade de vezes em que cada experimento foi executado variou de 10 a 30, visando diminuir o desvio padrão [61] e [18].

Vale ressaltar que para todos os experimentos deste Capítulo, sejam referentes ao *GFlow* ou ao *GSink*, existe um repositório público, contendo os códigos fontes de tais experimentos em: https://gitlab.com/rui.rmj/dsc-experimentos .

Foram utilizadas duas máquinas hospedeiras para a realização dos experimentos. A primeira delas possui um processador Intel® Core™ 2 Duo P8400 (2.26GHz) com 4GB de memória DDR2 400MHz, com o Sistema Operacional Linux, distribuição Debian 9 Stretch. Já a segunda máquina possui um processador Intel® Core™ i7-6700HQ (2.60Hz) com 16GB de memória DDR3 800MHz, sistema operacional Windows 10 Home.

## 5.1 GFlow

### 5.1.1 Procedimentos Experimentais

Utilizamos uma metodologia empírica para conduzir os experimentos do *GFlow*, na qual o objetivo foi discutir e apresentar a corretude das conversões propostas. Para tanto, foram elencados dois grupos de experimentos distintos.

No primeiro grupo de experimentos, foram escritas manualmente aplicações em linguagem de montagem do *TALM* sem a utilização de sua linguagem de alto nível, o *THLL*. Dessa maneira, este primeiro grupo de experimentos não utilizou o *Couillard* para gerar a linguagem de montagem do *TALM*, uma vez que queríamos ter maior controle das instruções constantes no código *TALM*.

Já para o segundo grupo de experimentos, utilizou-se o *Couillard* para gerar linguagem de montagem do *TALM* a partir de um código inicial escrito em C, com anotações de blocos em *THLL*. Esse código em TALM foi então submetido à conversão pelo *GFlow* gerando código Gamma equivalente. Para esse grupo de experimentos, o objetivo foi analisar a corretude da resposta da execução do código C inicial comparado à execução do código Gamma equivalente executado em alguma implementação do paradigma Gamma. Dessa maneira, utilizamos as implementações *Gamma-Sequencial* [18, 30] e *Gamma-MPI* [18, 30] fornecidas por Juarez Muylaert.

Para os casos de teste idealizados para o *GFlow*, a quantidade de vezes em que cada experimento foi executado foi de 10. Isso deve-se ao fato de que não houve variação do resultado dos experimentos, por tratar-se de análise de conversões e, mesmo no segundo grupo de experimentos, o resultado obtido é determinístico ao passo que não foram realizadas análises de desempenho.

### 5.1.2 Conversões a partir do *TALM*

Este é a primeira categoria de experimentos idealizados para o *GFlow*. Trata-se da análise e verificação de detalhes das conversões entre os modelos computacionais envolvidos. Desta forma, apresentaremos informações à respeito da listagem de reações, montagem do multiconjunto inicial e definição das reações que compõem o código Gamma equivalente. Tendo em vista o que apresentamos no Capítulo 4, em sua Seção 4.2.3.3, alguns conjuntos de instruções apresentam comportamentos semelhantes, do ponto de vista de sua conversão em reações Gamma. Assim, para a execução destes testes, foram agrupadas instruções aritméticas e suas variações com operando imediato, instruções lógicas e suas variações com operando imediato, instruções *Steer* e *Inctag*. Por fim, foi construído um experimento contendo instruções de todos os grupos mencionados.

Os nomes (*strings*) atribuídos aos registradores de saída pelo *Couillard* possuem um padrão, onde inicia-se sempre pela *string "flowInst"* concatenada à uma descrição do tipo de instrução, como *"Const"* (para constantes), *"Binop"* (instruções com dois operandos), *"Steer"* (instruções para desvios), *"Inctag"* (para instruções utilizadas em *Loops*), entre outros. Por fim, um valor inteiro é concatenado visando a identificação unívoca de determinado registrador de saída, como por exemplo: *"flowInstConst140451325315120"*. Entretanto, como as aplicações foram escritas manualmente, as *strings* que representam os registradores de saída possuem formato mais simples, exclusivamente para identificar univocamente cada instrução, sem perda de generalidade do código.

#### 5.1.2.1 Experimento 1 - Instruções Aritméticas

Para este primeiro experimento, considere o seguinte exemplo (Código 5.1), em linguagem de montagem do *TALM*:

```
1  const instconst1, 10
2  const instconst2, 20
3  const instconst3, 30
4  const instconst4, 40
5  const instconst5, 50
6  add instadd, instconst1, instconst2
7  addi instaddi, instconst1, 25
8  sub instsub, [instadd, instconst2], instconst3
9  div instdiv, instconst4, instconst5
10 const instconst6, 60
11 subi instsubi, [instconst1, instconst2], 335
12 divi instdivi, instconst4, 45
13 mult instmult, instadd, instconst2
14 mod instmod, [instconst6, instadd], [instconst3, instconst4]
15 and instand, instconst1, instconst3
16 modi instmodi, [instconst1, instsubi], 15
17 or instor, instconst6, instconst2
18 add instadd2, [instconst6, instconst2], [instconst1, instaddi]
19 div instdiv2, instconst1, [instconst5, instconst2]
20 divi instdivi2, instaddi, 45
```

Código 5.1: Experimento 1 - Linguagem de Montagem do *TALM*.

Repare que na aplicação acima possuímos exemplos de instruções aritméticas suportadas pelo *TALM* e suas variações com operadores imediatos (por exemplo, instruções das linhas 7, 11, 16, entre outras), conforme apresentado na Tabela 4.1. Também estão presentes no código em referência 6 constantes listadas entre as linhas 1 a 5 e na linha 10. Note também que, o registrador de saída de algumas instruções

é utilizado várias vezes como operando de outras instruções, como é o caso da instrução de soma contida na linha 6 do código, cujo registrador de saída é igual a *"instadd"*. Esta instrução especificamente foi utilizada como operando de outras três instruções (linhas 8, 13 e 14), o que significa que esta instrução deverá produzir três novos elementos no momento que sua reação equivalente for executada. Por fim, é importante notar que as 6 constantes existentes deverão ser replicadas, tendo em vista a quantidade de vezes em que seu registrador de saída foi utilizado como operando de alguma instrução. Por exemplo, a constante definida na primeira linha de código, cujo registrador de saída é igual a *"instconst1"* é utilizada por 7 instruções, definidas nas linhas 6, 7, 11, 15, 16, 18 e 19. Agora iremos analisar detalhes do código Gamma equivalente convertido através do *GFlow*, conforme apresentado no Código 5.2 abaixo:

```
1 R_instadd | R_instaddi | R_instsub | R_instdiv | R_instsubi |
     R_instdivi | R_instmult | R_instmod | R_instand | R_instmodi |
      R_instor | R_instadd2 | R_instdiv2 | R_instdivi2 {[10, 120,
     0], [10, 100, 0], [10, 80, 0], [10, 60, 0], [10, 40, 0], [10,
     20, 0], [10, 0, 0], [30, 42, 0], [30, 22, 0], [30, 2, 0], [20,
      121, 0], [20, 101, 0], [20, 81, 0], [20, 61, 0], [20, 41, 0],
      [20, 21, 0], [20, 1, 0], [50, 24, 0], [50, 4, 0], [40, 43,
     0], [40, 23, 0], [40, 3, 0], [60, 49, 0], [60, 29, 0], [60, 9,
      0]}
2
3 where
4
5 R_instadd = replace [x, 120, tag], [y, 121, tag1]
6 by [x + y, 5, tag], [x + y, 45, tag], [x + y, 25, tag]
7 if (tag == tag1)
8
9 R_instaddi = replace [x, 100, tag]
10 by [x + 25, 6, tag], [x + 25, 26, tag]
11 if true
12
13 R_instsub = replace [x, w, tag], [y, 42, tag1]
14 by [x - y, 7, tag]
15 if (tag == tag1) and (w == 45 or w == 101)
16
17 R_instdiv = replace [x, 43, tag], [y, 24, tag1]
18 by [x / y, 8, tag]
19 if (tag == tag1)
20
21 R_instsubi = replace [x, w, tag]
22 by [x - 335, 10, tag]
23 if (w == 80 or w == 81)
24
```

```
R_instdivi = replace [x, 23, tag]
by [x / 45, 11, tag]
if true

R_instmult = replace [x, 25, tag], [y, 61, tag1]
by [x * y, 12, tag]
if (tag == tag1)

R_instmod = replace [x, w, tag], [y, z, tag1]
by [x % y, 13, tag]
if (tag == tag1) and (w == 49 or w == 5) and (z == 22 or z == 3)

R_instand = replace [x, 60, tag], [y, 2, tag1]
by [x and y, 14, tag]
if (tag == tag1)

R_instmodi = replace [x, w, tag]
by [x % 15, 15, tag]
if (w == 40 or w == 10)

R_instor = replace [x, 29, tag], [y, 41, tag1]
by [x or y, 16, tag]
if (tag == tag1)

R_instadd2 = replace [x, w, tag], [y, z, tag1]
by [x + y, 17, tag]
if (tag == tag1) and (w == 9 or w == 21) and (z == 20 or z == 26)

R_instdiv2 = replace [x, 0, tag], [y, z, tag1]
by [x / y, 18, tag]
if (tag == tag1) and (z == 4 or z == 1)

R_instdivi2 = replace [x, 6, tag]
by [x / 45, 19, tag]
if true

/* Qtde Labels =  20  */

/* Square_nodes_values            = {[10, instconst1], [30,
   instconst3], [20, instconst2], [50, instconst5], [40,
   instconst4], [60, instconst6]}  */

/* Square_nodes (Ocorrencias)     = {[8, instconst1], [4,
   instconst3], [8, instconst2], [3, instconst5], [4, instconst4
   ], [4, instconst6]}  */

/* Output_registers (Ocorrencias) = {[0, instdiv2], [2, instaddi],
```

```
    [1, instsubi], [0, instmodi], [3, instadd], [0, instor], [0,
   instmod], [0, instmult], [0, instdiv], [0, instdivi2], [0,
   instsub], [0, instand], [0, instadd2], [0, instdivi]}  */
68
69 /* Labels                          = {[18, instdiv2], [6, instaddi
   ], [10, instsubi], [15, instmodi], [5, instadd], [0,
   instconst1], [16, instor], [2, instconst3], [1, instconst2],
   [4, instconst5], [3, instconst4], [8, instdiv], [9, instconst6
   ], [7, instsub], [19, instdivi2], [13, instmod], [17, instadd2
   ], [14, instand], [11, instdivi], [12, instmult]}  */
70
71 /* Ocorrencias                     = {[2, instaddi], [1, instsubi],
    [3, instadd], [7, instconst1], [3, instconst3], [7,
   instconst2], [2, instconst5], [3, instconst4], [3, instconst6
   ]}  */
```

Código 5.2: Experimento 1 - Código Gamma convertido a partir do Código 5.1.

Inicialmente, analisando o Código 5.2 podemos verificar a correta geração da listagem de reações:

```
R_instadd | R_instaddi | R_instsub | R_instdiv | R_instsubi |
R_instdivi | R_instmult | R_instmod | R_instand | R_instmodi |
R_instor | R_instadd2 | R_instdiv2 | R_instdivi2
```

Note que temos 14 reações Gamma, exatamente a quantidade de instruções do código *TALM* que deu origem a esta conversão. Uma vez que no Código 5.1 não possuímos instruções que dão origem a duas reações, conforme explicado no Capítulo anterior, esta relação de 1 para 1 entre a quantidade de instruções em linguagem de montagem e a quantidade de reações está correta.

Além disso, verificamos que o nome das reações também foi montado de maneira satisfatória, pela concatenação da *string "R_"* ao registrador de saída da instrução, de forma que a instrução contida na linha 9 do código *TALM*, definida pelo código abaixo, possa dar origem à entrada *"R_instdiv"* na lista de reações:

```
div  instdiv , instconst4 , instconst5
```

Repare ainda que a listagem das reações encontra-se separada pelo operador de execução paralela ("|").

Após a definição da listagem de reações, o multiconjunto inicial é apresentado. Veja que existem vários elementos replicados, o que está diretamente relacionado à quantidade de utilização destas constantes por instruções expressas no código. Dessa maneira, conforme citado anteriormente, a constante cujo registrador de saída é igual a *"instconst1"*, definida na linha 1 do código de montagem inicial, dará origem a 7 tuplas no multiconjunto inicial, a saber:

```
[10, 120,0], [10, 100, 0], [10, 80, 0], [10, 60, 0], [10, 40, 0],
[10,20, 0], [10, 0, 0]
```

É importante notar que os índices (segundo campo da tupla) de cada elemento que representa instâncias da constante em questão são diferentes e respeitam o posicionamento dentro dos *buffers* (Seção 4.2.3.3), que para este exemplo, possuem tamanho 20 (De 0 a 19 - vide comentário gerado pelo *GFlow* na linha 61 do Código 5.2). Ainda sobre a replicação dos elementos referentes à constante *"instconst1"*, verifique que cada reação que consome tal constante está referindo-se a uma instância específica deste elemento. Por exemplo, a reação *"R_ instdiv2"* consome como primeiro elemento de sua cláusula *REPLACE* o elemento referente à constante *"instconst1"* cujo índice é igual a zero. Já as reações *"R_ instand"* e *"R_ instaddi"* consomem cópias da mesma constante com identificadores 60 e 100, respectivamente. Visando conferir tal informação no código *TALM* (Código 5.1) que deu origem à essa conversão, observe que na linha 19 a instrução *"instdiv2"* possui *"instconst1"* como operando, assim como nas linhas 7 (instrução *"instaddi"*) e 15 (instrução *"instand"*) tal constante também aparece como operando, entre outras.

No Código 5.1 as instruções cujos operandos possam assumir múltiplos candidatos (linhas 8, 11, 14, 16, 18 e 19) dão origem à reações que, utilizam-se de variáveis auxiliares para identificar a instância na cláusula *REPLACE* e usam uma combinação dos valores que esta variável auxiliar pode assumir na cláusula *IF*. Tomemos como exemplo a instrução da linha 19 do Código 5.1. Seu segundo operando possui dois candidatos possíveis:

```
div  instdiv2 , instconst1 , [instconst5 , instconst2]
```

A reação Gamma equivalente gerada foi:

```
R_instdiv2 = replace [x, 0, tag], [y, z, tag1]
    by [x / y, 18, tag]
    if (tag == tag1) and (z == 4 or z == 1)
```

Para o segundo elemento selecionado pela cláusula *REPLACE* acima foi utilizada uma variável auxiliar *"z"*, que, conforme expresso na cláusula *IF*, pode assumir os valores 4 ou 1. Tais valores, referem-se à *"instconst5"* e *"instconst2"*, conforme a listagem de *labels* impressas ao final do código na linha 69 do Código Gamma equivalente.

Como mencionamos no início da discussão sobre este experimento, a instrução *add*, que encontra-se na linha 6 do Código 5.1, é utilizada por outras 3 instruções. Este fato determina que a execução da reação equivalente a esta instrução deva criar três elementos distintos. Assim, vejamos a reação Gamma equivalente (conforme linhas 5, 6 e 7 do Código Gamma gerado):

```
R_instadd = replace [x, 120, tag], [y, 121, tag1]
    by [x + y, 5, tag], [x + y, 45, tag], [x + y, 25, tag]
    if (tag == tag1)
```

De fato a cláusula *BY* está montada de forma a gerar 3 elementos, cujos identificadores são 5, 25 e 45. Conferindo na listagem de *labels* geradas para fins de *Debug*, na linha 69 verificamos a entrada *[5, instadd]*, conforme esperado.

Vale reforçar mais uma vez que alguns identificadores utilizados na replicação de elementos no multiconjunto inicial, na replicação de elementos gerados pela cláusula *BY* ou pelos elementos selecionados nas cláusulas *REPLACE/IF*, apresentam valores maiores que a quantidade de *labels* (que neste exemplo é igual a 20). Isso deve-se ao fato de tais identificadores estarem reposicionados em *buffers*, conforme discutido no Capítulo anterior.

Por fim, pela análise da listagem de reações gerada, pela composição do multiconjunto inicial e pela geração de cada reação, passando por questões de operandos com múltiplos candidatos, replicação de elementos a serem gerados e outras questões aqui apresentadas, vimos que, para este primeiro experimento, a conversão realizada pelo *GFlow* está correta. Agora veremos detalhes de outras categorias de instruções que merecem atenção especial em um segundo experimento.

#### 5.1.2.2 Experimento 2 - Instruções Lógicas

O segundo experimento proposto apresenta uma aplicação em linguagem de montagem do *TALM* onde somente utilizamos instruções pertencente à categoria de instruções lógicas e suas variações com operando imediato. Para tanto, considere o seguinte código *TALM* abaixo (Código 5.3):

```
1  const instconst1, 10
2  const instconst2, 20
3  const instconst3, 30
4  const instconst4, 40
5  const instconst5, 50
6  const instconst6, 60
7  const instconst7, 70
8  lthan instlthan, [instconst1, instconst2], instconst3
9  gthan instgthan, instconst4, instlthan
10 leq instleq, instconst1, [instlthan, instgthan]
11 geq instgeq, instgthan, instconst5
12 lthani instlthani, instlthan, 45
13 gthani instgthani, [instconst4, instconst5], 465
14 leqi instleqi, instconst6, 845
15 geqi instgeqi, instlthani, 332
```

Código 5.3: Experimento 2 - Linguagem de Montagem do *TALM*.

Para o experimento elaborado no Código 5.3, utilizamos somente instruções que geram duas reações cada, por ocasião do procedimento de conversão, pela inexistência da cláusula *ELSE* na sintaxe das implementações de Gamma usadas (Seção 4.2.3.4). Desta maneira, as 8 instruções que constam na linguagem de montagem deste exemplo, darão origem a 16 reações em Gamma. Da mesma maneira, utilizou-se operandos com múltiplos candidatos e registradores de saída de instruções são utilizados por mais de uma instrução diferente. Vale notar que para este exemplo, uma constante foi definida porém não utilizada por nenhuma instrução. Trata-se da constante definida na linha 7, cujo registrador de saída é igual a *"instconst7"*.

Agora vejamos a aplicação Gamma equivalente ao Código 5.3 gerado através da utilização do *GFlow*, apresentado no Código 5.4 abaixo:

```
1  R_instlthan_true | R_instlthan_false | R_instgthan_true |
       R_instgthan_false | R_instleq_true | R_instleq_false |
       R_instgeq_true | R_instgeq_false | R_instlthani_true |
       R_instlthani_false | R_instgthani_true | R_instgthani_false |
       R_instleqi_true | R_instleqi_false | R_instgeqi_true |
       R_instgeqi_false {[10, 15, 0], [10, 0, 0], [30, 2, 0], [20, 1,
        0], [50, 19, 0], [50, 4, 0], [40, 18, 0], [40, 3, 0], [60, 5,
        0]}
2
3  where
4
5  R_instlthan_true = replace [x, w, tag], [y, 2, tag1]
6  by [1, 7, tag], [1, 37, tag], [1, 22, tag]
7  if (tag == tag1) and (x  <  y) and (w == 15 or w == 1)
8
9  R_instlthan_false = replace [x, w, tag], [y, 2, tag1]
10 by [0, 7, tag], [0, 37, tag], [0, 22, tag]
11 if (tag == tag1) and (x  >=  y) and (w == 15 or w == 1)
12
13 R_instgthan_true = replace [x, 18, tag], [y, 37, tag1]
14 by [1, 8, tag], [1, 23, tag]
15 if (tag == tag1) and (x  >  y)
16
17 R_instgthan_false = replace [x, 18, tag], [y, 37, tag1]
18 by [0, 8, tag], [0, 23, tag]
19 if (tag == tag1) and (x  <=  y)
20
21 R_instleq_true = replace [x, 0, tag], [y, z, tag1]
22 by [1, 9, tag]
23 if (tag == tag1) and (x  <=  y) and (z == 22 or z == 23)
24
25 R_instleq_false = replace [x, 0, tag], [y, z, tag1]
26 by [0, 9, tag]
27 if (tag == tag1) and (x  >  y) and (z == 22 or z == 23)
```

```
28
29 R_instgeq_true = replace [x, 8, tag], [y, 19, tag1]
30 by [1, 10, tag]
31 if (tag == tag1) and (x  >=  y)
32
33 R_instgeq_false = replace [x, 8, tag], [y, 19, tag1]
34 by [0, 10, tag]
35 if (tag == tag1) and (x  <  y)
36
37 R_instlthani_true = replace [x, 7, tag]
38 by [1, 11, tag]
39 if (x  < 45)
40
41 R_instlthani_false = replace [x, 7, tag]
42 by [0, 11, tag]
43 if (x  >= 45)
44
45 R_instgthani_true = replace [x, w, tag]
46 by [1, 12, tag]
47 if (x  > 465) and (w == 3 or w == 4)
48
49 R_instgthani_false = replace [x, w, tag]
50 by [0, 12, tag]
51 if (x  <= 465) and (w == 3 or w == 4)
52
53 R_instleqi_true = replace [x, 5, tag]
54 by [1, 13, tag]
55 if (x  <= 845)
56
57 R_instleqi_false = replace [x, 5, tag]
58 by [0, 13, tag]
59 if (x  > 845)
60
61 R_instgeqi_true = replace [x, 11, tag]
62 by [1, 14, tag]
63 if (x  >= 332)
64
65 R_instgeqi_false = replace [x, 11, tag]
66 by [0, 14, tag]
67 if (x  < 332)
68
69 /* Qtde Labels =  15  */
70
71 /* Square_nodes_values            = {[10, instconst1], [30,
      instconst3], [20, instconst2], [50, instconst5], [40,
      instconst4], [70, instconst7], [60, instconst6]}  */
72
```

```
73 /* Square_nodes (Ocorrencias)     = {[3, instconst1], [2,
      instconst3], [2, instconst2], [3, instconst5], [3, instconst4
      ], [1, instconst7], [2, instconst6]}  */
74
75 /* Output_registers (Ocorrencias) = {[0, instleq], [0, instleqi],
      [3, instlthan], [1, instlthani], [0, instgeqi], [0, instgeq],
      [2, instgthan], [0, instgthani]}  */
76
77 /* Labels                         = {[9, instleq], [13, instleqi],
       [0, instconst1], [7, instlthan], [2, instconst3], [1,
      instconst2], [4, instconst5], [3, instconst4], [6, instconst7
      ], [5, instconst6], [14, instgeqi], [11, instlthani], [10,
      instgeq], [8, instgthan], [12, instgthani]}  */
78
79 /* Ocorrencias                    = {[2, instconst1], [2,
      instconst5], [1, instconst3], [1, instconst2], [3, instlthan],
       [2, instconst4], [1, instconst6], [1, instlthani], [2,
      instgthan]}  */
```

Código 5.4: Experimento 2 - Código Gamma convertido a partir do Código 5.3.

Tendo em vista a quantidade de instruções (todas categorizadas como instruções lógicas do *TALM*) no Código 5.3 que originou esta conversão, verificamos que o Código 5.4 gerou corretamente 16 reações, duas para cada uma das 8 instruções em linguagem de montagem:

```
R_instlthan_true | R_instlthan_false | R_instgthan_true |
R_instgthan_false | R_instleq_true | R_instleq_false |
R_instgeq_true | R_instgeq_false | R_instlthani_true |
R_instlthani_false | R_instgthani_true | R_instgthani_false |
R_instleqi_true | R_instleqi_false | R_instgeqi_true |
R_instgeqi_false
```

Dessa maneira, cada instrução deu origem a duas reações distintas: uma cuja *string* que identifica o nome da reação termina em *"_ true"* e outra finalizada por *"_ - false"*. Isso deve-se ao fato da necessidade de implementar operações "antagônicas" ou "complementares", pela inexistência da cláusula *ELSE* nas reações que utilizam sintaxe Gamma proposta por Juarez Muylaert.

Tomemos como exemplo a instrução abaixo, declarada na linha 11 do Código 5.3:

```
geq  instgeq , instgthan , instconst5
```

Esta instrução deu origem as seguintes reações definidas entre as linhas 29 - 35 do Código 5.4 convertido:

```
R_instgeq_true = replace [x, 8, tag], [y, 19, tag1]
    by [1, 10, tag]
    if (tag == tag1) and (x   >=   y)

R_instgeq_false = replace [x, 8, tag], [y, 19, tag1]
    by [0, 10, tag]
    if (tag == tag1) and (x   <   y)
```

Note que a primeira reação (*R_ instgeq_ true*) implementa a operação descrita pela instrução originária *">="* (conforme consta na cláusula *IF* de *R_ instgeq_ true*). Veja ainda que para esta reação, o elemento a ser criado por ocasião de sua execução possui valor 1 ($[1, 10, tag]$) o que faz bastante sentido pois, caso a reação ocorra (caso x >= y), o valor Booleano *"true"* deve ser criado como resultado da instrução em *TALM*.

Entretanto, caso o primeiro operando da instrução *"instgeq"* não seja "maior ou igual" ao segundo operando, a instrução deve produzir uma saída cujo valor é igual a zero. Daí surge a necessidade de produção de mais uma reação, onde a comparação realizada seria "antagônica" ou "complementar" à instrução *"instgeq"*. Assim a instrução *"R_ instgeq_ false"* cumpre esse papel, realizando a comparação "<" e criando um elemento cujo valor é igual a zero ($[0, 10, tag]$). Note que aqui o identificador do elemento gerado por ambas as reações é idêntico e possui valor igual a 10 (cláusula *BY*), o que também faz sentido, uma vez que as reações são mutuamente excludentes. Assim, sempre que existirem os elementos cujos identificadores forem iguais a 8 e 19, somente uma dessas duas reações poderá executar e, nesse caso, um elemento de identificador 10 será gerado.

Da mesma maneira conforme apresentado no primeiro experimento, o multiconjunto inicial foi gerado corretamente, respeitando a quantidade de vezes que cada constante foi utilizada como operando de alguma instrução:

```
{[10, 15, 0], [10, 0, 0], [30, 2, 0], [20, 1, 0], [50, 19, 0],
 [50, 4, 0], [40, 18, 0], [40, 3, 0], [60, 5, 0]}
```

A Tabela 5.1 abaixo, apresenta a quantidade de vezes em que cada constante foi utilizada como operando de alguma instrução, além de apresentar quais instruções (e em qual linha do referido código) as utilizaram, referente a aplicação em linguagem de montagem apresentada no Código 5.3:

Tabela 5.1: Experimento 2 - Utilização de constantes por instruções.

| Constante | Ocorrências como Operando | Instrução / Linha |
|---|---|---|
| instconst1 | 2 | instlthan/8 ; instleq/10 |
| instconst2 | 1 | instlthan/8 |
| instconst3 | 1 | instlthan/8 |
| instconst4 | 2 | instgthan/9 ; instgthani/13 |
| instconst5 | 2 | instgeq/11 ; instgthani/13 |
| instconst6 | 1 | instleqi/14 |
| instconst7 | 0 | N/A |

Conforme os dados apresentados na Tabela 5.1, a instrução *"instconst1"* fora utilizada por duas outras instruções, resultando nos elementos $[10, 15, 0]$ *e* $[10, 0, 0]$ no multiconjunto inicial, de maneira correta. Outras instruções também foram utilizadas por mais de uma instrução, como o caso das instruções *"instconst4"* e *"instconst5"* dando origem aos elementos $[40, 18, 0]$, $[40, 3, 0]$ e $[50, 19, 0]$, $[50, 4, 0]$, respectivamente. Entretanto repare que a instrução *"instconst7"* (linha 7 do Código 5.3) não foi utilizada como operando de nenhuma instrução. Por esse motivo, não existe elemento no multiconjunto inicial referente a esta constante.

Podemos verificar que as instruções que possuem operandos com múltiplos candidatos (linhas 8, 10 e 13) deram origem a reações que trataram adequadamente tais características, utilizando-se de variáveis auxiliares em suas cláusulas *REPLACE* associadas ao correto tratamento nas cláusulas *IF*, conforme vimos no primeiro experimento.

A exemplo do primeiro experimento, instruções cujo registrador de saída seja utilizado por mais de uma instrução deverão dar origem a reações que produzam tantos elementos quantas forem as utilizações de seus registradores de saída. Assim, as reações *"R_instlthan_true"* e *"R_instlthan_false"* produzem 3 elementos (referentes a sua utilização pelas instruções das linhas 9, 10 e 12 do Código *TALM*). Da mesma maneira as instruções *"R_instgthan_true"* e *"R_instgthan_false"* produzem 2 elementos cada (pela sua utilização como operando das instruções definidas nas linhas 10 e 11 do Código 5.3).

Por fim, pela análise das particularidades das conversões das instruções lógicas do *TALM* em código Gamma, que dentre outros aspectos necessita duplicar cada reação equivalente, além da verificação de detalhes mais gerais do processo em pauta, vimos que a conversão fornecida pelo *GFlow* está correta. O próximo experimento visa analisar a conversão dos dois últimos tipos de instrução: *Steer* e *Inctag*.

#### 5.1.2.3 Experimento 3 - Instruções *Steer* e *Inctag*

O terceiro experimento proposto para o *GFlow* visa analisar detalhes inerentes às instruções *Steer* e *Inctag*, utilizadas para mecanismos de desvios e controle de laços, respectivamente. Dessa maneira, propomos o seguinte conjunto de instruções (Código 5.5):

```
1  const instconst1, 10
2  const instconst2, 20
3  const instconst3, 30
4  const instconst4, 40
5  steer inststeer1, instinctag1, instconst1
6  steer inststeer2, inststeer1.t, [instconst2, instconst3]
7  steer inststeer3, [instconst2, instinctag1], inststeer1.t
8  steer inststeer4, [instconst2, inststeer1.t], [instconst3,
       instconst4]
9  inctag instinctag1, inststeer2.t
10 inctag instinctag2, [inststeer2.t, inststeer1.f]
```

Código 5.5: Experimento 3 - Linguagem de Montagem do *TALM*.

O Código acima é composto por 4 reações do tipo *Steer*, 2 instruções *Inctag* além de 4 constantes. Note que foram selecionadas algumas instruções cujos operadores possam assumir múltiplos candidatos (linhas 6, 7, 8 e 10). É importante verificar que caso seja necessário utilizar o registrador de saída de algum *steer*, o mesmo deve ser feito indicando a saída *"true"* ou *"false"* específica, identificada através da concatenação de *".t"* ou *".f"* respectivamente ao final da *string* do registrador de saída. Assim, a instrução *"Inctag"* definida na linha 9 utiliza como operando a saída *"true"* do *steer "inststeer2"*, através da indicação *"inststeer2.t"*.

```
1  R_inststeer1_true | R_inststeer1_false | R_inststeer2_true |
       R_inststeer2_false | R_inststeer3_true | R_inststeer3_false |
       R_inststeer4_true | R_inststeer4_false | R_instinctag1 |
       R_instinctag2 {[10, 0, 0], [30, 16, 0], [30, 2, 0], [20, 29,
       0], [20, 15, 0], [20, 1, 0], [40, 3, 0]}
2
3  where
4
5  R_inststeer1_true = replace [x, 26, tag], [y, 0, tag1]
6  by [y, 4, tag], [y, 32, tag], [y, 18, tag]
7  if (tag == tag1) and (x == 1)
8
9  R_inststeer1_false = replace [x, 26, tag], [y, 0, tag1]
10 by [y, 5, tag]
11 if (tag == tag1) and (x != 1)
12
13 R_inststeer2_true = replace [x, 32, tag], [y, z, tag1]
```

```
by [y, 6, tag], [y, 20, tag]
if (tag == tag1) and (x == 1) and (z == 29 or z == 16)

R_inststeer2_false = replace [x, 32, tag], [y, z, tag1]
by [y, 7, tag]
if (tag == tag1) and (x != 1) and (z == 29 or z == 16)

R_inststeer3_true = replace [x, w, tag], [y, 18, tag1]
by [y, 8, tag]
if (tag == tag1) and (x == 1) and (w == 15 or w == 12)

R_inststeer3_false = replace [x, w, tag], [y, 18, tag1]
by [y, 9, tag]
if (tag == tag1) and (x != 1) and (w == 15 or w == 12)

R_inststeer4_true = replace [x, w, tag], [y, z, tag1]
by [y, 10, tag]
if (tag == tag1) and (x == 1) and (w == 1 or w == 4) and (z == 2
   or z == 3)

R_inststeer4_false = replace [x, w, tag], [y, z, tag1]
by [y, 11, tag]
if (tag == tag1) and (x != 1) and (w == 1 or w == 4) and (z == 2
   or z == 3)

R_instinctag1 = replace [x, 20, tag]
by [x, 12, tag+1], [x, 26, tag+1]
if true

R_instinctag2 = replace [x, w, tag]
by [x, 13, tag+1]
if (w == 6 or w == 5)

/* Qtde Labels =  14  */

/* Square_nodes_values            = {[10, instconst1], [30,
   instconst3], [20, instconst2], [40, instconst4]}  */

/* Square_nodes (Ocorrencias)     = {[2, instconst1], [3,
   instconst3], [4, instconst2], [2, instconst4]}  */

/* Output_registers (Ocorrencias) = {[0, inststeer3.t], [3,
   inststeer1.t], [0, instinctag2], [2, instinctag1], [0,
   inststeer4.t], [0, inststeer3.f], [0, inststeer2.f], [1,
   inststeer1.f], [0, inststeer4.f], [2, inststeer2.t]}  */

/* Labels                         = {[8, inststeer3.t], [4,
```

```
   inststeer1.t], [10, inststeer4.t], [13, instinctag2], [12,
   instinctag1], [0, instconst1], [2, instconst3], [1, instconst2
   ], [3, instconst4], [7, inststeer2.f], [5, inststeer1.f], [9,
   inststeer3.f], [11, inststeer4.f], [6, inststeer2.t]}  */
54
55 /* Ocorrencias                    = {[3, inststeer1.t], [2,
   instinctag1], [1, instconst1], [2, instconst3], [3, instconst2
   ], [1, instconst4], [1, inststeer1.f], [2, inststeer2.t]}  */
```

Código 5.6: Experimento 3 - Código Gamma convertido a partir do Código 5.5.

Da mesma forma como ocorre com as instruções lógicas, instruções *Steer* também dão origem a duas reações, conforme discutimos no Capítulo anterior. Dessa maneira, verificamos que a aplicação convertida em Gamma apresentada no Código 5.6, gera corretamente duas reações para cada instrução do tipo *steer* existente. Por outro lado, instruções *Inctag* dão origem a somente uma reação. Dessa forma, as 4 instruções *Steer* definidas no Código 5.5 deram origem a 8 reações Gamma (Código 5.6) e as 2 instruções *Inctag* geraram 2 reações Gamma, totalizando 10 reações no Código Gamma convertido, corretamente.

O multiconjunto inicial também apresentou conversão correta, replicando corretamente a quantidade de constantes levando em consideração a quantidade de utilização das mesmas como operando de alguma instrução. Portanto, a constante *"instconst1"* fora utilizada uma vez, *"instconst2"* 3 vezes, *"instconst3"* 2 vezes e *"instconst4"* teve seu registrador de saída utilizado somente por uma instrução.

Como cada instrução do tipo *Steer* ao ser utilizada como operando de alguma instrução, deve ter sua saída *"true"* ou *"false"* explicitamente especificada, pode ocorrer que a quantidade de elementos criados pela equivalente reação *"true"* seja diferente de sua reação *"false"*. Analisemos a instrução do tipo *"Steer"* definida na linha 5 do Código 5.5:

```
steer  inststeer1 , instinctag1 , instconst1
```

Note que seu registrador de saída *"true"* está sendo utilizado como operando de 3 instruções (linhas 6, 7 e 8) ao passo que sua saída *"false"* é utilizada por somente uma instrução, definida na linha 10 do referido código fonte. Assim, a reação Gamma que reproduz o valor do segundo elemento selecionado (segundo operando), caso o valor do primeiro elemento (operando seletor) seja igual a *"true"* (reação cujo nome é *R_inststeer1_true*) cria três elementos no multiconjunto (caso venha a reagir), conforme apresentado na linha 6 do Código 5.6. Por outro lado, a reação *R_inststeer1_false* gera somente um elemento (vide linha 10 do código Gamma convertido). Corroborando com o teste condicional necessário às duas reações geradas por um *Steer*, note que em sua cláusula *IF* tais reações *"true"* apresentam comparação x == 1, ao passo que reações *"false"*, comparação oposta (x != 1).

Ao contrário do que foi apresentado no experimento anterior, as tuplas geradas pelas reações *"true"* e *"false"* referentes a um *Steer* possuem identificadores diferentes, o que faz sentido, uma vez que estes dados são utilizados por reações que desejam diferencia-los (vide necessidade de expressar *".t"* e *".f"* ao utilizar o registrador de saída de algum *Steer* no *TALM*).

Já instruções do tipo *Inctag* apresentam somente 1 operando (vide linhas 9 e 10 do Código 5.5), podendo este possuir múltiplos candidatos. A função de tal instrução consiste no incremento do rótulo de iteração, que em nosso cenário, corresponde ao último campo da tupla que representa um elemento pertencente ao multiconjunto. Dessa forma, nas reações Gamma equivalentes a uma instrução do tipo *Inctag*, os rótulos de iteração são incrementados, conforme linhas 38 e 42 do Código 5.6 (cláusulas *BY* das reações equivalentes). Repare que, em todas as reações, convertidas a partir de qualquer instrução *TALM* (com exceção àquelas que utilizam operando imediato) existe um teste condicional (tag == tag1) permitindo que reajam somente elementos pertencentes à mesma interação.

Tendo em vista o que foi exposto acima à respeito das conversões de instruções do tipo *Steer* e *Inctag* pudemos verificar que a implementação de tais conversões através do *GFlow* encontra-se correta.

Finalizando este primeiro grupo de experimentos, o experimento 4 propõe uma aplicação que contém instruções de todas as categorias citadas anteriormente.

#### 5.1.2.4 Experimento 4 - Instruções Variadas

Para finalizar a primeira etapa de experimentos afetos ao *GFlow*, que tratam de demonstrações das conversões a partir de um código manualmente escrito em linguagem de montagem dataflow, propomos o código fonte abaixo:

```
1  const instconst1, 10
2  const instconst2, 20
3  const instconst3, 30
4  const instconst4, 40
5  const instconst5, 50
6  add instadd, instconst1, [instconst2, instconst3]
7  subi instsubi, [instconst1, instconst2], 335
8  mult instmult, instadd, instconst2
9  const instconst6, 60
10 mod instmod, [instconst6, instadd], [instconst3, instconst4]
11 divi instdivi, instadd, 45
12 steer inststeer, instinctag, instlthan
13 lthan instlthan, [instconst1, instconst2], instconst3
14 leq instleq, instconst1, [instlthan, inststeer.t]
15 geqi instgeqi, instadd, 332
16 const instconst7, 70
```

```
17 gthani instgthani, [instconst4, instconst5], 465
18 inctag instinctag, [inststeer.t, instconst5]
19 sub instsub, instlthan, inststeer.f
```

Código 5.7: Experimento 4 - Linguagem de Montagem do *TALM*.

Neste caso de teste, utilizamos exemplos de instruções das categorias anteriormente citadas, sejam elas: instruções aritméticas e suas variações com operando imediato, instruções lógicas e suas variações com operando imediato, *Steers* e *Inctags*. Portanto, temos as seguintes instruções aritméticas: *add* (linha 6), *subi* (linha 7), *mult* (linha 8), *mod* (linha 10), *divi* (linha 11) e *sub* (linha 19). O exemplo também apresenta as seguintes instruções lógicas: *lthan* (linha 13), *leq* (linha 14), *geqi* (linha 15) e *gthani* (linha 17). Temos ainda um exemplo de *Steer* (linha 12), um *Inctag* (linha 18), além de constantes definidas nas linhas de 1 a 5, 9 e 16.

Diversas instruções apresentam operandos que podem assumir múltiplos candidatos, além de registradores de saída de instruções serem utilizados como operandos de instruções. Vejamos agora o código Gamma correspondente extraído da conversão realizada pelo *GFlow*:

```
1  R_instadd | R_instsubi | R_instmult | R_instmod | R_instdivi |
       R_inststeer_true | R_inststeer_false | R_instlthan_true |
       R_instlthan_false | R_instleq_true | R_instleq_false |
       R_instgeqi_true | R_instgeqi_false | R_instgthani_true |
       R_instgthani_false | R_instinctag | R_instsub {[10, 60, 0],
       [10, 40, 0], [10, 20, 0], [10, 0, 0], [30, 42, 0], [30, 22,
       0], [30, 2, 0], [20, 61, 0], [20, 41, 0], [20, 21, 0], [20, 1,
        0], [50, 24, 0], [50, 4, 0], [40, 23, 0], [40, 3, 0], [60, 8,
        0]}
2
3  where
4
5  R_instadd = replace [x, 60, tag], [y, z, tag1]
6  by [x + y, 5, tag], [x + y, 65, tag], [x + y, 45, tag], [x + y,
       25, tag]
7  if (tag == tag1) and (z == 61 or z == 42)
8
9  R_instsubi = replace [x, w, tag]
10 by [x - 335, 6, tag]
11 if (w == 40 or w == 41)
12
13 R_instmult = replace [x, 65, tag], [y, 21, tag1]
14 by [x * y, 7, tag]
15 if (tag == tag1)
16
17 R_instmod = replace [x, w, tag], [y, z, tag1]
18 by [x % y, 9, tag]
```

```
if (tag == tag1) and (w == 8 or w == 45) and (z == 22 or z == 23)

R_instdivi = replace [x, 25, tag]
by [x / 45, 10, tag]
if true

R_inststeer_true = replace [x, 18, tag], [y, 53, tag1]
by [y, 11, tag], [y, 31, tag]
if (tag == tag1) and (x == 1)

R_inststeer_false = replace [x, 18, tag], [y, 53, tag1]
by [y, 12, tag]
if (tag == tag1) and (x != 1)

R_instlthan_true = replace [x, w, tag], [y, 2, tag1]
by [1, 13, tag], [1, 53, tag], [1, 33, tag]
if (tag == tag1) and (x  <  y) and (w == 20 or w == 1)

R_instlthan_false = replace [x, w, tag], [y, 2, tag1]
by [0, 13, tag], [0, 53, tag], [0, 33, tag]
if (tag == tag1) and (x  >=  y) and (w == 20 or w == 1)

R_instleq_true = replace [x, 0, tag], [y, z, tag1]
by [1, 14, tag]
if (tag == tag1) and (x  <=  y) and (z == 33 or z == 31)

R_instleq_false = replace [x, 0, tag], [y, z, tag1]
by [0, 14, tag]
if (tag == tag1) and (x  >  y) and (z == 33 or z == 31)

R_instgeqi_true = replace [x, 5, tag]
by [1, 15, tag]
if (x  >= 332)

R_instgeqi_false = replace [x, 5, tag]
by [0, 15, tag]
if (x  < 332)

R_instgthani_true = replace [x, w, tag]
by [1, 17, tag]
if (x  > 465) and (w == 3 or w == 24)

R_instgthani_false = replace [x, w, tag]
by [0, 17, tag]
if (x  <= 465) and (w == 3 or w == 24)

R_instinctag = replace [x, w, tag]
```

```
66 by [x, 18, tag+1]
67 if (w == 11 or w == 4)
68
69 R_instsub = replace [x, 13, tag], [y, 12, tag1]
70 by [x - y, 19, tag]
71 if (tag == tag1)
72
73 /* Qtde Labels =  20  */
74
75 /* Square_nodes_values            = {[10, instconst1], [30,
      instconst3], [20, instconst2], [50, instconst5], [40,
      instconst4], [70, instconst7], [60, instconst6]}  */
76
77 /* Square_nodes (Ocorrencias)     = {[5, instconst1], [4,
      instconst3], [5, instconst2], [3, instconst5], [3, instconst4
      ], [1, instconst7], [2, instconst6]}  */
78
79 /* Output_registers (Ocorrencias) = {[1, inststeer.f], [0,
      instmult], [0, instsubi], [0, instleq], [4, instadd], [0,
      instmod], [3, instlthan], [0, instgeqi], [0, instsub], [2,
      inststeer.t], [0, instgthani], [0, instdivi], [1, instinctag]}
        */
80
81 /* Labels                         = {[12, inststeer.f], [15,
      instgeqi], [7, instmult], [6, instsubi], [14, instleq], [5,
      instadd], [0, instconst1], [13, instlthan], [2, instconst3],
      [1, instconst2], [4, instconst5], [3, instconst4], [16,
      instconst7], [8, instconst6], [19, instsub], [11, inststeer.t
      ], [9, instmod], [17, instgthani], [10, instdivi], [18,
      instinctag]}  */
82
83 /* Ocorrencias                    = {[1, inststeer.f], [4, instadd
      ], [4, instconst1], [2, instconst5], [3, instconst3], [4,
      instconst2], [3, instlthan], [2, instconst4], [1, instconst6],
       [2, inststeer.t], [1, instinctag]}  */
```

Código 5.8: Experimento 4 - Código Gamma convertido a partir do Código 5.5.

A conversão deu origem a uma quantidade de 17 reações, onde para cada uma das 6 instruções aritméticas, exatamente 1 reação foi gerada, totalizando 6 reações. Cada uma das 4 instruções lógicas deu origem a duas reações, num total de 8 reações. A instrução *Steer* deu origem a duas reações e o *Inctag* gerou uma única reação. Verificamos ainda a corretude na geração dos nomes das instruções:

```
R_instadd | R_instsubi | R_instmult |
R_instmod | R_instdivi | R_inststeer_true |
R_inststeer_false | R_instlthan_true | R_instlthan_false |
```

```
R_instleq_true | R_instleq_false | R_instgeqi_true |
R_instgeqi_false | R_instgthani_true | R_instgthani_false |
R_instinctag | R_instsub
```

Tendo em vista a geração do multiconjunto inicial, o mesmo respeitou a quantidade de elementos a serem replicados, assim como a não geração de elemento referente à constante cujo registrador de saída é igual a *"instconst7"* (definida na linha 16 do Código 5.7), uma vez que a mesma não foi utilizada como operando de nenhuma instrução. Como exemplo, verifique que a constante *"instconst1"* é utilizada como operando de 4 instruções (definidas nas linhas 6, 7, 13 e 14). Note ainda que a declaração da constante *"instconst6"* fora do bloco inicial de constantes não gerou nenhum tipo de prejuízo a corretude da conversão.

A respeito da geração dos códigos das reações, as instruções cujos registradores de saída foram utilizados mais de uma vez como operandos de outras instruções, tiveram seus códigos Gamma correspondentes com replicação de elementos na cláusula *BY* (linhas 6 - quatro replicações, linha 26 - duas replicações, linha 34 - três replicações e linha 38 - três replicações).

Operadores com múltiplos candidatos também tiveram processamento adequado, como foi o caso das reações: *R_instadd*, *R_instsubi*, *R_instmod*, *R_instlthan_true*, *R_instlthan_false*, *R_instleq_true*, *R_instleq_false*, *R_instgthani_true*, *R_instgthani_false* e *R_instinctag*.

Podemos verificar a corretude da conversão em detalhes específicos das conversões, como por exemplo, a reação que converte a instrução *Steer* gerou duas reações (*"true"* e *"false"*) com quantidades de elementos a serem criadas na cláusula *BY* compatíveis com a utilização de seus registradores de saída. As operações realizadas por cada reação estão de acordo e corretas com suas instruções no código em linguagem de montagem que deu início à conversão.

Repare que a reação *R_instgeqi_true* e sua antagônica *R_instgeqi_false* (assim como toda e qualquer reação convertida a partir de uma instrução com operando imediato) não possui a comparação *"tag == tag1"* em sua cláusula *IF*, conforme mencionado anteriormente. Isso deve-se ao fato de que tal tipo de reação seleciona somente um elemento do multiconjunto e a comparação é realizada com um valor numérico, o que faz sentido.

Por fim, podemos verificar que todos os detalhes já verificados nos experimentos anteriores continuam sendo verdade neste exemplo contendo categorias de instruções variadas, o que demonstra que a conversão do *GFlow* para o experimento 4 também foi correta. Agora iniciaremos a segunda parte dos experimentos com o GFlow, onde utilizaremos a *THLL* e o *Couillard*.

### 5.1.3 Conversões a partir da *THLL*

Tendo abordado detalhes das conversões a partir de um código escrito manualmente em linguagem de montagem do *TALM*, esta segunda categoria de experimentos pretende avaliar a corretude de um código C inicial comparado à seu código Gamma equivalente. Para tanto, foram escritos códigos em C, utilizando-se a *THLL* somente para anotações de blocos, ou seja, nenhuma super-instrução fora designada, uma vez que nossa intenção é realizar somente os testes da conversão utilizando dataflow de granularidade fina. Tais códigos C anotados com a *THLL* serão compilados pelo *Couillard*, que irá gerar linguagem de montagem do *TALM* que finalmente será convertido para Gamma através do *GFlow*. Vale ressaltar que aqui não iremos tratar do arquivos de extensão *".dot"* e *".lib.c"* (discutidos no Capítulo anterior), gerados no processo de compilação do *Couillard*. Após a conversão, o código Gamma equivalente será submetido à execução em duas implementações de Gamma distintas, Gamma-Sequencial e Gamma-MPI, apresentadas em maiores detalhes em 4.3.1. Por fim, o resultado da execução do código Gamma nas implementações citadas será comparado ao resultado das aplicações em código C inicialmente definido, compiladas e executadas no *gcc*.

#### 5.1.3.1 Experimento 5 - Operações Aritméticas

O primeiro exemplo sugerido para esta segunda categoria de experimentos do *GFlow*, relaciona algumas operações aritméticas simples em um código C (Código 5.9), listado abaixo:

```
1  // Experimento 5
2  #BEGINBLOCK
3     #include <stdio.h>
4  #ENDBLOCK
5
6  int main()
7  {
8      int a, b, c, d, e, f, x, y, z, w, n, m;
9
10     a = 5;
11     b = 4;
12     c = 7;
13     d = 3;
14     e = 1;
15     f = 2;
16     x = a + b;
17     y = c - d;
18     z = d + e;
```

```
19      w = x * y;
20      n = z / f;
21      m = w + n;
22  }
```

Código 5.9: Experimento 5 - Exemplo em Linguagem C.

Note que o código C acima possui anotações de blocos *THLL* (o código que encontra-se entre as anotações de blocos *THLL* não são compilados pelo *Couillard* - Capítulo 4), entretanto, não possui anotações de super-instruções, conforme discutimos no Capítulo anterior. Desta maneira, visando execução pelo gcc, ao excluirmos as anotações *THLL* (linhas 2 e 4) e imprimirmos o valor da variável *"m"*, o resultado 38 é obtido através da compilação e execução pelo gcc.

Por outro lado, ao compilarmos o código acima (Código 5.9) através do *Couillard*, o código em linguagem de montagem (arquivo cuja extensão é *".fl"*) gerado é descrito abaixo (Código 5.10):

```
1  const flowInstConst139794235451728, 5
2  const flowInstConst139794235451296, 4
3  const flowInstConst139794235505120, 7
4  const flowInstConst139794235504688, 3
5  const flowInstConst139794235505264, 1
6  const flowInstConst139794235504760, 2
7  add flowInstBinop139794235505768, flowInstConst139794235451728,
       flowInstConst139794235451296
8  sub flowInstBinop139794235505696, flowInstConst139794235505120,
       flowInstConst139794235504688
9  add flowInstBinop139794235505840, flowInstConst139794235504688,
       flowInstConst139794235505264
10 mult flowInstBinop139794235505912, flowInstBinop139794235505768,
       flowInstBinop139794235505696
11 div flowInstBinop139794235505984, flowInstBinop139794235505840,
       flowInstConst139794235504760
12 add flowInstBinop139794235506056, flowInstBinop139794235505912,
       flowInstBinop139794235505984
```

Código 5.10: Experimento 5 - Linguagem de Montagem do *TALM* após compilação pelo Couillard (a partir do Código 5.9).

Conforme realizado nos casos de teste pertencentes à primeira categoria de experimentos do *GFlow*, vamos submeter o Código em linguagem *assembly* acima (Código 5.10) à conversão através do *GFlow*, o que dará origem ao código Gamma (Código 5.11) descrito abaixo:

```
(R_flowInstBinop139794235505768 | R_flowInstBinop139794235505696 |
    R_flowInstBinop139794235505840) ; (
   R_flowInstBinop139794235505912 |
   R_flowInstBinop139794235505984) ;
   R_flowInstBinop139794235506056 {[4, 1, 0], [7, 2, 0], [1, 4,
   0], [5, 0, 0], [2, 5, 0], [3, 15, 0], [3, 3, 0]}

/* R_flowInstBinop139794235505768 | R_flowInstBinop139794235505696
    | R_flowInstBinop139794235505840 |
   R_flowInstBinop139794235505912 |
   R_flowInstBinop139794235505984 |
   R_flowInstBinop139794235506056 {[4, 1, 0], [7, 2, 0], [1, 4,
   0], [5, 0, 0], [2, 5, 0], [3, 15, 0], [3, 3, 0]} */

where

R_flowInstBinop139794235505768 = replace [x, 0, tag], [y, 1, tag1]
by [x + y, 6, tag]
if tag == tag1

R_flowInstBinop139794235505696 = replace [x, 2, tag], [y, 15, tag1
   ]
by [x - y, 7, tag]
if tag == tag1

R_flowInstBinop139794235505840 = replace [x, 3, tag], [y, 4, tag1]
by [x + y, 8, tag]
if tag == tag1

R_flowInstBinop139794235505912 = replace [x, 6, tag], [y, 7, tag1]
by [x * y, 9, tag]
if tag == tag1

R_flowInstBinop139794235505984 = replace [x, 8, tag], [y, 5, tag1]
by [x / y, 10, tag]
if tag == tag1

R_flowInstBinop139794235506056 = replace [x, 9, tag], [y, 10, tag1
   ]
by [x + y, 11, tag]
if tag == tag1
```

Código 5.11: Experimento 5 - Código Gamma convertido a partir do Código 5.10.

Conforme abordamos na Fundamentação Teórica desta tese (Capítulo 2) quando apresentamos "Composição de Operadores", as implementações de Gamma utilizadas para executar os códigos acima permitem a utilização de operadores sequenciais

(“;”) e paralelos (“|”). Ainda sobre tais implementações do paradigma Gamma, em algumas ocasiões existem características que, quando são utilizados somente operadores paralelos e alguma reação não possui os elementos que irá necessitar disponíveis no multiconjunto inicial, tal reação pode não voltar mais a ser testada para ser executada. Visando sobrepor esse equívoco das implementações, nos casos de teste abordados nesta segunda categoria de testes, iremos inserir alguns operadores sequenciais, manualmente no código Gamma gerado. Tal atitude visa garantir a correta execução da aplicação. Entretanto, vale a pena mencionar que, nossa conversão apresenta a listagem de reações de maneira correta, todas utilizando operador paralelo de execução. Tendo em vista o exposto, a listagem das reações do Código 5.11 acima, foi alterada pela inserção de operadores sequenciais (“;”) como podemos verificar na linha 1 do referido Código. Note ainda que listagem de reações originalmente gerada pelo *GFlow* encontra-se entre comentários (linha 3).

Visando contribuir para o melhor entendimento, a Figura 5.1 apresenta o grafo correspondente à linguagem de montagem apresentada no Código 5.10:

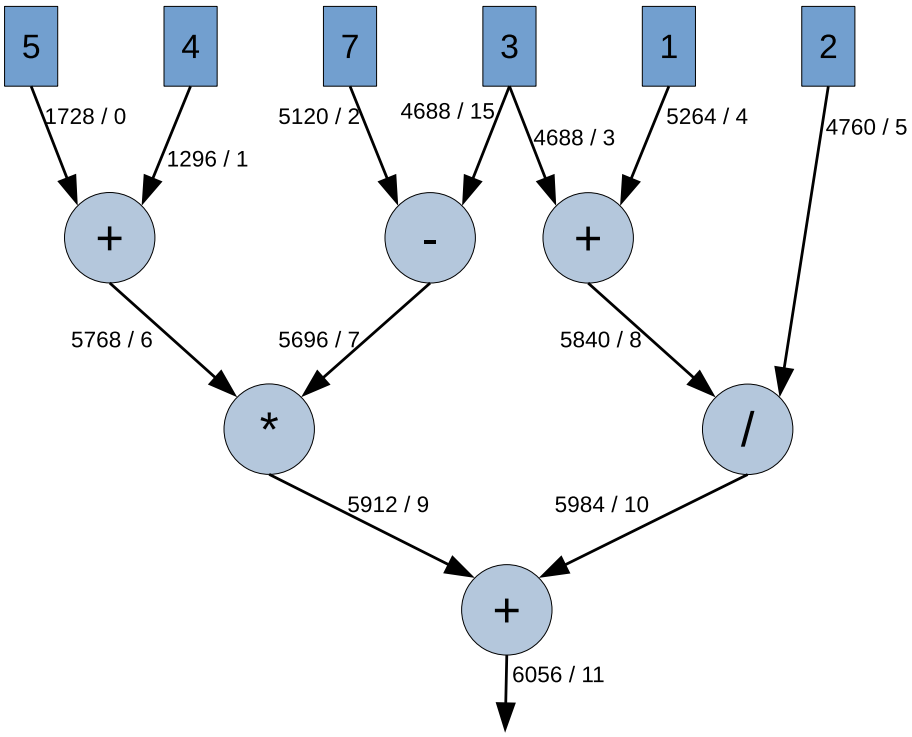


Figura 5.1: GFlow - Experimento 5 - Grafo Correspondente.

Na Figura 5.1, os vértices representam as instruções em TALM (ou reações Gamma) dos códigos equivalentes. Cada aresta possui um rótulo que é formado pelos 4 últimos caracteres do registrador de saída da instrução *TALM* (ou reação Gamma) responsável por sua geração separado por um inteiro correspondente ao identificador da tupla do equivalente código Gamma. Tomemos como exemplo a

constante cujo registrador de saída é *"flowInstConst1397942355046**88**"* (linha 4 do Código 5.10). Tal constante representa o valor inteiro 3 e é utilizada por duas outras instruções. Na figura em questão, as arestas rotuladas por "4688 / 15" e "4688 / 3" representam tal constante. O valor "4688" representa os 4 últimos caracteres do registrador de saída e os valores 3 e 15 são os identificadores destes elementos no multiconjunto inicial (vide definição do multiconjunto inicial ao final da linha 1 do Código 5.11). Tomemos agora como exemplo a instrução de divisão, cuja aresta de saída possui rótulo igual a "5984 / 10". Tal instrução refere-se aquela definida na linha 11 do código *TALM* (*"flowInstBinop139794235505**984**"*), que corresponde à reação definida entre as linhas 23 e 25 do código Gamma correspondente, cuja tupla gerada possui identificador igual a **10**.

A execução do Código 5.11 nas implementações de Gamma (*Gamma-Sequencial* e *Gamma-MPI*) obteve resultados idênticos e iguais a: $[38, 11, 0]$. Tal resultado está correto, uma vez que pertence a tupla gerada pelo último nó do grafo apresentado na Figura 5.1, uma vez que possui identificador igual a 11. Além disso, o valor gerado (38 - primeiro campo da tupla) está idêntico ao valor gerado pela execução do código C inicial (Código 5.9) compilado pelo gcc, conforme afirmamos no início do experimento.

#### 5.1.3.2 Experimento 6 - Desvios Condicionais

O caso de teste apresentado a seguir visa avaliar os resultados de uma conversão onde o código C inicial utiliza-se de estruturas de decisão. Para tanto, propomos o Código abaixo:

```
1  #BEGINBLOCK
2     #include <stdio.h>
3  #ENDBLOCK
4
5  int main()
6  {
7      int a, b, x;
8
9      a = 5;
10     b = 4;
11     x = 0;
12
13     if (a < b)
14         x = a + b;
15     else
16         x = a * b;
17 }
```

Código 5.12: Experimento 6 - Exemplo em Linguagem C.

A execução do código acima, após retirada dos blocos *THLL*, impressão da variável "x" e compilação pelo gcc, gera o valor inteiro igual a 20.

Agora vejamos o código em linguagem da montagem obtido através da compilação pelo *Couillard*:

```
1 const flowInstConst140661069950128, 5
2 const flowInstConst140661069947176, 4
3 const flowInstConst140661069948040, 0
4 lthan flowInstBinop140661069947680, flowInstConst140661069950128,
     flowInstConst140661069947176
5 steer flowInstSteer140661069950920, flowInstBinop140661069947680,
     flowInstConst140661069950128
6 steer flowInstSteer140661070004448, flowInstBinop140661069947680,
     flowInstConst140661069947176
7 add flowInstBinop140661069950776, flowInstSteer140661069950920.t,
     flowInstSteer140661070004448.t
8 steer flowInstSteer140661070004808, flowInstBinop140661069947680,
     flowInstConst140661069948040
9 mult flowInstBinop140661070004952, flowInstSteer140661069950920.f,
      flowInstSteer140661070004448.f
```

Código 5.13: Experimento 6 - Linguagem de Montagem do *TALM* após compilação pelo Couillard (a partir do Código 5.12).

O arquivo *".fl"* acima (Código 5.13) foi submetido à conversão pelo *GFlow* originando a seguinte aplicação Gamma (onde as inserções de operadores sequenciais na listagem de reações já foram realizadas):

```
1 (R_flowInstBinop140661069947680_true |
     R_flowInstBinop140661069947680_false) ; (
     R_flowInstSteer140661069950920_true |
     R_flowInstSteer140661069950920_false |
     R_flowInstSteer140661070004448_true |
     R_flowInstSteer140661070004448_false |
     R_flowInstSteer140661070004808_true |
     R_flowInstSteer140661070004808_false) ; (
     R_flowInstBinop140661069950776 |
     R_flowInstBinop140661070004952) {[0, 2, 0], [4, 13, 0], [4, 1,
      0], [5, 12, 0], [5, 0, 0]}
2
3 /* R_flowInstBinop140661069947680_true |
     R_flowInstBinop140661069947680_false |
     R_flowInstSteer140661069950920_true |
     R_flowInstSteer140661069950920_false |
     R_flowInstSteer140661070004448_true |
     R_flowInstSteer140661070004448_false |
     R_flowInstBinop140661069950776 |
     R_flowInstSteer140661070004808_true |
```

```
   R_flowInstSteer140661070004808_false |
   R_flowInstBinop140661070004952 {[0, 2, 0], [4, 13, 0], [4, 1,
   0], [5, 12, 0], [5, 0, 0]} */

where

R_flowInstBinop140661069947680_true = replace [x, 12, tag], [y,
   13, tag1]
by [1, 3, tag], [1, 27, tag], [1, 15, tag]
if tag == tag1 and (x  <  y)

R_flowInstBinop140661069947680_false = replace [x, 12, tag], [y,
   13, tag1]
by [0, 3, tag], [0, 27, tag], [0, 15, tag]
if tag == tag1 and (x  >=  y)

R_flowInstSteer140661069950920_true = replace [x, 27, tag], [y, 0,
    tag1]
by [y, 4, tag]
if tag == tag1 and (x == 1)

R_flowInstSteer140661069950920_false = replace [x, 27, tag], [y,
   0, tag1]
by [y, 5, tag]
if tag == tag1 and (x != 1)

R_flowInstSteer140661070004448_true = replace [x, 15, tag], [y, 1,
    tag1]
by [y, 6, tag]
if tag == tag1 and (x == 1)

R_flowInstSteer140661070004448_false = replace [x, 15, tag], [y,
   1, tag1]
by [y, 7, tag]
if tag == tag1 and (x != 1)

R_flowInstBinop140661069950776 = replace [x, 4, tag], [y, 6, tag1]
by [x + y, 8, tag]
if tag == tag1

R_flowInstSteer140661070004808_true = replace [x, 3, tag], [y, 2,
   tag1]
by [y, 9, tag]
if tag == tag1 and (x == 1)

R_flowInstSteer140661070004808_false = replace [x, 3, tag], [y, 2,
    tag1]
```

```
40 by [y, 10, tag]
41 if tag == tag1 and (x != 1)
42
43 R_flowInstBinop140661070004952 = replace [x, 5, tag], [y, 7, tag1]
44 by [x * y, 11, tag]
45 if tag == tag1
```

Código 5.14: Experimento 6 - Código Gamma convertido a partir do Código 5.13.

A Figura 5.2 apresenta o grafo dataflow relacionado a este experimento:

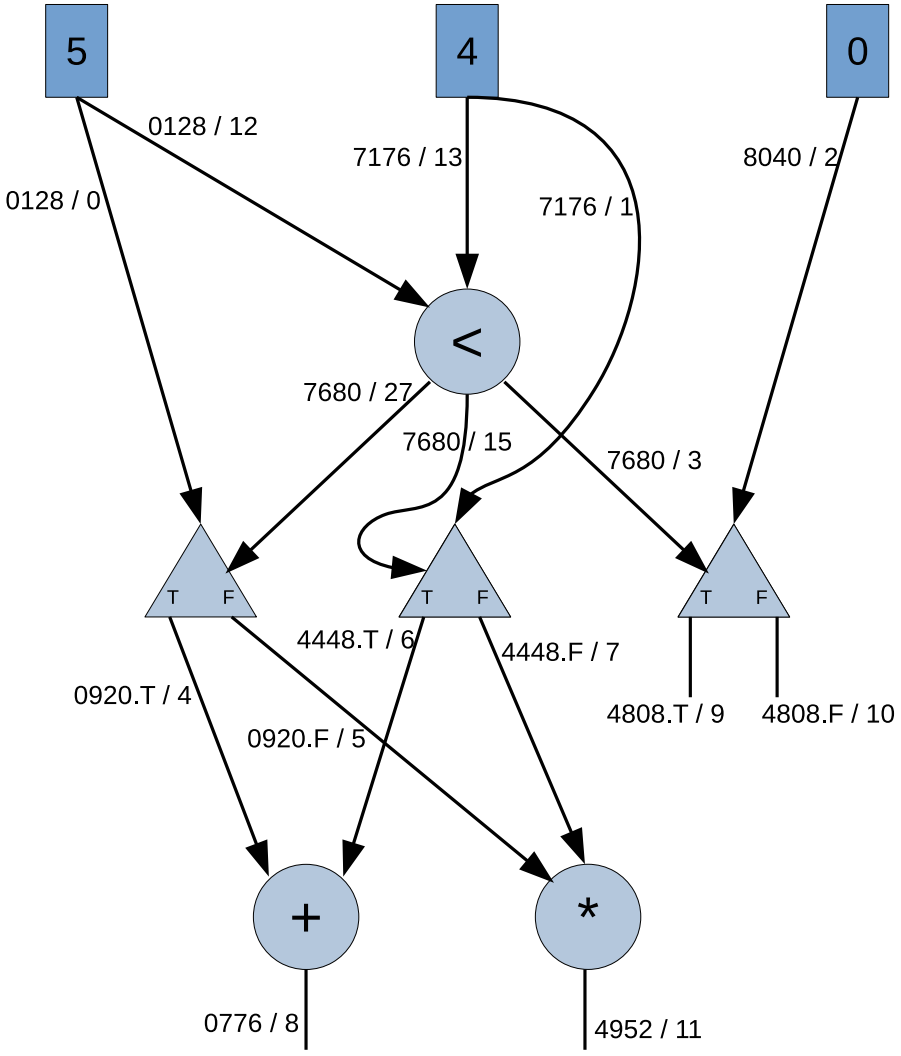


Figura 5.2: GFlow - Experimento 6 - Grafo Correspondente.

Assim, ao submetermos a aplicação Gamma (Código 5.14) aos dois ambientes de execução citados (*Gamma-Sequencial* e *Gamma-MPI*) os resultados obtidos foram iguais e consistiram nas seguintes tuplas: $[20, 11, 0]$ e $[1, 10, 0]$. Repare que, conforme o teste condicional da estrutura *IF* (linha 13 do Código 5.12), a cláusula *ELSE* fora executada (uma vez que o teste $a < b$ apresentou resultado falso). Dessa maneira,

somente a instrução de multiplicação foi executada, que na Figura 5.2 representa o nó cuja aresta de saída possui rótulo igual a *"4952 / 11"*. Assim, o elemento do multiconjunto cujo campo identificador é igual a "11" contém o resultado, que nesse caso é igual a 20, idêntico ao valor obtido pela execução do código C inicial. Entretanto, note que uma tupla adicional foi gerada no resultado final: $[1, 10, 0]$. Conforme podemos observar na Figura 5.2, trata-se de um elemento gerado pelo nó *Steer* mais a direita da Figura (cujas arestas de saída são iguais a "4808.T / 9" e "4808.F / 10"). Ocorre que as arestas de saída deste *Steer* não estão sendo utilizadas por outras instruções, desta forma o elemento em questão foi produzido e não consumido.

#### 5.1.3.3 Experimento 7 - Laços com *FOR*

Já neste caso de teste, o experimento tem o objetivo de analisar as respostas das execuções dos códigos equivalentes, sejam eles código C, *TALM* e Gamma tendo em vista o comportamento de um laço implementado com uma estrutura *"FOR"* em C. Considere o código C abaixo:

```
#BEGINBLOCK
   #include <stdio.h>
#ENDBLOCK

int main()
{
    int i, cont;
    cont = 7;

    for(i=0; i<10; i++)
        cont = cont + 1;
}
```

Código 5.15: Experimento 7 - Exemplo em Linguagem C.

A execução do código acima, tendo em vista as alterações necessárias conforme realizado nos Experimentos 5 e 6, que dizem respeito à retirada dos blocos *THLL* e impressão da resposta, apresentam o resultado 17, para a variável "cont" (após compilação pelo gcc).

A aplicação em linguagem de montagem gerado pelo *Couillard*, tendo em vista o código acima exposto é apresentado a seguir:

```
const flowInstConst140451325315120, 7
const flowInstConst140451325313896, 0
inctag flowInstIncTag140451325314256, [
   flowInstConst140451325313896, flowInstBinopI140451325370952]
lthani flowInstBinopI140451325313392,
   flowInstIncTag140451325314256, 10
```

```
5 inctag flowInstIncTag140451325316992, [
    flowInstConst140451325315120, flowInstBinopI140451325316848]
6 steer flowInstSteer140451325370520, flowInstBinopI140451325313392,
     flowInstIncTag140451325316992
7 addi flowInstBinopI140451325316848, flowInstSteer140451325370520.t
    , 1
8 steer flowInstSteer140451325371168, flowInstBinopI140451325313392,
     flowInstIncTag140451325314256
9 addi flowInstBinopI140451325370952, flowInstSteer140451325371168.t
    , 1
```

Código 5.16: Experimento 7 - Linguagem de Montagem do *TALM* após compilação pelo Couillard (a partir do Código 5.15).

A seguir apresentaremos o Código 5.17, obtido através da conversão do código acima pelo *GFlow*. Entretanto, algumas considerações merecem ser elencadas. Conforme mencionamos no Experimento 5, a utilização de operadores sequenciais na listagem de reações foi necessária diante de características das implementações de Gamma utilizadas. Como este exemplo trata-se de execução de um laço, as instruções (ou reações) devem executar mais de uma vez, ao passo que sempre que houver elementos no multiconjunto com *tags* (último campo da tupla que representa cada elemento no multiconjunto) iguais, tais elementos podem vir a reagir. Entretanto, pelo mesmo motivo apresentado no Experimento 5, uma reação que realiza teste condicional falso, pode vir a nunca mais ser escalonada para reagir. Desta maneira, além da utilização de operadores sequenciais, foi necessário replicar o bloco de reações (listagem de reações), por quantas vezes forem a quantidade de iterações do laço, visando sobrepor tais características encontradas nas implementações de Gamma utilizadas. Assim, este experimento replicou por doze vezes a listagem de reações. O código Gamma apresentado abaixo já contempla tais alterações, entretanto, só apresentados duas vezes a lista de reações, visando uma melhor visibilidade do código apresentado:

```
1 (R_flowInstIncTag140451325314256 | R_flowInstIncTag140451325316992
    ) ; (R_flowInstBinopI140451325313392_true |
    R_flowInstBinopI140451325313392_false) ; (
    R_flowInstSteer140451325370520_true |
    R_flowInstSteer140451325370520_false |
    R_flowInstSteer140451325371168_true |
    R_flowInstSteer140451325371168_false) ; (
    R_flowInstBinopI140451325316848 |
    R_flowInstBinopI140451325370952) ;
2
3 (R_flowInstIncTag140451325314256 | R_flowInstIncTag140451325316992
    ) ; (R_flowInstBinopI140451325313392_true |
    R_flowInstBinopI140451325313392_false) ; (
```

```
   R_flowInstSteer140451325370520_true |
   R_flowInstSteer140451325370520_false |
   R_flowInstSteer140451325371168_true |
   R_flowInstSteer140451325371168_false) ; (
   R_flowInstBinopI140451325316848 |
   R_flowInstBinopI140451325370952)

{[0, 1, 0], [7, 0, 0]}

/*  R_flowInstIncTag140451325314256 |
   R_flowInstBinopI140451325313392_true |
   R_flowInstBinopI140451325313392_false |
   R_flowInstIncTag140451325316992 |
   R_flowInstSteer140451325370520_true |
   R_flowInstSteer140451325370520_false |
   R_flowInstBinopI140451325316848 |
   R_flowInstSteer140451325371168_true |
   R_flowInstSteer140451325371168_false |
   R_flowInstBinopI140451325370952 {[0, 1, 0], [7, 0, 0]}  */

where

R_flowInstIncTag140451325314256 = replace [x, w, tag]
by [x, 2, tag+1], [x, 13, tag+1]
if (w == 1 or w == 10)

R_flowInstBinopI140451325313392_true = replace [x, 13, tag]
by [1, 3, tag], [1, 14, tag]
if (x  < 10)

R_flowInstBinopI140451325313392_false = replace [x, 13, tag]
by [0, 3, tag], [0, 14, tag]
if (x  >= 10)

R_flowInstIncTag140451325316992 = replace [x, w, tag]
by [x, 4, tag+1]
if (w == 0 or w == 7)

R_flowInstSteer140451325370520_true = replace [x, 14, tag], [y, 4,
    tag1]
by [y, 5, tag]
if (tag == tag1) and (x == 1)

R_flowInstSteer140451325370520_false = replace [x, 14, tag], [y,
   4, tag1]
by [y, 6, tag]
if (tag == tag1) and (x != 1)
```

```
34
35 R_flowInstBinopI140451325316848 = replace [x, 5, tag]
36 by [x + 1, 7, tag]
37 if true
38
39 R_flowInstSteer140451325371168_true = replace [x, 3, tag], [y, 2,
       tag1]
40 by [y, 8, tag]
41 if (tag == tag1) and (x == 1)
42
43 R_flowInstSteer140451325371168_false = replace [x, 3, tag], [y, 2,
        tag1]
44 by [y, 9, tag]
45 if (tag == tag1) and (x != 1)
46
47 R_flowInstBinopI140451325370952 = replace [x, 8, tag]
48 by [x + 1, 10, tag]
49 if true
```

Código 5.17: Experimento 7 - Código Gamma convertido a partir do Código 5.16.

Visando facilitar a visualização e explanação a respeito das conversões, fornecemos a Figura 5.3:

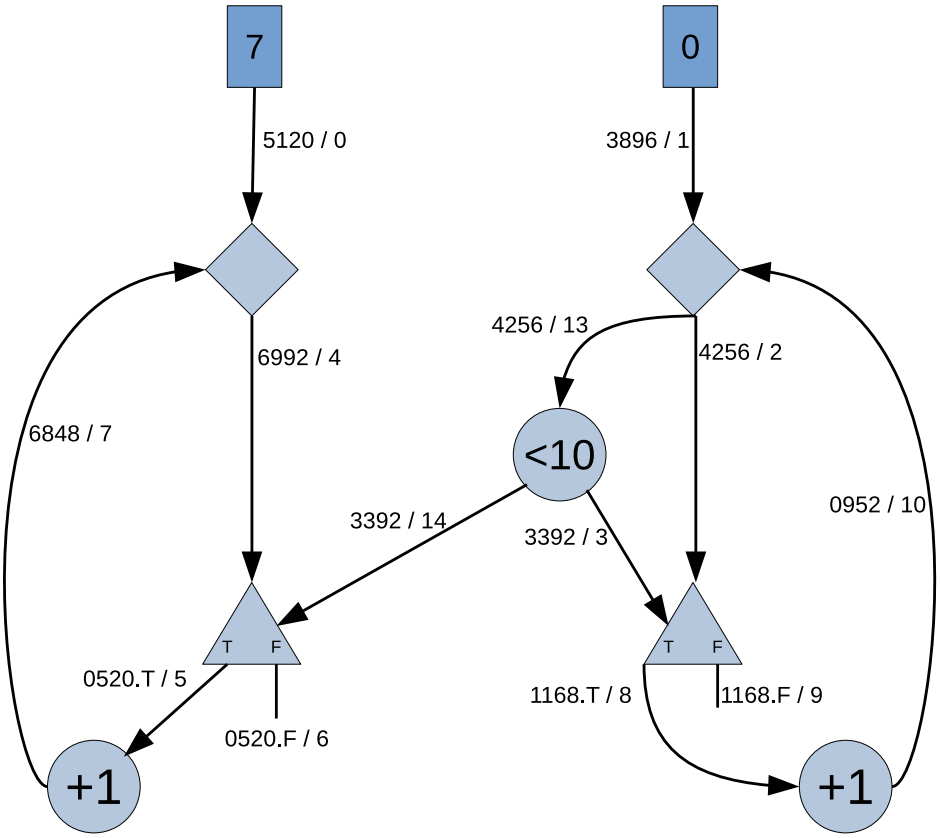


Figura 5.3: GFlow - Experimento 7 - Grafo Correspondente.

O resultado obtido pela execução do Código 5.17 nos ambientes Gamma utilizados foi: $[17, 6, 11]$ e $[10, 9, 11]$. As duas tuplas geradas referem-se às saídas *"false"* dos dois vértices do tipo *Steer* na Figura 5.3. Isso ocorre pois, após o incremento realizado pelo nó na parte inferior à direita da imagem (nó "+1"), alcançar o valor 10, a saída "false" de ambos *Steer* serão ativadas a cada iteração subsequente do laço. Ou seja, a partir do momento em que estas duas tuplas são geradas, nenhuma reação do código irá utilizar estes elementos para reagir (conforme a Figura 5.3 demonstra) e, portanto, nenhuma reação irá produzir novos dados. Repare ainda que o rótulo de iteração de ambas as tuplas é igual a 11, o que faz sentido, uma vez que antes do teste condicional, existe o incremento do rótulo de iteração (vértices *Inctag* do Grafo). Por fim, note que a resposta (valor 17) encontra-se na tupla cujo identificador é igual a 6, lembrando que a outra tupla remanescente (identificador 9) diz respeito à variável de controle de laço (variável "i" do Código 5.15). Dessa forma temos que o resultado encontrado do código Gamma executado confere com o valor 17 apresentado na execução do código C inicial.

#### 5.1.3.4 Experimento 8 - Laços com *WHILE*

Por fim, finalizando os experimentos afetos ao *GFlow* apresentaremos um laço que utiliza a construção *WHILE* em C, conforme podemos verificar no código abaixo:

```
#BEGINBLOCK
   #include <stdio.h>
#ENDBLOCK

int main() {
  int i, a, n, resultado;

  i = 0;
  a = 2;
  n = 5;
  resultado = 0;

  while (i<n) {
    i = i + 1;
    a = a + i;
    resultado = i * a;
  }
}
```

Código 5.18: Experimento 8 - Exemplo em Linguagem C.

O resultado da execução do código acima, compilado pelo gcc, após as adequações já mencionadas nos outros experimentos (remoção de anotações *THLL*) foi o

inteiro 85. Através da compilação do Código 5.18 pelo *Couillard*, temos o seguinte código *TALM*:

```
const flowInstConst139658106973936, 0
const flowInstConst139658106971704, 2
const flowInstConst139658106971128, 5
const flowInstConst139658106971488, 0
inctag flowInstIncTag139658107024664, [
    flowInstConst139658106973936, flowInstBinopI139658107024592]
inctag flowInstIncTag139658107023512, [
    flowInstConst139658106971128, flowInstSteer139658106577696.t]
lthan flowInstBinop139658106973792, flowInstIncTag139658107024664,
     flowInstIncTag139658107023512
steer flowInstSteer139658107025240, flowInstBinop139658106973792,
    flowInstIncTag139658107024664
addi flowInstBinopI139658107024592, flowInstSteer139658107025240.t
    , 1
inctag flowInstIncTag139658107025672, [
    flowInstConst139658106971704, flowInstBinop139658107025384]
steer flowInstSteer139658107025888, flowInstBinop139658106973792,
    flowInstIncTag139658107025672
add flowInstBinop139658107025384, flowInstSteer139658107025888.t,
    flowInstBinopI139658107024592
mult flowInstBinop139658107026320, flowInstBinopI139658107024592,
    flowInstBinop139658107025384
inctag flowInstIncTag139658107026824, [
    flowInstConst139658106971488, flowInstBinop139658107026320]
steer flowInstSteer139658107026968, flowInstBinop139658106973792,
    flowInstIncTag139658107026824
steer flowInstSteer139658106577696, flowInstBinop139658106973792,
    flowInstIncTag139658107023512
```

Código 5.19: Experimento 8 - Linguagem de Montagem do *TALM* após compilação pelo Couillard (a partir do Código 5.18).

Passando pelo processo de conversão entre modelos computacionais fornecido pelo *GFlow*, o código acima deu origem ao seguinte código Gamma:

```
(R_flowInstIncTag139658107024664 | R_flowInstIncTag139658107023512
     | R_flowInstIncTag139658107025672 |
    R_flowInstIncTag139658107026824) ; (
    R_flowInstBinop139658106973792_true |
    R_flowInstBinop139658106973792_false) ; (
    R_flowInstSteer139658107025240_true |
    R_flowInstSteer139658107025240_false |
    R_flowInstSteer139658107026968_true |
    R_flowInstSteer139658107026968_false |
```

```
   R_flowInstSteer139658106577696_true |
   R_flowInstSteer139658106577696_false  |
   R_flowInstSteer139658107025888_true |
   R_flowInstSteer139658107025888_false) ; (
   R_flowInstBinopI139658107024592 |
   R_flowInstBinop139658107025384) ;
   R_flowInstBinop139658107026320

{[0, 3, 0], [5, 2, 0], [0, 0, 0], [2, 1, 0]}

where

R_flowInstIncTag139658107024664 = replace [x, w, tag]
by [x, 4, tag+1], [x, 24, tag+1]
if (w == 0 or w == 49)

R_flowInstIncTag139658107023512 = replace [x, w, tag]
by [x, 5, tag+1], [x, 25, tag+1]
if (w == 2 or w == 18)

R_flowInstBinop139658106973792_true = replace [x, 24, tag], [y,
   25, tag1]
by [1, 6, tag], [1, 66, tag], [1, 46, tag], [1, 26, tag]
if (tag == tag1) and (x  <  y)

R_flowInstBinop139658106973792_false = replace [x, 24, tag], [y,
   25, tag1]
by [0, 6, tag], [0, 66, tag], [0, 46, tag], [0, 26, tag]
if (tag == tag1) and (x  >=  y)

R_flowInstSteer139658107025240_true = replace [x, 66, tag], [y, 4,
    tag1]
by [y, 7, tag]
if (tag == tag1) and (x == 1)

R_flowInstSteer139658107025240_false = replace [x, 66, tag], [y,
   4, tag1]
by [y, 8, tag]
if (tag == tag1) and (x != 1)

R_flowInstBinopI139658107024592 = replace [x, 7, tag]
by [x + 1, 9, tag], [x + 1, 49, tag], [x + 1, 29, tag]
if true

R_flowInstIncTag139658107025672 = replace [x, w, tag]
by [x, 10, tag+1]
if (w == 1 or w == 33)
```

```
38
39 R_flowInstSteer139658107025888_true = replace [x, 46, tag], [y,
       10, tag1]
40 by [y, 11, tag]
41 if (tag == tag1) and (x == 1)
42
43 R_flowInstSteer139658107025888_false = replace [x, 46, tag], [y,
       10, tag1]
44 by [y, 12, tag]
45 if (tag == tag1) and (x != 1)
46
47 R_flowInstBinop139658107025384 = replace [x, 11, tag], [y, 29,
       tag1]
48 by [x + y, 13, tag], [x + y, 33, tag]
49 if (tag == tag1)
50
51 R_flowInstBinop139658107026320 = replace [x, 9, tag], [y, 13, tag1
       ]
52 by [x * y, 14, tag]
53 if (tag == tag1)
54
55 R_flowInstIncTag139658107026824 = replace [x, w, tag]
56 by [x, 15, tag+1]
57 if (w == 3 or w == 14)
58
59 R_flowInstSteer139658107026968_true = replace [x, 26, tag], [y,
       15, tag1]
60 by [y, 16, tag]
61 if (tag == tag1) and (x == 1)
62
63 R_flowInstSteer139658107026968_false = replace [x, 26, tag], [y,
       15, tag1]
64 by [y, 17, tag]
65 if (tag == tag1) and (x != 1)
66
67 R_flowInstSteer139658106577696_true = replace [x, 6, tag], [y, 5,
       tag1]
68 by [y, 18, tag]
69 if (tag == tag1) and (x == 1)
70
71 R_flowInstSteer139658106577696_false = replace [x, 6, tag], [y, 5,
        tag1]
72 by [y, 19, tag]
73 if (tag == tag1) and (x != 1)
```

Código 5.20: Experimento 8 - Código Gamma convertido a partir do Código 5.19.

Da mesma maneira como ocorreu no Experimento 7, também foi necessário utilizar operadores sequenciais e replicar a listagem de reações por 6 vezes. Entretanto, o Código 5.20 acima só apresenta a listagem de reações uma única vez, visando facilitar a visualização do código apresentado.

Para facilitar a visualização e conclusões acerca da conversão e resultados obtidos neste experimento, fornecemos a Figura 5.4 abaixo:

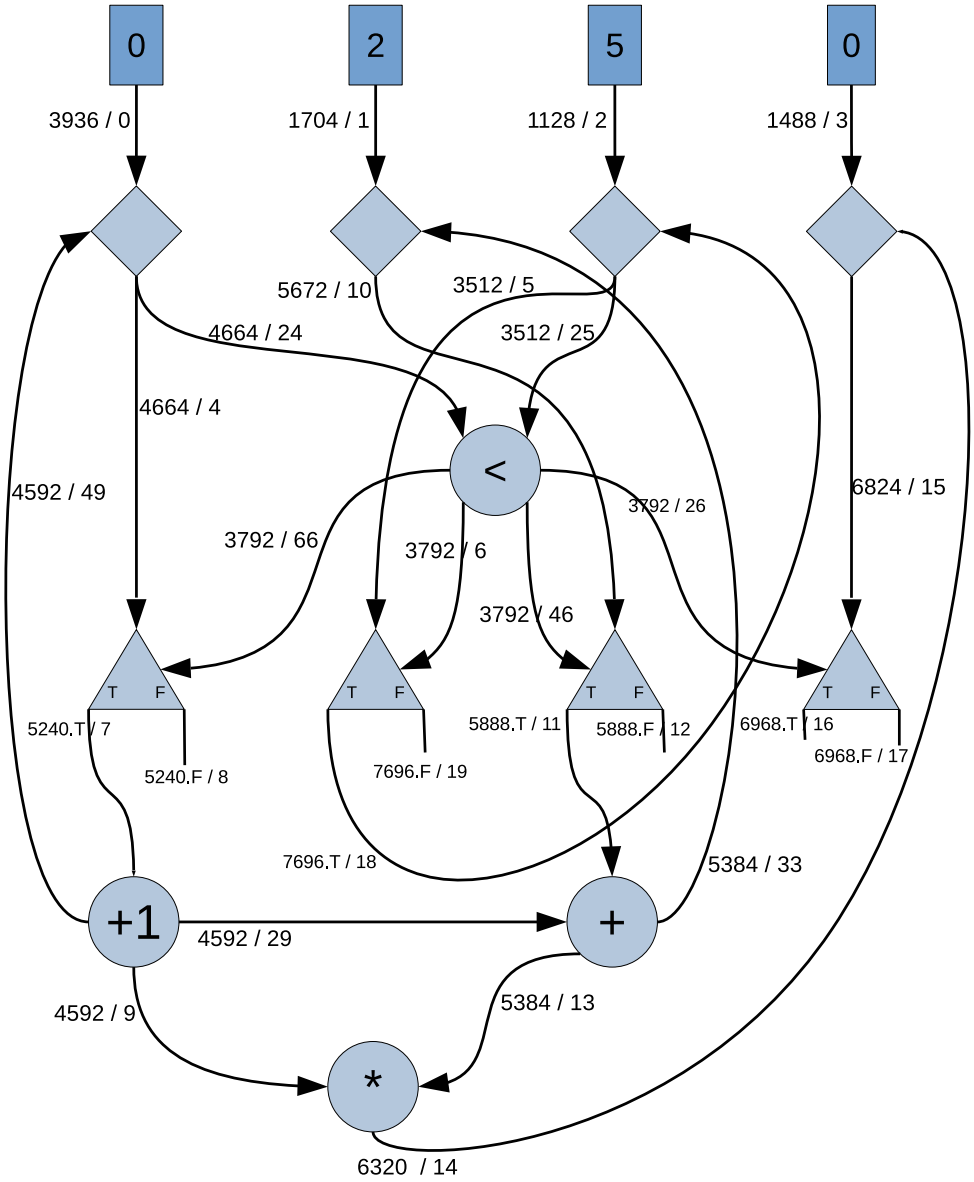


Figura 5.4: GFlow - Experimento 8 - Grafo Correspondente.

O resultado do programa Gamma (Código 5.20) submetido à execução pelas implementações de Juarez Muylaert foi igual a : $[17, 12, 6]$, $[5, 19, 6]$, $[85, 17, 6]$, $[5, 8, 6]$, $[48, 16, 5]$, $[24, 16, 4]$, $[10, 16, 3]$, $[3, 16, 2]$, $[0, 16, 1]$. O resultado, que apresentou mesmo valor da execução do código C inicial, foi igual a 85, identificado na tupla

$[85, 17, 6]$. Na Figura 5.4, tal tupla corresponde ao dado que é produzido pelo vértice que executa a multiplicação e posterior incremento de seu rótulo de iteração, antes de fornecer dados ao *Sterr* mais a direita da Figura. Entretanto, alguns comentários sobre as demais tuplas restantes na solução final são necessários. As tuplas cujo identificador é igual a 16 ($[48, 16, 5]$, $[24, 16, 4]$, $[10, 16, 3]$, $[3, 16, 2]$, $[0, 16, 1]$), dizem respeito aos dados produzidos pela saída *"true"* do *Steer* mais a direita da Figura. Note que enquanto o teste condicional do vértice central da figura (cuja operação é "$<$") for verdadeiro, serão produzidos pelo *Steer* em questão elementos (tuplas) com identificadores iguais a 16. Repare que foram produzidas 5 tuplas com este identificador, uma para cada iteração (vide rótulo de iteração das tuplas) do laço *WHILE* do código C inicial. A partir do momento em que o teste condicional é falso, tal *Sterr* produz elemento através de sua saída *"false"*, cujo rótulo de iteração será 6 e identificador 17. Repare ainda que as demais tuplas geradas referem-se à elementos gerados pelos demais *Steer* do código que não foram consumidos por outras reações (tuplas cujos identificadores são iguais a: 8, 12 e 19.

Note que posteriormente poderá ser proposto um procedimento visando eliminar estas tuplas não utilizadas no código, seja por execução de reações que eliminam estes elementos, seja por alterações no códigos das reações geradas pelo *GFlow*. Entretanto, para nos manter fiel ao código gerado pelo *TALM* e a proposta de conversão, optamos em converter exatamente toda e qualquer instrução da linguagem de montagem em reação (ou reações) Gamma.

Tendo em vista o exposto e comparando os resultados obtidos tanto no ambiente Gamma quanto no código compilado pelo gcc, vimos que a solução está correta.

## 5.2 GSink

### 5.2.1 Procedimentos Experimentais

Para os experimentos selecionados para o *GSink*, utilizamos uma abordagem empírica, onde o objetivo foi apresentar a corretude das execuções dos programas Gamma. Dessa maneira, comparamos os resultados das execuções do nosso ambiente com aqueles obtidos através das implementações *Gamma-Sequencial* e *Gamma-MPI*. Além disso, desenvolvemos casos de teste visando apresentar o potencial de desempenho de Gamma à medida em que a granularidade das reações aumente. Assim, dois grupos distintos de experimentos foram propostos.

A Seção 5.2.2, contém experimentos que dedicam-se à verificação e comparação dos resultados obtidos pela execução dos programas Gamma. Além disso, observaremos alguns detalhes importantes a respeito da execução e montagem do código C, realizada pelo *Front-End*. Para tanto, os resultados obtidos foram comparados com *Gamma-Sequencial* e *Gamma-MPI*, sem a realização de análise de desempenho, por tratar-se de análise da corretude da execução.

Já a Seção 5.2.3, visa apresentar o potencial de desempenho de nosso ambiente de execução, a medida em que a granularidade das reações Gamma aumente. Dessa maneira, inserimos um custo computacional maior do que somente uma única operação aritmética ou lógica no escopo de uma reação. Em outras palavras, sem prejuízo à corretude da execução das reações, foi inserido um cálculo de multiplicação matricial, onde a dimensão das matrizes foi alterada visando análise do experimento. Dessa forma, inserimos manualmente, no escopo das funções que executam as reações nas implementações de Juarez Muylaert, tal cálculo, sem prejuízo à corretude e com única intenção de aumento do custo computacional. Da mesma forma, no âmbito das *threads* que executam as instâncias de reações contidas nos nossos *sinks*, foi inserido o mesmo cálculo computacional, com o mesmo objetivo de aumento do custo computacional.

Por fim, para os casos de teste idealizados para fins de corretude (Seção 5.2.2), cada experimento foi executado 10 vezes, seja no *GSink*, seja nas duas implementações de Gamma citadas. Já para a segunda categoria de experimentos (Seção 5.2.3), por tratar-se de cálculo de tempo de execução, utilizamos de 10 a 30 execuções, visando a diminuição do desvio padrão [61] e [18]. Além disso, o maior e menor valor de tempo de execução foram descartados, onde a média passou a ser calculada com os tempos restantes.

### 5.2.2 Corretude da Execução

Esta Seção é destinada a condução dos experimentos para o *GSink*. Trata-se de alguns exemplos de códigos Gamma, o qual iremos submeter à execução pelas implementações *Gamma-Sequencial*, *Gamma-MPI* e *GSink*. Procuraremos chamar atenção, quando necessário, para aspectos que dizem respeito ao código fonte gerado pelo *Front-End* (*"main.c"*), tendo em vista a montagem de tal código, através da extração de informações da árvore sintática.

#### 5.2.2.1 Experimento 1

Para este primeiro experimento, propomos um código Gamma que calcula os números primos dado o multiconjunto inicial formado pelos 100 primeiros inteiros. A aplicação é conhecida como o Crivo de Eratóstenes (*"Sieve of Eratosthenes"*). O Código 5.21 abaixo apresenta o programa Gamma correspondente:

```
1 sieve {2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
2 21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
3 41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,
4 61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,
5 81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100}
6
7 where
8
9 sieve = replace x1, x2 by x2 if x1 % x2 == 0
```

Código 5.21: Experimento 1 - Crivo de Erastótenes.

O código Gamma acima contém uma única reação que seleciona elementos do multiconjunto dois a dois e mantém o primeiro, caso o módulo do primeiro pelo segundo seja igual a zero. Tal exemplo corrobora para a afirmação de que Gamma provê uma forma simples e elegante de especificar programas paralelos.

Os resultados obtidos pelas implementações *Gamma-Sequencial*, *Gamma-MPI* e *GSink* foram idênticos e serão expostos abaixo. Devido ao fato dos elementos do multiconjunto não guardarem nenhuma ordenação entre si, a impressão dos resultados em cada ambiente pode variar pela ordem dos elementos, entretanto, os multiconjuntos são iguais:

***Gamma-Sequencial:***

```
{ 3, 2, 5, 17, 37, 19, 11, 41, 43, 7, 23, 47, 97, 89, 83, 79, 73,
71, 67, 61, 59, 53, 31, 29, 13 }
```

***Gamma-MPI:***

```
{ 97, 89, 83, 79, 73, 71, 67, 61, 59, 53, 47, 43, 41, 37, 31, 29,
23, 19, 17, 13, 11, 7, 5, 3, 2 }
```

***GSink:***

```
########################
END OF THE PROGRAM!

Final Multiset:

Id (172) Element (3)
Id (171) Element (13)
Id (170) Element (11)
Id (169) Element (5)
Id (168) Element (7)
Id (166) Element (17)
Id (165) Element (2)
Id (164) Element (19)
Id (152) Element (23)
Id (145) Element (37)
Id (136) Element (29)
Id (135) Element (31)
Id (95) Element (97)
Id (87) Element (89)
Id (81) Element (83)
Id (77) Element (79)
Id (71) Element (73)
Id (69) Element (71)
Id (65) Element (67)
Id (59) Element (61)
Id (57) Element (59)
Id (51) Element (53)
Id (45) Element (47)
Id (41) Element (43)
Id (39) Element (41)
########################
```

Observe que para os três resultados apresentados acima, os formatos estão apresentados propositalmente de maneiras distintas. Isso deve-se ao fato que querermos reproduzir exatamente as saídas produzidas pelos ambientes citados. Para os próximos experimentos apresentaremos somente a resposta, sem o formato específico apresentado por cada *Runtime*. Repare que para o *GSink*, cada elemento do multiconjunto é impresso juntamente com seu identificador.

Note que para os três ambientes citados, os valores encontrados como números primos entre os 100 primeiros inteiros estão corretos, implicando na corretude da resposta de cada *Runtime*. Agora, analisaremos detalhes do código *".c"* gerado por ocasião da execução do *GSink*, que pode ser encontrado em sua íntegra no repositório público https://gitlab.com/rui.rmj/dsc-experimentos.

Inicialmente, foram gerados corretamente as inclusões de *Headers* necessárias à compilação e execução do código. Da mesma forma, a declaração e inicialização das variáveis. Como o código Gamma é composto de somente uma reação (Código 5.21 linha 9), foi criado somente um *ReAgente*. Assim, somente uma lista de instâncias foi criada, assim como uma única cópia do multiconjunto e uma lista contendo histórico de instâncias:

```
reg_I * instances_I1 ; // Instances List (I1)
reg_multiset * multiset_M1 ; // Copy of Multiset Elements List (M1)
reg_I * usedinstances_I1 ; // Used Instances List (I1) - store the
history of instances provided
```

O multiconjunto também foi corretamente inicializado, tendo em vista as entradas no *"main.c"* contendo a utilização da função que adiciona elementos ao multiconjunto (***pm = AddMultisetElement (& id_ element , x);*** ) onde *"x"* refere-se ao valor de cada elemento inicial do multiconjunto.

As demais definições apresentadas no código também encontram-se corretas, com destaque para a função executada pelas *threads*. Conforme mencionado anteriormente, como temos somente uma reação no código Gamma correspondente, somente um *ReAgente* foi inicializado, com sua cópia do Multiconjunto, lista de instâncias e histórico de instâncias utilizado. Pelo mesmo motivo da existência de somente uma reação no código correspondente, somente uma função será executada pelas *threads*, que diz respeito à execução das instâncias da reação chamada *"sieve"* no código equivalente. Tal função foi montada corretamente e será descrita abaixo no Código 5.22:

```
1 /*************************************************************/
2 /*******  Function related to the R1 reaction execution ********/
3 /*************************************************************/
4 int R1Function(reg_sink *ps){
5     // Name of the gamma reaction - according the gamma code: sieve
6     int x1, x2, result;
```

```
7     reg_multiset *pm;
8     reg_I element_Ix, *pi;
9
10    x1  = ps->data.element1;
11    x2  = ps->data.element2;
12    element_Ix.data.element1 = ps->data.element1;
13    element_Ix.data.element2 = ps->data.element2;
14    result = x2;
15
16    if (((x1 % x2) ==  0 )) {
17        sem_wait(&sem_multiset);
18        pm = AddMultisetElement (&id_element, result);
19        pi = SearchInstanceIx(&instances_I1, element_Ix);
20        DeleteMultisetElementId (pi->id_data.id1);
21        DeleteMultisetElementId (pi->id_data.id2);
22        sem_post(&sem_multiset);
23        return TRUE;
24    }
25    else {
26        return FALSE;
27    }
28 }
```

Código 5.22: Experimento 1 - Extrato do *main.c* - função *R1function.*

Note que o nome da reação foi extraído da árvore sintática e está apresentado como comentário no código, para fins de *Debug* (linha 5). Entretanto, o padrão para nome das funções a serem executadas pelas *threads* é *"R"* seguido de um inteiro, concatenado à *string* "Function" neste caso, $R1Function$. Repare que extraímos da árvore sintática o nome das variáveis utilizadas no código Gamma (linha 6) e a própria condição de reação (linha 16).

Por fim, vemos que todas as demais construções estão corretas, seguindo o que foi preconizado na Seção 4.3 que apresenta o *GSink*, incluindo aspectos de implementação. Tal fato é também corroborado pela corretude da resposta apresentada anteriormente.

#### 5.2.2.2 Experimento 2

O segundo experimento proposto para o *GSink* calcula o maior elemento dado um multiconjunto qualquer, que admite a existência de elementos repetidos e onde não existe ordenação entre os mesmos, conforme apresentado no Código 5.23 abaixo:

```
1 max {1,2,3,4,5,6,7,8,9,1,1,1,1,1,11,1,111,434,34234,345,66,554,
2 25,26,26,28,33}
3
4 where
5
6 max = replace x,y by x if x >= y
```

Código 5.23: Experimento 2 - Cálculo do maior elemento do Multiconjunto.

A única reação que consta do código acima (*"max"*) seleciona dois elementos quaisquer no multiconjunto e remove o segundo, caso a condição de reação (que neste caso verifica se o primeiro elemento selecionado é maior ou igual ao segundo) seja atendida.

O resultado apresentado pela execução do código acima nos três ambientes selecionados, incluindo o *GSink* foi igual a 34234, o que apresenta o resultado correto e atesta a corretude de execução entre as implementações de Gamma citados.

Da mesma forma como no primeiro experimento, temos a criação de somente um *ReAgente*. Sendo assim, no correspondente código *"main.c"* gerou-se somente uma lista de instâncias, uma cópia do Multiconjunto e uma lista de histórico de instâncias utilizadas. Exatamente por isso, somente foram geradas instâncias para o primeiro *ReAgente*, de maneira correta:

```
GeneratesInstancesRx(&multiset_M1, &instances_I1, &usedinstances_I1);
```

Por fim, a função gerada para ser executada pelas *threads* que contém as instâncias de execução também executaram a extração de informações da árvore sintática corretamente, como podemos observar no Código 5.24 abaixo:

```
1  /*****************************************************************/
2  /*******  Function related to the R1 reaction execution ********/
3  /*****************************************************************/
4  int R1Function(reg_sink *ps){
5    // Name of the gamma reaction - according the gamma code: max
6    int x, y, result;
7    reg_multiset *pm;
8    reg_I element_Ix, *pi;
9
10   x  = ps->data.element1;
11   y  = ps->data.element2;
12   element_Ix.data.element1 = ps->data.element1;
13   element_Ix.data.element2 = ps->data.element2;
14   result = x;
15
16   if ((x >= y)) {
17     sem_wait(&sem_multiset);
18     pm = AddMultisetElement (&id_element, result);
19     pi = SearchInstanceIx(&instances_I1, element_Ix);
20     DeleteMultisetElementId (pi->id_data.id1);
21     DeleteMultisetElementId (pi->id_data.id2);
22     sem_post(&sem_multiset);
23     return TRUE;
24   }
25   else {
26     return FALSE;
27   }
28 }
```

Código 5.24: Experimento 2 - Extrato do *main.c* - função *R1function*.

Note que o cálculo necessário à reação "max" é corretamente expresso pela função acima, onde a condição de reação encontra-se em uma estrutura de decisão contida na linha 16 do Código 5.24. Além disso, as variáveis utilizadas seguem a nomenclatura daquelas extraídas da árvore sintática (x,y).

#### 5.2.2.3 Experimento 3

Nosso último caso de teste contendo somente uma reação calcula o fatorial de um determinado inteiro. Desta forma, o Código 5.25 abaixo, apresenta tal aplicação escrita de acordo com o paradigma Gamma:

```
1 factorial { 1,2,3,4,5,6,7,8,9,10 }
2
3 where
4
5 factorial  = replace x,y by x*y if true
```

Código 5.25: Experimento 3 - Cálculo do Fatorial.

Veja que, para esta reação, a condição de reação será sempre verdadeira, a medida em que puderem ser escolhidos dois elementos para serem selecionados. Assim, a condição de parada da computação será alcançada quando o multiconjunto for composto por somente um elemento.

O resultado obtido da execução do código acima nos ambientes *Gamma-Sequencial*, *Gamma-MPI* e *GSink* foi o mesmo e igual a 3628800. Assim, o resultado das execuções encontra-se correto.

Da mesma forma, analisando o código C montado a partir do *Front-End* do *GSink*, verificamos a corretude das montagens de código referentes ao *ReAgente* e outras construções. Como tal experimento utiliza-se de somente uma reação, os comentários anteriores para os dois primeiros experimentos com relação à criação de somente um *ReAgente*, também são válidos para este terceiro experimento.

Por fim, apresentamos a função executada pelas *threads* disparadas pelos *sinks* do mecanismo de escalonamento proposto para o *GSink*:

```
1  /*************************************************************/
2  /*******  Function related to the R1 reaction execution ********/
3  /*************************************************************/
4  int R1Function(reg_sink *ps){
5    // Name of the gamma reaction - according the gamma code: factorial
6    int x, y, result;
7    reg_multiset *pm;
8    reg_I element_Ix, *pi;
9
10   x  = ps->data.element1;
11   y  = ps->data.element2;
12   element_Ix.data.element1 = ps->data.element1;
13   element_Ix.data.element2 = ps->data.element2;
14   result = (x * y);
```

```
15
16   if (1) {
17     sem_wait(&sem_multiset);
18     pm = AddMultisetElement (&id_element, result);
19     pi = SearchInstanceIx(&instances_I1, element_Ix);
20     DeleteMultisetElementId (pi->id_data.id1);
21     DeleteMultisetElementId (pi->id_data.id2);
22     sem_post(&sem_multiset);
23     return TRUE;
24   }
25   else {
26     return FALSE;
27   }
28 }
```

Código 5.26: Experimento 3 - Extrato do *main.c* - função *R1function.*

A função *R1Function* (apresentada no Código 5.26) gerada a partir do código Gamma 5.25 descrito anteriormente, também está correta conforme extrato reproduzido acima. Podemos verificar que a montagem do código respeita as informações apresentadas no código Gamma originário.

#### 5.2.2.4 Experimento 4

Para o quarto experimento proposto, analisaremos um programa Gamma que realiza um cálculo hipotético, através de um código composto por três reações distintas. O multiconjunto é extenso e formado por todos os elementos que compreendem os seguintes intervalos de inteiros:
$\{x \in \mathbb{Z} \mid 1 \leq x \leq 200\}$
$\{x \in \mathbb{Z} \mid 21000 \leq x \leq 22000\}$
$\{x \in \mathbb{Z} \mid 23000 \leq x \leq 23200\}$

Pela extensão do multiconjunto, que faz parte do programa conforme a sintaxe de Gamma que utilizamos, apresentaremos no Código 5.27 somente as três reações que compõem o programa, lembrando que o Código ".gm" está integralmente disponível no repositório público https://gitlab.com/rui.rmj/dsc-experimentos:

```
1 first | second | third { /* Multiconjunto omitido */ }
2
3 where
4
5 first = replace x,y by x+y if (x <= 20100) and (y <= 20100)
6
7 second = replace m,n by m if (m > n) and (m >= 21000) and (m <=
    22000) and (n >= 21000) and (n <= 22000)
8
9 third = replace a,b by (a+b)-10000 if (a >= 23000) and (b >=
    23000)
```

Código 5.27: Experimento 4 - Cálculo Hipotético, com três reações.

As três reações propostas para este experimento não interferem no subconjunto de elementos manipulados pelas outras reações. Ou seja, foram definidos intervalos dentro do multiconjunto de modo que os elementos gerados por uma reação não sejam úteis para as outras duas reações. Assim, a primeira reação, *"first"* realiza a soma de elementos, desde que estes sejam maiores ou iguais a 20100. Assim, como tal reação efetuará a soma dos duzentos primeiros inteiros, o valor de tal operação será de 20100. Repare agora que a segunda reação (*"second"*), somente utiliza elementos existentes no segundo intervalo (entre 21000 e 22000) conforme expresso em sua condição de reação, no Código 5.27, linha 7. Como esta segunda reação seleciona o maior elemento deste intervalo, novos elementos não serão gerados por ela, somente serão excluídos elementos até que se tenha o maior elemento do intervalo. Por fim, a terceira reação, *"third"* realiza um cálculo específico somente se os elementos selecionados forem maiores que 23000. Repare que tal cálculo cria elementos pertencentes ao terceiro intervalo de elementos, não gerando elementos que possam vir a ser selecionados por outras reações.

Repare que o motivo da separação de intervalos de elementos dentro de um mesmo multiconjunto tem o objetivo de facilitar a apresentação da resposta, através de um melhor controle e visualização dos elementos do multiconjunto. Em outras palavras, para fins de *Debug* e demonstração dos resultados é mais simples definir os limites de atuação de cada reação, no escopo dos subconjuntos de elementos que estas utilizam.

Dessa forma, o resultado obtido pela execução do Código 5.27 nos ambientes *Gamma-Sequencial*, *Gamma-MPI* e *GSink* foram iguais e correspondem ao seguinte:

```
{20100, 22000, 2643100}
```

O resultado está correto, uma vez que 20100 corresponde a soma dos 200 primeiros inteiros, 22000 é o elemento maior do segundo intervalo definido dentro do escopo do multiconjunto inicial e 2643100 corresponde ao resultado da operação (a+b)-10000, sobre os elementos do terceiro intervalo inicialmente definido por: $\{23000, 23001, ..., 23200\}$. Assim, podemos atestar a corretude da execução deste quarto experimento e a igualdade das respostas apresentadas pelos ambientes de execução.

Como nosso código é formado por três reações distintas, é importante analisar a montagem do Código C (*"main.c"*) tendo em vista tal quantidade de reações. É importante lembrar que o *GSink* possui um *Front-End*, responsável pela montagem do código *"main.c"*, que será submetido ao ambiente de execução do *GSink* baseado em um grafo de instâncias de reações. Inicialmente, verificamos corretamente a geração de estruturas de dados a serem utilizadas pelos três *ReAgentes* necessários a execução:

```
reg_I *instances_I1;        // Instances List (I1)
reg_I *instances_I2;        // Instances List (I2)
reg_I *instances_I3;        // Instances List (I3)
reg_multiset *multiset_M1;  // Copy of Multiset Elements List (M1)
reg_I *usedinstances_I1;    // Used Instances List (I1) - store the
history of instances provided
reg_multiset *multiset_M2;  // Copy of Multiset Elements List (M2)
reg_I *usedinstances_I2;    // Used Instances List (I2) - store the
history of instances provided
reg_multiset *multiset_M3;  // Copy of Multiset Elements List (M3)
reg_I *usedinstances_I3;    // Used Instances List (I3) - store the
history of instances provided
```

Da mesma forma, a declaração das funções utilizadas pelas *threads* foram corretamente realizadas:

```
int R1Function(reg_sink *ps);
int R2Function(reg_sink *ps);
int R3Function(reg_sink *ps);
```

As inclusões de linhas de código, referentes à inserção dos elementos do multiconjunto também apresentam-se corretas, da mesma maneira que os experimentos anteriores, através da utilização da função implementada para adicionar os elementos no multiconjunto: (***pm = AddMultisetElement (& id_ element , x);*** ) onde *"x"* refere-se ao valor de cada elemento inicial do multiconjunto.

As variáveis e estruturas de dados foram inicializadas e declaradas. Além disso, a geração de instâncias foi corretamente realizada, para cada *ReAgente*:

```
GeneratesInstancesRx(&multiset_M1, &instances_I1, &usedinstances_I1);
GeneratesInstancesRx(&multiset_M2, &instances_I2, &usedinstances_I2);
GeneratesInstancesRx(&multiset_M3, &instances_I3, &usedinstances_I3);
```

Com relação a geração do Grafo, foram criados blocos de código contendo os *loops* necessários à inclusão das instâncias de cada *ReAgente* como vértices deste grafo. Tal grafo é o regente da execução no *GSink*, onde a identificação e posterior execução dos *sinks* permitirá a execução do programa Gamma correspondente.

Agora, tomemos como base o Código 5.28, que apresenta o extrato das funções *"R1Function"*, *"R2Function"* e *"R3Function"*, do código gerado pelo *Front-End* do *GSink*:

```
/*****************************************************************/
/*******  Function related to the R1 reaction execution ********/
/*****************************************************************/
int R1Function(reg_sink *ps){
  // Name of the gamma reaction - according the gamma code: first
  int x, y, result;
  reg_multiset *pm;
  reg_I element_Ix, *pi;

  x  = ps->data.element1;
  y  = ps->data.element2;
  element_Ix.data.element1 = ps->data.element1;
  element_Ix.data.element2 = ps->data.element2;
  result = (x + y);

  if (((x <=  20100 ) && (y <=  20100 ))) {
    sem_wait(&sem_multiset);
    pm = AddMultisetElement (&id_element, result);
    pi = SearchInstanceIx(&instances_I1, element_Ix);
    DeleteMultisetElementId (pi->id_data.id1);
    DeleteMultisetElementId (pi->id_data.id2);
    sem_post(&sem_multiset);
    return TRUE;
  }
  else {
    return FALSE;
  }
}

/*****************************************************************/
/*******  Function related to the R2 reaction execution ********/
/*****************************************************************/
int R2Function(reg_sink *ps){
  // Name of the gamma reaction - according the gamma code: second
  int m, n, result;
  reg_multiset *pm;
  reg_I element_Ix, *pi;

  m  = ps->data.element1;
  n  = ps->data.element2;
  element_Ix.data.element1 = ps->data.element1;
  element_Ix.data.element2 = ps->data.element2;
  result = m;

  if ((((((m > n) && (m >=  21000 )) && (m <=  22000 )) && (n >=  21000 )) && (n
    <=  22000 ))) {
    sem_wait(&sem_multiset);
    pm = AddMultisetElement (&id_element, result);
    pi = SearchInstanceIx(&instances_I2, element_Ix);
    DeleteMultisetElementId (pi->id_data.id1);
    DeleteMultisetElementId (pi->id_data.id2);
    sem_post(&sem_multiset);
    return TRUE;
  }
  else {
    return FALSE;
  }
}

```

```
59 /*************************************************************/
60 /*******  Function related to the R3 reaction execution ********/
61 /*************************************************************/
62 int R3Function(reg_sink *ps){
63   // Name of the gamma reaction - according the gamma code: third
64   int a, b, result;
65   reg_multiset *pm;
66   reg_I element_Ix, *pi;
67 
68   a  = ps->data.element1;
69   b  = ps->data.element2;
70   element_Ix.data.element1 = ps->data.element1;
71   element_Ix.data.element2 = ps->data.element2;
72   result = ((a + b) -  10000 );
73 
74   if (((a >=  23000 ) && (b >=  23000 ))) {
75     sem_wait(&sem_multiset);
76     pm = AddMultisetElement (&id_element, result);
77     pi = SearchInstanceIx(&instances_I3, element_Ix);
78     DeleteMultisetElementId (pi->id_data.id1);
79     DeleteMultisetElementId (pi->id_data.id2);
80     sem_post(&sem_multiset);
81     return TRUE;
82   }
83   else {
84     return FALSE;
85   }
86 }
```

Código 5.28: Experimento 4 - Extrato do *main.c* - funções executadas pelas *threads*.

Note que as funções foram corretamente geradas. As condições para execução de cada reação foram corretamente extraídas do código Gamma correspondente, como podemos verificar nas linhas 16, 45 e 74 do Código 5.28). Note ainda que a nomenclatura das variáveis utilizadas foi extraída do código Gamma correspondente conforme as declarações de variáveis das linhas 6, 35 e 64 do Código 5.28).

Desta forma, pela verificação dos resultados apresentados e pela análise das construções fornecidas pelo nosso *Front-End*, podemos afirmar que o experimento 4 apresentou execução e montagem de códigos corretos.

#### 5.2.2.5 Experimento 5

Para este último caso de teste, da primeira categoria de experimentos criados para o *GSink*, utilizamos um código Gamma composto por duas reações, $R1$ e $R2$. Ao contrário do que foi apresentado no experimento anterior, aqui as reações interferem nos elementos do multiconjunto que podem ser utilizados entre si. Ou seja, $R1$ pode consumir e criar elementos que podem ser utilizados por $R2$ e vice-e-versa. Assim, os aspectos de corretude analisados aqui dizem respeito somente às construções inseridas no (*"main.c"*) a ser executado no *Runtime* do *GSink*. Tomemos como base, o seguinte código Gamma (Código 5.29), referente ao nosso quinto experimento:

```
R1 | R2  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
   17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
   32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
   47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
   62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
   77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
   92, 93, 94, 95, 96, 97, 98, 99, 100 }

where

R1 = replace x,y by x+y if x>20

R2 = replace a,b by a*b if (a+b)<100
```

Código 5.29: Experimento 5 - Código Gamma com duas reações hipotéticas.

A primeira reação realiza a soma de elementos selecionados, desde que o primeiro seja maior que 20. Já a segunda reação realiza a multiplicação dos elementos escolhidos pela cláusula *REPLACE*, desde que a soma destes seja menor que o valor 100. A computação expressa nas reações descritas não possuem objetivo de realizar nenhum cálculo específico.

O resultado das execuções do Código 5.29 nas implementações *Gamma-Sequencial*, *Gamma-MPI* e *GSink* obtiveram respostas distintas. O motivo de tal discrepância de resultados é a diferença entre os algoritmos de escalonamento da execução utilizado. Na medida em que a versões *Gamma-Sequencial* e *Gamma-MPI* utilizam mecanismos que "entregam" o multiconjunto para uma reação por vez (mantendo as devidas diferenças de execução distribuída e sequencial), o *GSink* utiliza-se de um grafo contendo instâncias de todas as reações envolvidas. Assim, em cada mecanismo de escalonamento utilizado, o multiconjunto vai sendo modificado, em tempo de execução, de maneira diferente.

Conforme verificado nos códigos gerados pelo *Front-End* do *GSink*, verificamos a corretude nas construções do código C a ser executado no *Runtime* do *GSink*. Assim as declarações e inicializações de variáveis e estrutura de dados utilizadas pelos *ReAgentes*, que para este exemplo totalizam 2, construção do grafo e demais blocos de código, estão corretos. Abaixo, o Código 5.30 apresenta as funções criadas para execução das *threads* responsáveis pelas instâncias de *R1* e *R2*:

```
/*************************************************************/
/*******  Function related to the R1 reaction execution ********/
/*************************************************************/
int R1Function(reg_sink *ps){
  // Name of the gamma reaction - according the gamma code: R1
  int x, y, result;
  reg_multiset *pm;
  reg_I element_Ix, *pi;
```

```
9
10    x  = ps->data.element1;
11    y  = ps->data.element2;
12    element_Ix.data.element1 = ps->data.element1;
13    element_Ix.data.element2 = ps->data.element2;
14    result = (x + y);
15
16    if ((x >  20 )) {
17      sem_wait(&sem_multiset);
18      pm = AddMultisetElement (&id_element, result);
19      pi = SearchInstanceIx(&instances_I1, element_Ix);
20      DeleteMultisetElementId (pi->id_data.id1);
21      DeleteMultisetElementId (pi->id_data.id2);
22      sem_post(&sem_multiset);
23      return TRUE;
24    }
25    else {
26      return FALSE;
27    }
28  }
29
30  /*****************************************************************/
31  /*******  Function related to the R2 reaction execution ********/
32  /*****************************************************************/
33  int R2Function(reg_sink *ps){
34    // Name of the gamma reaction - according the gamma code: R2
35    int a, b, result;
36    reg_multiset *pm;
37    reg_I element_Ix, *pi;
38
39    a  = ps->data.element1;
40    b  = ps->data.element2;
41    element_Ix.data.element1 = ps->data.element1;
42    element_Ix.data.element2 = ps->data.element2;
43    result = (a * b);
44
45    if (((a + b) <  100 )) {
46      sem_wait(&sem_multiset);
47      pm = AddMultisetElement (&id_element, result);
48      pi = SearchInstanceIx(&instances_I2, element_Ix);
49      DeleteMultisetElementId (pi->id_data.id1);
50      DeleteMultisetElementId (pi->id_data.id2);
51      sem_post(&sem_multiset);
52      return TRUE;
53    }
54    else {
55      return FALSE;
56    }
57  }
```

Código 5.30: Experimento 5 - Extrato do *main.c* - funções executadas pelas *threads*.

Como podemos verificar no Código 5.30 a montagem das funções referentes às reações estão corretas, contendo condições de reações (linhas 16 e 45) dentre outros aspectos montados corretamente.

### 5.2.3 GSink - Análise do Potencial de Desempenho

A presente Seção aborda a segunda categoria de testes idealizados para o *GSink*. Pretendemos apresentar o potencial de obtenção de desempenho de nosso ambiente para execução de códigos Gamma a medida em que a granularidade da reação aumenta.

#### 5.2.3.1 Experimento 6

**Experimento 6 - Aspectos Gerais**

Para nosso sexto caso de teste, propomos uma aplicação que possui uma única reação que realiza a soma dos mil primeiros inteiros. Pela extensão do multiconjunto que deve constar na descrição do programa Gamma, o mesmo não será apresentado no código abaixo. Entretanto, os arquivos utilizados para este experimento, também serão disponibilizados no repositório de experimentos desta tese: https://gitlab.com/rui.rmj/dsc-experimentos. Assim, o código 5.31 apresenta um extrato da aplicação Gamma utilizada para este experimento:

```
1 sum  { /* Multiconjunto omitido */ }
2
3 where
4
5 sum = replace x,y by x+y if true
```

Código 5.31: Experimento 6 - Código Gamma.

Dessa forma, a reação *"sum"* do Código 5.31 seleciona elementos dois a dois e realiza a soma destes, sempre que for possível selecionar dois elementos. Ou seja, a computação termina necessariamente quando existir somente um elemento no multiconjunto.

O caso de teste proposto no Código 5.31 foi executado nas implementações *Gamma-Sequencial*, *Gamma-MPI* e *GSink*. Os resultados serão apresentados posteriormente, de maneira consolidada, após explanação do restante dos testes executados. As implementações de Gamma utilizadas possuem uma característica de não permitirem a utilização de reações que possuam granularidade grossa. Ou seja, a operação executada por cada reação corresponde a uma operação lógica ou aritmética, dentre outras operações simples, conforme podemos verificar em [18] e [30]. Conforme percebemos na proposta da *Trebuchet*, a eficiência de um programa paralelo feito com o *TALM* depende da escolha da granularidade ideal das superinstruções de maneira a expor paralelismo sem comprometer os custos de comunicação [61]. Da mesma forma, o *GSink* tem potencial para obtenção de desempenho, à medida em que a granularidade das operações possa ser alterada.

Por isso, realizamos alterações nos códigos executados nos três ambientes de execução de programas Gamma de forma a inserir um custo computacional maior, sem prejuízo à corretude das operações realizadas. Em outras palavras, inserimos manualmente cálculos matemáticos, mais notadamente multiplicações matriciais, no momento em que cada reação realiza a operação de soma explicitada na cláusula *BY* da reação *"sum'"*(Código 5.31). Entretanto, tal multiplicação matricial não foi convertida em alteração no multiconjunto, teve o objetivo somente de inserir um custo computacional maior na operação realizada por cada instância de reação, o que seria similar a um custo computacional mais complexo de uma reação com granularidade grossa. Abaixo iremos apresentar detalhes das alterações realizadas em cada ambiente.

***Gamma-Sequencial:*** O *Front-End* desta implementação gera o arquivo *"run.c"*, a ser executado em seu *Runtime*. O arquivo *"code.c"* que faz parte do *Front-End* teve a função *"void GenCodeExpression();"* alterada para gerar a multiplicação matricial citada. O extrato da referida função será reproduzido no Código 5.32:

```
void GenCodeExpression (pe, name,nl)
EXPRESSION *pe;
char *name; /* the name of the generated function */
NAMELIST *nl;
{
    register EXPRESSION *paux = pe;

    /*
    * Generate the Header of the Function.
    */
    fprintf (Output, "%s (", name);
    while (nl != (NAMELIST *) NULL)
    {
        fprintf (Output, "%s", nl->n_name);
        if ((nl = nl->n_next) != (NAMELIST *) NULL)
        fprintf (Output, ", ");
    }
    fprintf (Output, ")\n{\n");
    fprintf (Output, "register int Result = 0;\n\n");

    // Test Matrix Multiplication
    fprintf (Output, "// Test Matrix Multiplication\n");
    fprintf (Output, "int line,row, i, aux;\n");
    fprintf (Output, "int dim = 200;\n");
    fprintf (Output, "int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];\n");
    fprintf (Output, "for(line=0; line<dim; line++)\n");
    fprintf (Output, "\tfor(row=0; row<dim; row++){\n");
    fprintf (Output, "\t\taux=0;\n");
    fprintf (Output, "\t\tfor(i=0; i<dim; i++) {\n");
    fprintf (Output, "\t\t\taux = aux + (mat1[line][i]*mat2[i][row]);\n");
    fprintf (Output, "\t\t}\n");
    fprintf (Output, "\tmat3[line][row]=aux;\n");
    fprintf (Output, "}\n");
    fprintf (Output, "// End Test Matrix Multiplication\n");
    fprintf (Output, "\n");
```

```
36      // Test Matrix Multiplication
37
38      fprintf (Output, "Result = ");
39      CodePrintExpression (pe);
40      fprintf (Output, ";\nreturn (Result);\n");
41      fprintf (Output, "\n}\t/* end %s */\n\n", name);
42  } /* end GenCodeExpression */
```

Código 5.32: Experimento 6 - Gamma Sequencial - Extrato de code.c (*Front-End*).

A alteração inserida no referido Código 5.32 encontra-se entre os comentários: "*// Test Matrix Multiplication*". Como resultado da alteração acima citada, o arquivo "run.c", teve a seguinte inserção de multiplicação matricial, na função que executa a ação (operação) da referida reação "sum":

```
1  sum_action_0_e (y, x)
2  {
3      register int Result = 0;
4
5      // Test Matrix Multiplication
6      int line,row, i, aux;
7      int dim = 200;
8      int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
9      for(line=0; line<dim; line++)
10         for(row=0; row<dim; row++){
11             aux=0;
12             for(i=0; i<dim; i++) {
13                 aux = aux + (mat1[line][i]*mat2[i][row]);
14             }
15             mat3[line][row]=aux;
16         }
17     // End Test Matrix Multiplication
18
19     Result = (x  + y );
20     return (Result);
21 } /* end sum_action_0_e */
```

Código 5.33: Experimento 6 - Gamma Sequencial - Extrato de run.c.

Dessa forma, o Código 5.33 apresenta o extrato da função, no arquivo *"run.c"*, que executa a ação de soma. Entretanto, a mesma executa uma multiplicação antes da execução da reação propriamente dita, sem prejuízo ao cálculo da reação.

***Gamma-MPI:*** Procedimento similar foi realizado nesta implementação. Alteramos o *Front-End*, visando gerar a correta alteração no "run.c". Desta maneira, obtivemos a seguinte alteração na função "*cell_2_action_0_e*", responsável pela execução da ação da reação "sum":

```
cell_2_action_0_e (y, x)
{
    register int Result = 0;

    // Test Matrix Multiplication
    int line,row, i, aux;
    int dim = 100;
    int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
    for(line=0; line<dim; line++)
        for(row=0; row<dim; row++){
            aux=0;
            for(i=0; i<dim; i++) {
                aux = aux + (mat1[line][i]*mat2[i][row]);
            }
            mat3[line][row]=aux;
        }
    // End Test Matrix Multiplication
    Result = (x  + y );
    return (Result);
} /* end cell_2_action_0_e */
```

Código 5.34: Experimento 6 - Gamma MPI - Extrato de run.c.

Assim, da mesma maneira como procedemos na implementação *Gamma-Sequencial*, em *Gamma-MPI*, o código do *Front-End* encarregado por gerar as funções responsáveis pela execução da ação das reações foi alterado. Tal alteração, visou realizar a inserção de uma multiplicação matricial para aumentar o custo computacional da operação efetuada pela execução de cada instância da reação *"sum"*.

***GSink:*** Por fim, as alterações realizadas no *GSink*, visando o aumento do custo computacional no momento da execução de cada instância da reação *"sum"*, foram realizadas manualmente. Assim, a função *R1Function*, executada pelas *threads* foi alterada conforme o extrato de código abaixo:

```
/*************************************************************/
/*******  Function related to the R1 reaction execution ********/
/*************************************************************/
int R1Function(reg_sink *ps){
    // Name of the gamma reaction - according the gamma code: sum
    int x, y, result;
    reg_multiset *pm;
    reg_I element_Ix, *pi;

    // Test - Matrix multiplication
    int line,row, i, aux;
    int dim = 100;
    int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
    for(line=0; line<dim; line++)
        for(row=0; row<dim; row++){
            aux=0;
            for(i=0; i<dim; i++) {
                aux = aux + (mat1[line][i]*mat2[i][row]);
```

```
19              }
20              mat3[line][row]=aux;
21          }
22      // End Test Matrix Multiplication
23
24      x  = ps->data.element1;
25      y  = ps->data.element2;
26      element_Ix.data.element1 = ps->data.element1;
27      element_Ix.data.element2 = ps->data.element2;
28      result = (x + y);
29
30      if (1) {
31          sem_wait(&sem_multiset);
32          pm = AddMultisetElement (&id_element, result);
33          pi = SearchInstanceIx(&instances_I1, element_Ix);
34          DeleteMultisetElementId (pi->id_data.id1);
35          DeleteMultisetElementId (pi->id_data.id2);
36          sem_post(&sem_multiset);
37          return TRUE;
38      }
39      else {
40          return FALSE;
41      }
42  }
```

Código 5.35: Experimento 6 - GSink - Extrato de main.c.

Conforme comentários inseridos no extrato da função acima (Código 5.35), do código "main.c" (*Runtime* do *GSink*), o aumento do custo computacional foi realizado da mesma maneira que nas duas implementações de Gamma anteriores: através da inserção de uma multiplicação matricial. Tal operação, utilizou matrizes quadradas, onde as dimensões utilizadas estão descritas na Tabela 5.2 abaixo. A seção seguinte discutirá os resultados obtidos por estas execuções.

Tabela 5.2: GSink - Configuração dos Casos de Teste (Experimento 6).

| **Casos de Teste** | |
|---|---|
| **Identificação** | **Observação** |
| sum | soma dos 1000 primeiros inteiros |
| sum 100 x 100 | soma com multiplicação matricial (100 x 100) |
| sum 200 x 200 | soma com multiplicação matricial (200 x 200) |
| sum 300 x 300 | soma com multiplicação matricial (300 x 300) |

**Experimento 6 - Resultados**

A presente seção destina-se a apresentar os resultados obtidos com as execuções de cada caso de teste constante na Tabela 5.2, cujos detalhes foram descritos na seção anterior, sobre as implementações *Gamma-Sequencial*, *Gamma-MPI* e *GSink*. Vale ressaltar que a respostas das execuções nos três *Runtime* foram idênticas e iguais a 500500. A Tabela 5.3 apresenta tais resultados, seguida dos gráficos específicos:

Tabela 5.3: Experimento 6 - Tempos de Execução (em segundos).

| | | Implementações de Gamma | | | | | |
|---|---|---|---|---|---|---|---|
| | | Gamma-Sequencial | | Gamma-MPI | | GSink | |
| | | Média | Desvio Padrão | Média | Desvio Padrão | Média | Desvio Padrão |
| Caso de Teste | sum | 0,01 | 0,04 | 22,41 | 0,91 | 0,09 | 0,005 |
| | sum 100x100 | 6,06 | 0,007 | 43,23 | 0,42 | 3,66 | 0,03 |
| | sum 200x200 | 47,87 | 0,17 | 124,56 | 1,25 | 25,81 | 0,09 |
| | sum 300x300 | 163,05 | 0,93 | 313,58 | 0,94 | 86,61 | 0,48 |

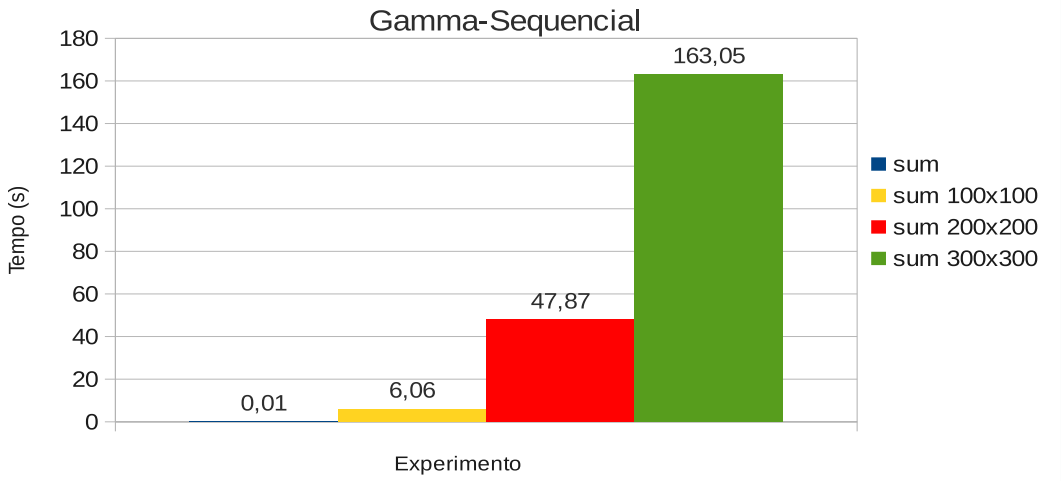


Figura 5.5: Experimento 6 - Tempos de Execução (s) - casos de teste *sum*, *sum 100x100*, *sum 200x200* e *sum 300x300* sobre a implementação *Gamma-Sequencial*.

As Figuras 5.5, 5.6 e 5.7 apresentam os tempos de execução dos casos de teste especificados na Tabela 5.2, para os ambientes de execução *Gamma-sequencial*, *Gamma-MPI* e *GSink*, respectivamente. A Tabela 5.3, apresenta a consolidação destes tempos de execução e o desvio padrão em cada caso específico.

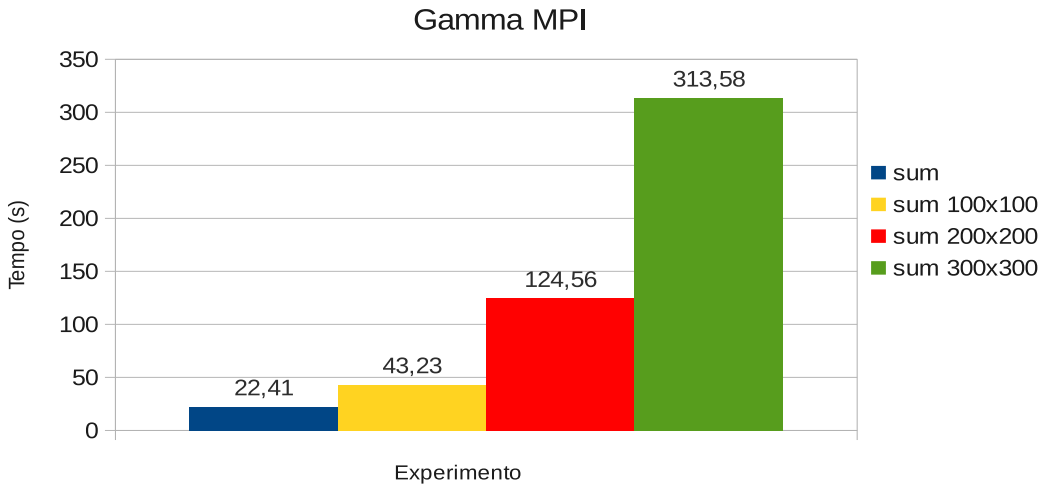


Figura 5.6: Experimento 6 - Tempos de Execução (s) - casos de teste *sum*, *sum 100x100*, *sum 200x200* e *sum 300x300* sobre a implementação *Gamma-MPI*.

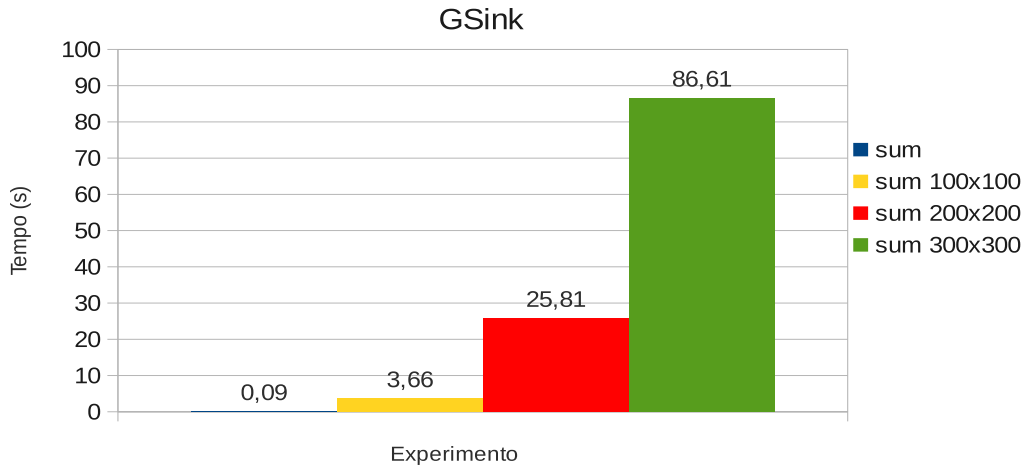


Figura 5.7: Experimento 6 - Tempos de Execução (s) - casos de teste *sum*, *sum 100x100*, *sum 200x200* e *sum 300x300* sobre a implementação *GSink*.

O *GSink* apresentou tempos de execução menores que as outras duas implementações, à medida em que o custo computacional para execução de cada reação aumentou. Note que para a execução do caso de teste cuja operação não utiliza a inclusão da multiplicação matricial (caso de teste identificado por *"sum"*) o *GSink* apresentou média de tempo de execução ligeiramente superior ao ambiente *Gamma-Sequencial*. Isso deve-se ao fato de que no *GSink*, para o caso de reações com granularidade fina, o custo computacional para gerenciamento do grafo de instâncias, seu mecanismo de reversão de arestas de *sinks* e execução por *threads* acaba sendo maior que um mecanismo mais simples de escalonamento, conforme *Gamma-Sequencial*.

A implementação *Gamma-Sequencial* simplesmente disponibiliza o multiconjunto à execução pela reação, que realizará tentativas para executar sequencialmente cada instância desta reação.

Entretanto, com o aumento do custo computacional envolvido na execução de cada instância de reação, o *GSink* apresenta melhores resultados de tempo de execução, comparado as outras duas implementações. Isso deve-se ao fato do *GSink* efetivamente proporcionar a execução das operações realizadas por cada *Sink* de uma mesma orientação acíclica de maneira paralela. Em outras palavras, uma vez identificados os *sinks* de uma orientação acíclica, o cálculo computacional que estes necessitam realizar (que correspondem a instâncias da reação associada a este *Sink*) ocorre de maneira paralela.

Por outro lado verificamos que a implementação *Gamma-MPI* apresentou os piores tempos de execução dentre as três implementações. Neste caso, o *overhead* de comunicação inserido pela utilização do protocolo MPI aliado a um gerenciamento centralizado e sequencial trouxe piores resultados de desempenho a esta implementação. Em *Gamma-MPI* o gerenciamento da execução ocorre de maneira centralizada. Os nós da rede (que executam somente uma reação cada) solicitam a utilização do multiconjunto à um gerenciador central. Este envia o multiconjunto, caso não esteja sendo utilizado por outro nó da rede. Quando tal nó que solicitou o multiconjunto recebe o mesmo, o mecanismo de execução utilizado ocorre de maneira sequencial, similar ao realizado pela implementação *Gamma-Sequencial*. Dessa forma, além da execução sequencial realizada, todo gerenciamento centralizado e utilização do protocolo MPI trazem grandes *overheads* a esta implementação de Gamma. Tal fato foi observado em [18].

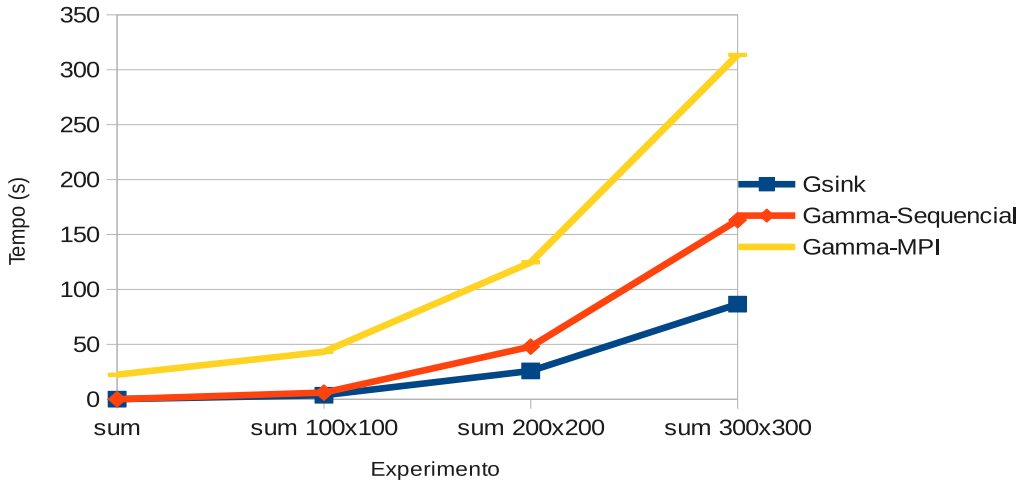


Figura 5.8: Experimento 6 - Tempos de Execução 1 (s) - três implementações de Gamma por cada caso de teste.

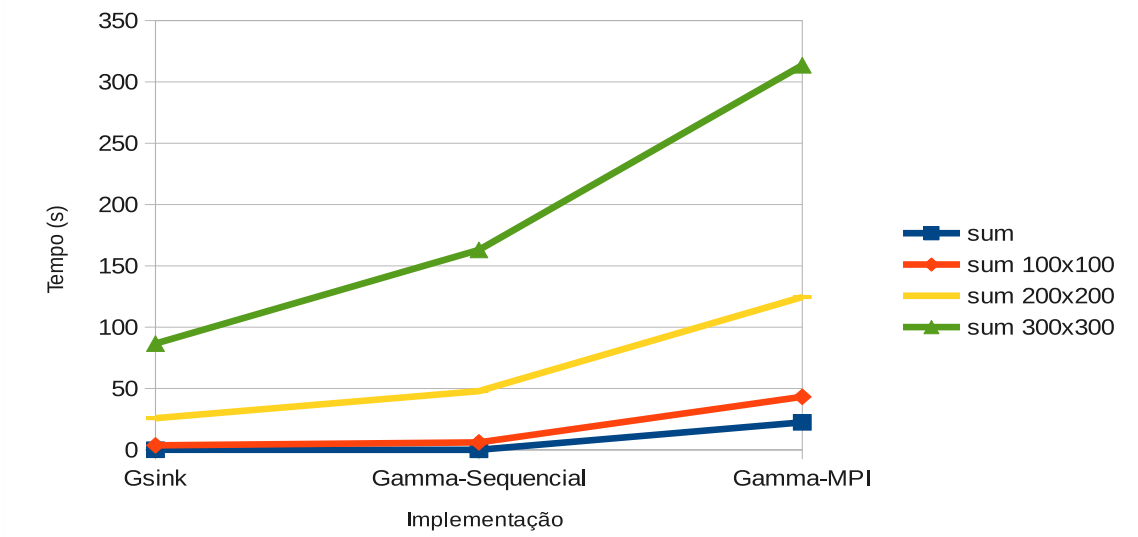


Figura 5.9: Experimento 6 - Tempos de Execução 2 (s) - três implementações de Gamma por cada caso de teste.

As Figuras 5.8 e 5.9 apresentam gráficos consolidados das execuções realizadas.

A Figura 5.10 apresenta os *speedups* obtidos pelo *GSink*, comparados à execução no ambiente *Gamma-Sequencial*. Conforme vimos anteriormente, o tempo de execução para o primeiro caso de teste (que não utiliza o artifício de multiplicação matricial para aumento do custo computacional) obtido pelo *GSink* foi maior que o tempo obtido pela utilização de *Gamma-Sequencial*, motivo este do *slowdown* obtido na Figura 5.10. Entretanto, na medida em que o custo computacional aumenta, percebemos *speedups* de 1,656, 1,855 e 1,883.

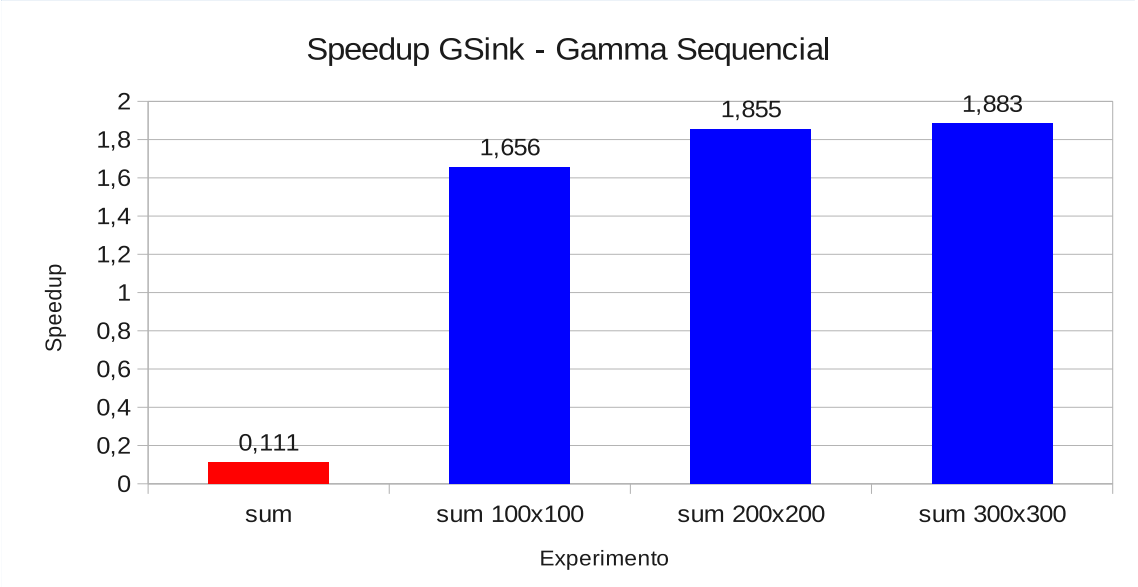


Figura 5.10: Experimento 6 - Speedup do *GSink* em relação ao *Gamma-Sequencial* para os casos de teste *sum*, *sum 100x100*, *sum 200x200* e *sum 300x300*.

Estes resultados corroboram a afirmação de que o *GSink* beneficia-se de custos computacionais maiores associados à execução de cada instância de reação. Tal fato mostra o quão promissor é investir em pesquisas relacionadas a dotar o paradigma Gamma de reações que possam vir a executar funções, e não somente operações aritméticas e lógicas simples.

Já a Figura 5.11 apresenta os *speedups* obtidos pelo *GSink* comparados à execução da implementação *Gamma-MPI*. Em todos os casos de teste foram apresentados *speedup*. Entretanto, note que para o primeiro caso de teste, a implementação Gamma-MPI apresenta tempos de execução extremamente altos, comparados ao *GSink*, motivo pelo qual o primeiro *speedup* atingiu valor de 249. Isso deve-se ao fato de que para execução de operações com granularidade fina, *Gamma-MPI* prejudica-se muito pelo excesso de troca de mensagens MPI aliado ao mecanismo sequencial de execução. Em certa medida, *Gamma-MPI* também beneficia-se do aumento de granularidade das operações. Entretanto, mesmo com esse aparente benefício, o ambiente *GSink* ainda fornece speedups consideráveis comparados à *Gamma-MPI*.

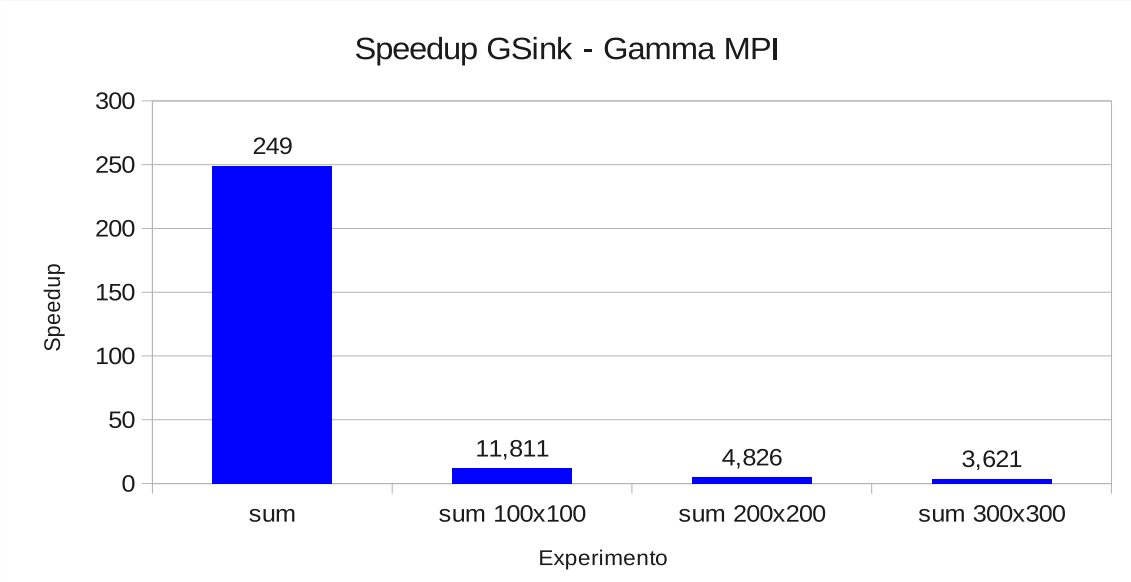


Figura 5.11: Experimento 6 - Speedup do *GSink* em relação ao *Gamma-MPI* para os casos de teste *sum*, *sum 100x100*, *sum 200x200* e *sum 300x300*.

#### 5.2.3.2 Experimento 7

**Experimento 7 - Aspectos Gerais**

O experimento 7 tem o mesmo objetivo do experimento anterior, realizar alterações nos códigos visando executar um custo computacional maior no que diz respeito à operação realizada por cada reação. Entretanto, para o presente experimento, utilizamos um código Gamma composto por três reações. Além disso, as condições de reação não são sempre verdadeiras, ao contrário do experimento 6. Com isso, utilizamos o mesmo Código Gamma utilizado no experimento 4 do *GSink* (código 5.27). A única diferença está na composição dos elementos do multiconjunto, uma vez que decidimos manter a mesma quantidade de elementos do experimento 6. Assim, para este sétimo experimento, o multiconjunto é formado por todos os elementos que compreendem os seguintes intervalos de inteiros:
$\{x \in \mathbb{Z} \mid 1 \leq x \leq 200\}$
$\{x \in \mathbb{Z} \mid 21000 \leq x \leq 21600\}$
$\{x \in \mathbb{Z} \mid 23000 \leq x \leq 23200\}$

Abaixo reproduzimos o código Gamma utilizado neste sétimo experimento (Código 5.36), composto por três reações e de um multiconjunto de 1000 elementos (omitido por questões de espaço). Cada reação possui uma condição de reação específica, fazendo com que nem sempre os elementos selecionados consigam reagir.

```
1 first | second | third { /* Multiconjunto omitido */ }
2
3 where
4
5 first = replace x,y by x+y if (x <= 20100) and (y <= 20100)
6
7 second = replace m,n by m if (m > n) and (m >= 21000) and (m <=
    22000) and (n >= 21000) and (n <= 22000)
8
9 third = replace a,b by (a+b)-10000 if (a >= 23000) and (b >=
    23000)
```

Código 5.36: Experimento 7 - Código Gamma (também utilizado na Section 5.2.2.4).

O custo computacional inserido foi idêntico ao experimento anterior: uma multiplicação de matrizes quadradas, onde as dimensões variaram entre 100x100, 200x200 e 300x300. Assim, da mesma forma que o experimento anterior, realizamos alterações no *Front-End* das implementações *Gamma-Sequencial* e *Gamma-MPI*, visando gerar um arquivo *"run.c"* contendo tais multiplicações matriciais. Assim, o arquivo *"code.c"* foi alterado nestas duas implementações, da mesma forma que o experimento anterior, gerando arquivos *"run.c"* (que serão executados nos respectivos *Runtimes*) compatíveis com tais alterações. Como os *Front-End* foram alterados da

mesma forma, apresentaremos somente um extrato das funções executadas por cada reação, inicialmente na implementação *Gamma-Sequencial*:

```
first_action_0_e (y, x)
{
    register int Result = 0;

    // Test Matrix Multiplication
    int line,row, i, aux;
    int dim = 100;
    int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
    for(line=0; line<dim; line++)
        for(row=0; row<dim; row++){
            aux=0;
            for(i=0; i<dim; i++) {
                aux = aux + (mat1[line][i]*mat2[i][row]);
            }
            mat3[line][row]=aux;
        }
    // End Test Matrix Multiplication

    Result = (x  + y );
    return (Result);
} /* end first_action_0_e */

second_action_0_e (m)
{
    register int Result = 0;

    // Test Matrix Multiplication
    int line,row, i, aux;
    int dim = 100;
    int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
    for(line=0; line<dim; line++)
        for(row=0; row<dim; row++){
            aux=0;
            for(i=0; i<dim; i++) {
                aux = aux + (mat1[line][i]*mat2[i][row]);
            }
            mat3[line][row]=aux;
        }
    // End Test Matrix Multiplication

    Result = m ;
    return (Result);
} /* end second_action_0_e */

third_action_0_e (b, a)
{
    register int Result = 0;

    // Test Matrix Multiplication
    int line,row, i, aux;
    int dim = 100;
    int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
    for(line=0; line<dim; line++)
        for(row=0; row<dim; row++){
            aux=0;
            for(i=0; i<dim; i++) {
```

```
57                  aux = aux + (mat1[line][i]*mat2[i][row]);
58              }
59              mat3[line][row]=aux;
60          }
61      // End Test Matrix Multiplication
62
63      Result = ((a  + b ) -  10000 );
64      return (Result);
65  } /* end third_action_0_e */
```

Código 5.37: Experimento 7 - Gamma Sequencial - Extrato de run.c.

Assim, as três funções que constam no arquivo *"run.c"* da versão Gamma-Sequencial que realizam as ações das reações tiveram o custo de uma multiplicação matricial inseridas em sua operação. Da mesma maneira, o arquivo *"run.c"* da versão *Gamma-MPI* teve suas funções responsáveis pela execução das ações das reações alteradas, conforme extrato do referido *"run.c"* reproduzido no Código 5.38, abaixo:

```
1  cell_6_action_0_e (b, a)
2  {
3      register int Result = 0;
4
5      // Test Matrix Multiplication
6      int line,row, i, aux;
7      int dim = 100;
8      int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
9      for(line=0; line<dim; line++)
10         for(row=0; row<dim; row++){
11             aux=0;
12             for(i=0; i<dim; i++) {
13                 aux = aux + (mat1[line][i]*mat2[i][row]);
14             }
15             mat3[line][row]=aux;
16         }
17     // End Test Matrix Multiplication
18
19     Result = ((a  + b ) -  10000 );
20     return (Result);
21 } /* end cell_6_action_0_e */
22
23 cell_5_action_0_e (m)
24 {
25     register int Result = 0;
26
27     // Test Matrix Multiplication
28     int line,row, i, aux;
29     int dim = 100;
30     int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
31     for(line=0; line<dim; line++)
32         for(row=0; row<dim; row++){
33             aux=0;
34             for(i=0; i<dim; i++) {
35                 aux = aux + (mat1[line][i]*mat2[i][row]);
36             }
37             mat3[line][row]=aux;
38         }
39     // End Test Matrix Multiplication
40
```

```
    Result = m ;
    return (Result);
} /* end cell_5_action_0_e */

cell_4_action_0_e (y, x)
{
    register int Result = 0;

    // Test Matrix Multiplication
    int line,row, i, aux;
    int dim = 100;
    int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
    for(line=0; line<dim; line++)
        for(row=0; row<dim; row++){
            aux=0;
            for(i=0; i<dim; i++) {
                aux = aux + (mat1[line][i]*mat2[i][row]);
            }
            mat3[line][row]=aux;
        }
    // End Test Matrix Multiplication

    Result = (x  + y );
    return (Result);
} /* end cell_4_action_0_e */
```

Código 5.38: Experimento 7 - Gamma MPI - Extrato de run.c.

Já para o *GSink* a operação de multiplicação matricial foi realizada manualmente, no âmbito das funções executadas pelas *threads* disparadas pelos *Sinks*. Um extrato das referidas funções, constantes do *"main.c"* será reproduzido no Código 5.39, abaixo:

```
/*************************************************************/
/*******  Function related to the R1 reaction execution ********/
/*************************************************************/
int R1Function(reg_sink *ps){
    // Name of the gamma reaction - according the gamma code: first
    int x, y, result;
    reg_multiset *pm;
    reg_I element_Ix, *pi;

    x  = ps->data.element1;
    y  = ps->data.element2;

    if (((x <=  20100 ) && (y <=  20100 ))) {
        element_Ix.data.element1 = ps->data.element1;
        element_Ix.data.element2 = ps->data.element2;
        result = (x + y);

        // Test - Matrix multiplication
        int line,row, i, aux;
        int dim = 100;
        int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
        for(line=0; line<dim; line++)
            for(row=0; row<dim; row++){
                aux=0;
                for(i=0; i<dim; i++) {
```

```
                    aux = aux + (mat1[line][i]*mat2[i][row]);
                }
                mat3[line][row]=aux;
            }
    // End Test Matrix Multiplication


        sem_wait(&sem_multiset);
        pm = AddMultisetElement (&id_element, result);
        pi = SearchInstanceIx(&instances_I1, element_Ix);
        DeleteMultisetElementId (pi->id_data.id1);
        DeleteMultisetElementId (pi->id_data.id2);
        sem_post(&sem_multiset);
        return TRUE;
    }
    else {
        return FALSE;
    }
}

/*****************************************************************/
/*******  Function related to the R2 reaction execution ********/
/*****************************************************************/
int R2Function(reg_sink *ps){
    // Name of the gamma reaction - according the gamma code: second
    int m, n, result;
    reg_multiset *pm;
    reg_I element_Ix, *pi;

    m  = ps->data.element1;
    n  = ps->data.element2;

    if ((((((m > n) && (m >=  21000 )) && (m <=  22000 )) && (n >=  21000 )) && (
    n <=  22000 ))) {
        element_Ix.data.element1 = ps->data.element1;
        element_Ix.data.element2 = ps->data.element2;
        result = m;

        // Test - Matrix multiplication
        int line,row, i, aux;
        int dim = 100;
        int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
        for(line=0; line<dim; line++)
            for(row=0; row<dim; row++){
                aux=0;
                for(i=0; i<dim; i++) {
                    aux = aux + (mat1[line][i]*mat2[i][row]);
                }
                mat3[line][row]=aux;
            }
        // End Test Matrix Multiplication

    sem_wait(&sem_multiset);
        pm = AddMultisetElement (&id_element, result);
        pi = SearchInstanceIx(&instances_I2, element_Ix);
        DeleteMultisetElementId (pi->id_data.id1);
        DeleteMultisetElementId (pi->id_data.id2);
        sem_post(&sem_multiset);
        return TRUE;
```

```
84      }
85      else {
86          return FALSE;
87      }
88  }
89
90  /*************************************************************/
91  /*******  Function related to the R3 reaction execution ********/
92  /*************************************************************/
93  int R3Function(reg_sink *ps){
94      // Name of the gamma reaction - according the gamma code: third
95      int a, b, result;
96      reg_multiset *pm;
97      reg_I element_Ix, *pi;
98
99      a  = ps->data.element1;
100     b  = ps->data.element2;
101
102     if (((a >=  23000 ) && (b >=  23000 ))) {
103         element_Ix.data.element1 = ps->data.element1;
104         element_Ix.data.element2 = ps->data.element2;
105         result = ((a + b) -  10000 );
106
107         // Test - Matrix multiplication
108         int line,row, i, aux;
109         int dim = 100;
110         int mat1[dim][dim], mat2[dim][dim], mat3[dim][dim];
111         for(line=0; line<dim; line++)
112             for(row=0; row<dim; row++){
113                 aux=0;
114                 for(i=0; i<dim; i++) {
115                     aux = aux + (mat1[line][i]*mat2[i][row]);
116                 }
117                 mat3[line][row]=aux;
118             }
119         // End Test Matrix Multiplication
120
121         sem_wait(&sem_multiset);
122         pm = AddMultisetElement (&id_element, result);
123         pi = SearchInstanceIx(&instances_I3, element_Ix);
124         DeleteMultisetElementId (pi->id_data.id1);
125         DeleteMultisetElementId (pi->id_data.id2);
126         sem_post(&sem_multiset);
127         return TRUE;
128     }
129     else {
130         return FALSE;
131     }
132 }
```

Código 5.39: Experimento 7 - GSink - Extrato de main.c.

Assim, para este experimento, também foram alterados os custos computacionais relacionados à execução das reações nas três implementações de Gamma utilizadas, mediante inserção de multiplicações matriciais. A configuração dos casos de teste utilizados neste sétimo experimento para o *GSink*, está descrita na Tabela 5.4 abaixo. A seção seguinte discutirá os resultados obtidos por estas execuções.

Tabela 5.4: GSink - Configuração dos Casos de Teste (Experimento 7).

| **Casos de Teste** | |
|---|---|
| **Identificação** | **Observação** |
| hip | Código Gamma composto de 3 reações hipotéticas |
| hip 100 x 100 | 3 reações hipotéticas com multiplicação matricial (100 x 100) |
| hip 200 x 200 | 3 reações hipotéticas com multiplicação matricial (200 x 200) |
| hip 300 x 300 | 3 reações hipotéticas com multiplicação matricial (300 x 300) |

### Experimento 7 - Resultados

O objetivo em oferecer este sétimo experimento foi utilizar um programa Gamma composto por mais de uma reação, onde as mesmas possuíssem condições de reação que não fossem sempre verdadeiras. No experimento 6, a condição de reação era sempre verdadeira, enquanto pudessem ser selecionados elementos a reagir. Para este sétimo experimento, cada uma das três reações possuem condições de reação específicas e diferentes.

Dessa forma, a versão *Gamma-Sequencial* realiza a execução das instâncias de cada reação de maneira sequencial. A sensação de paralelismo é dada mediante a eventual alternância entre a execução das reações. Tal alternância acontece sempre quando ocorre uma tentativa bem sucedida de execução de uma reação, onde, após tal execução, um mecanismo de aleatoriedade é responsável por decidir qual reação irá tentar reagir novamente. Dessa forma, esta implementação irá testar diversas combinações de elementos até que consiga reagir, momento este em que o mecanismo de aleatoriedade entra em ação, tentando alternar a execução de reações.

Já a implementação *Gamma-MPI* possui um mecanismo de execução onde um Elemento de Processamento (EP) central envia o multiconjunto (utilizando protocolo MPI) para um EP que queira executar uma reação. Inicialmente os EP são mapeados de forma que uma reação será executada especificamente por um EP escolhido em tempo de compilação. Dessa forma, uma vez que um EP esteja com o direito de manipular o multiconjunto, os demais EPs aguardam a futura disponibilização deste. Assim, um EP permanece com o direito de execução do multiconjunto enquanto estiver tentando reagir, ou seja, enquanto os testes de condição de reação estiverem sendo falsos. Caso o teste condicional da reação seja verdadeiro, isso significa que os elementos escolhidos podem reagir. Neste momento, o EP executa a ação relacionada à sua reação e devolve o multiconjunto para o EP centralizador, que irá disponibilizar a outro EP, responsável pela execução de outra reação.

No *GSink*, o mecanismo de execução utiliza um grafo composto por instâncias de todas as reações, onde cada *sink* irá disparar uma *thread* específica. Assim, em uma mesma orientação acíclica, *sinks* distintos podem estar relacionados a instâncias de reações distintas. Vale lembrar que, conforme mencionamos no Capítulo 4, quando um *sink* consegue reagir, antes de reverter suas arestas, este necessita excluir os vértices que tenham sido afetados com esta alteração (além de realizar modificações no multiconjunto). Já no caso de fracasso na tentativa de reagir, as alterações não são realizadas no multiconjunto, entretanto os elementos que fizeram parte da instância deste *sink* permanecem no multiconjunto para serem utilizados para uma nova escolha de instâncias.

Note que os mecanismos citados acima, que dizem respeito à alternância de execução de reações, só aparecem em códigos compostos por mais de uma reação, fato este que motivou a disponibilização deste experimento.

Agora apresentaremos os resultados obtidos nas execuções envolvidas neste sétimo experimento, onde a configuração de tais casos de teste foram apresentadas na Tabela 5.4. A Tabela 5.5 apresenta tais resultados, a seguir:

Tabela 5.5: Experimento 7 - Tempos de Execução (em segundos).

| | | Implementações de Gamma | | | | | |
|---|---|---|---|---|---|---|---|
| | | Gamma-Sequencial | | Gamma-MPI | | GSink | |
| | | Média | Desvio Padrão | Média | Desvio Padrão | Média | Desvio Padrão |
| Caso de Teste | hip | 0,04 | 0,007 | 123,25 | 2,48 | 0,34 | 0,01 |
| | hip 100x100 | 6,33 | 0,05 | 180,15 | 1,29 | 3,84 | 0,06 |
| | sum 200x200 | 47,96 | 0,03 | 363,06 | 3,72 | 26,38 | 0,20 |
| | sum 300x300 | 162,57 | 1,64 | 824,57 | 3,30 | 87,03 | 1,21 |

As Figuras 5.12, 5.13 e 5.14 apresentam os tempos de execução dos casos de teste afetos ao experimento 7, mediante os ambientes de execução *Gamma-Sequencial*, *Gamma-MPI* e *GSink*. A configuração de tais casos de teste podem ser verificadas na Tabela 5.4, assim como os tempos de execução, que constam da Tabela 5.5. Com relação aos resultados obtidos em cada um dos *Runtimes* utilizados, os mesmos foram idênticos e correspondem ao multiconjunto $\{20100, 21600, 2643100\}$.

Para o experimento 7, vimos que os resultados apresentam comportamentos similares aos encontrados no experimento anterior, para as implementações *Gamma-Sequencial* e *GSink*. Ou seja, para estas duas implementações, os tempos de execução encontrados para o Experimento 6 e 7 foram bastante próximos. Já os tempos de

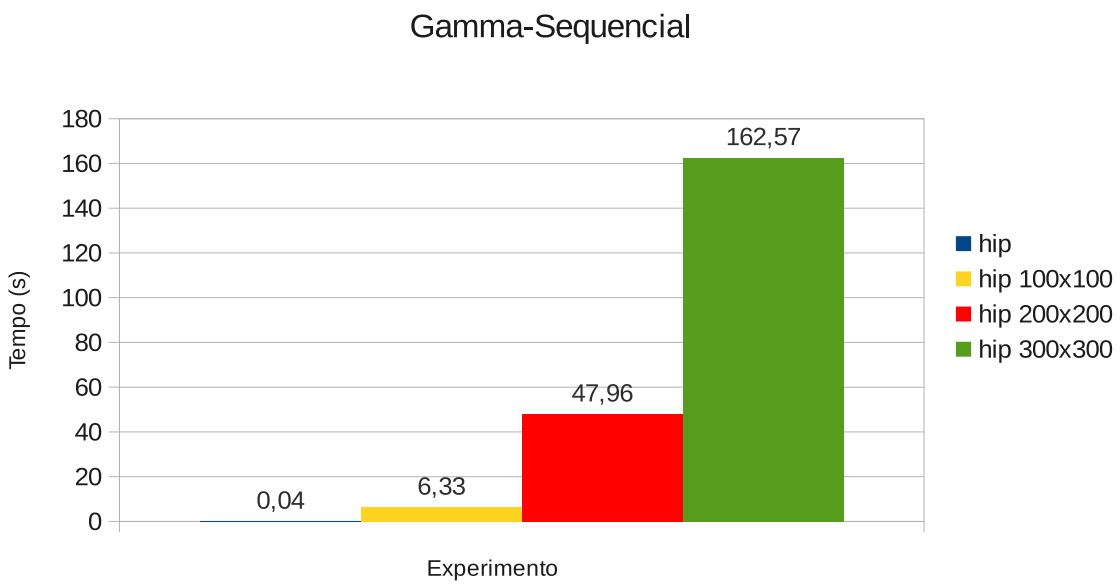


Figura 5.12: Experimento 7 - Tempos de Execução (s) - casos de teste *hip*, *hip 100x100*, *hip 200x200* e *hip 300x300* sobre a implementação *Gamma-Sequencial*.

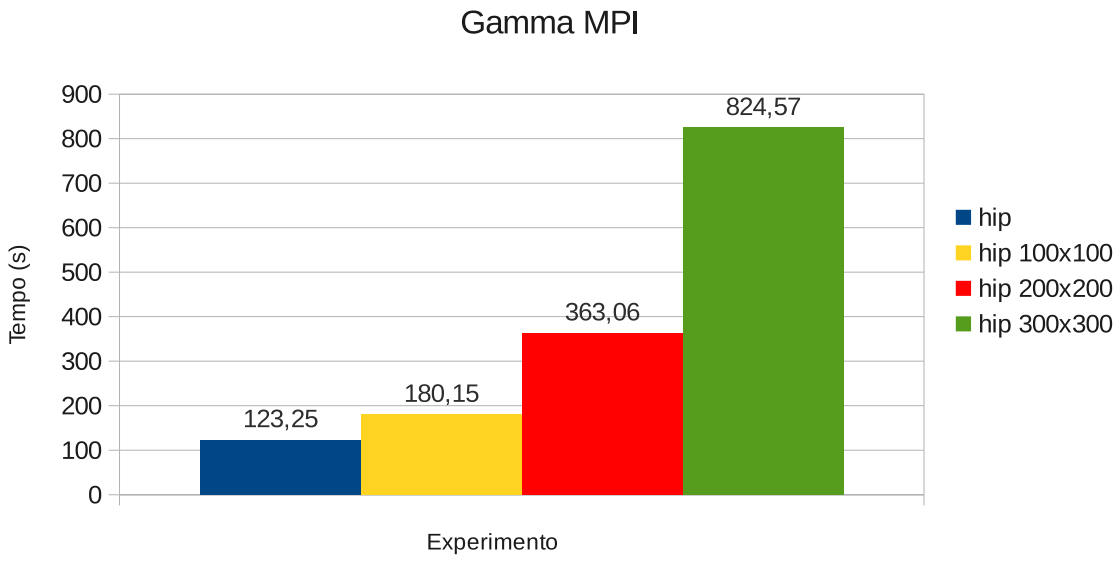


Figura 5.13: Experimento 7 - Tempos de Execução (s) - casos de teste *hip*, *hip 100x100*, *hip 200x200* e *hip 300x300* sobre a implementação *Gamma-MPI*.

execução apresentados no experimento 7 da versão *Gamma-MPI* foram consideravelmente maiores, comparados aos tempos encontrados para o experimento 6.

Isso ocorre com a versão *Gamma-MPI* devido ao alto custo computacional envolvido na troca de mensagens MPI. A medida em que a quantidade de reações que compõem o programa Gamma aumenta, a configuração das células processadoras se modifica. Assim, uma aplicação contendo somente uma reação (Experimento

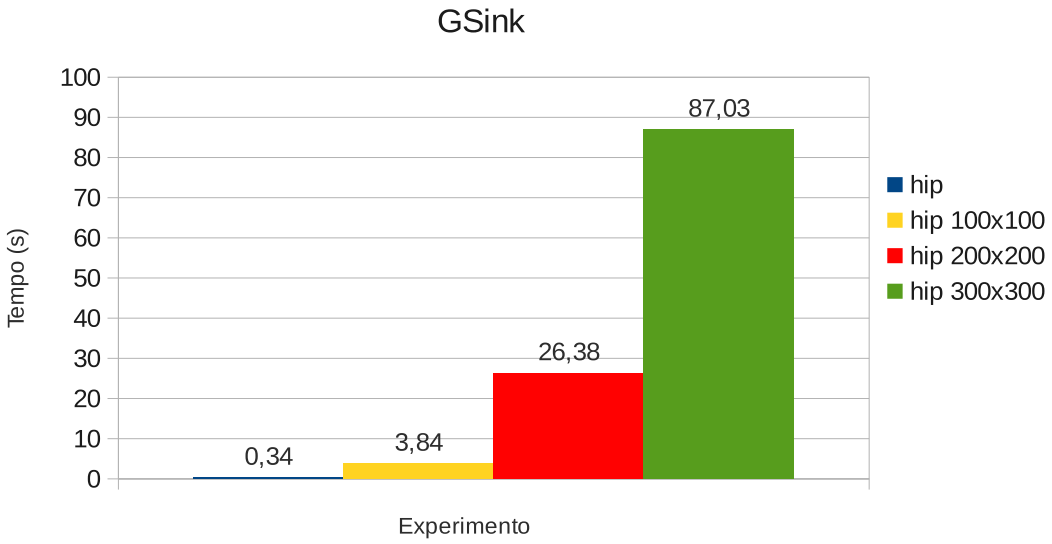


Figura 5.14: Experimento 7 - Tempos de Execução (s) - casos de teste *hip*, *hip 100x100*, *hip 200x200* e *hip 300x300* sobre a implementação *GSink*.

6) utiliza três células processadoras, ao passo em que uma aplicação composta por três reações em paralelo (experimento 7) possui 7 células entre *Workers*, *Main Controller*, *Controllers* e *Bag-Cell* [30]. Toda a troca de mensagens entre as células ocorre através do protocolo MPI, justificando o aumento do custo de comunicação e respectivamente do tempo de execução final, comparado ao experimento 6.

A inserção de reações que possuem condições de reação que não estão sempre satisfeitas não apresentou um aumento considerável no tempo de execução para as implementações *Gamma-Sequencial* e *GSink*. Da mesma forma, o aumento na quantidade de reações, para estas duas reações, não foi tão prejudicial em termos de tempo de execução. Tal fato pode ser corroborado através da comparação dos resultados entre os experimentos 6 e 7.

Exatamente como ocorreu no Experimento 6, o *GSink* se beneficiou do aumento do custo computacional envolvido na operação realizada em cada reação, obtendo tempos de execução menores que as duas outras implementações. A mesma ressalva realizada para o primeiro caso de teste (execução sem aumento do custo computacional) do Experimento 6 é válida aqui: para este caso de teste, o *GSink* também apresentou tempo maior de execução que a abordagem Sequencial, tendo em vista o custo adicional em gerenciar um grafo de instâncias onde a execução ocorre mediante o disparo de *threads*.

As Figuras 5.15 e 5.16 apresentam os resultados de tempos de execução consolidados.

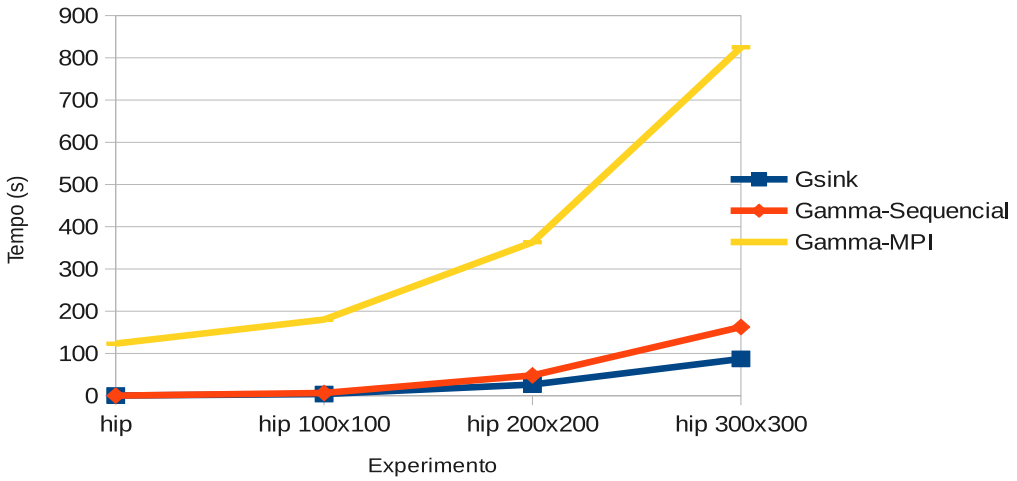


Figura 5.15: Experimento 7 - Tempos de Execução 1 (s) - três implementações de Gamma por cada caso de teste.

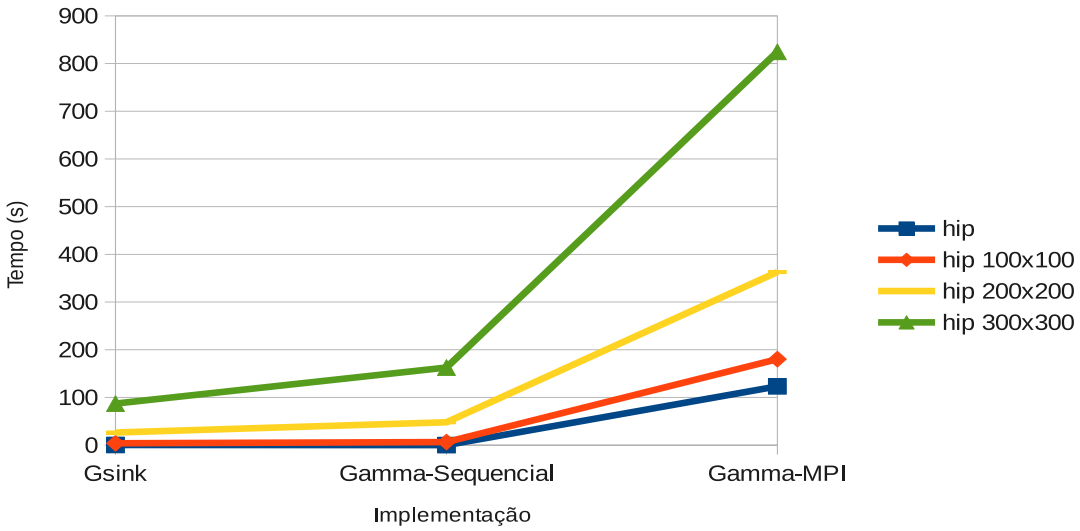


Figura 5.16: Experimento 7 - Tempos de Execução 2 (s) - três implementações de Gamma por cada caso de teste.

As Figuras 5.17 e 5.18 apresentam os *speedups* obtidos através das comparações com as versões *Gamma-Sequencial* e *Gamma-MPI*, respectivamente. Em comparação à implementação *Gamma-Sequencial* (Figura 5.17) obtivemos *speedups* semelhantes aos obtidos no sexto experimento, com um *slowdown* relativo ao primeiro caso de teste. Exatamente como no sexto experimento, o motivo do *slowdown* é o custo computacional para o gerenciamento do nosso mecanismo de escalonamento que, para reações com granularidade fina, acaba sendo mais custoso que um mecanismo mais simples e sequencial de escalonamento.

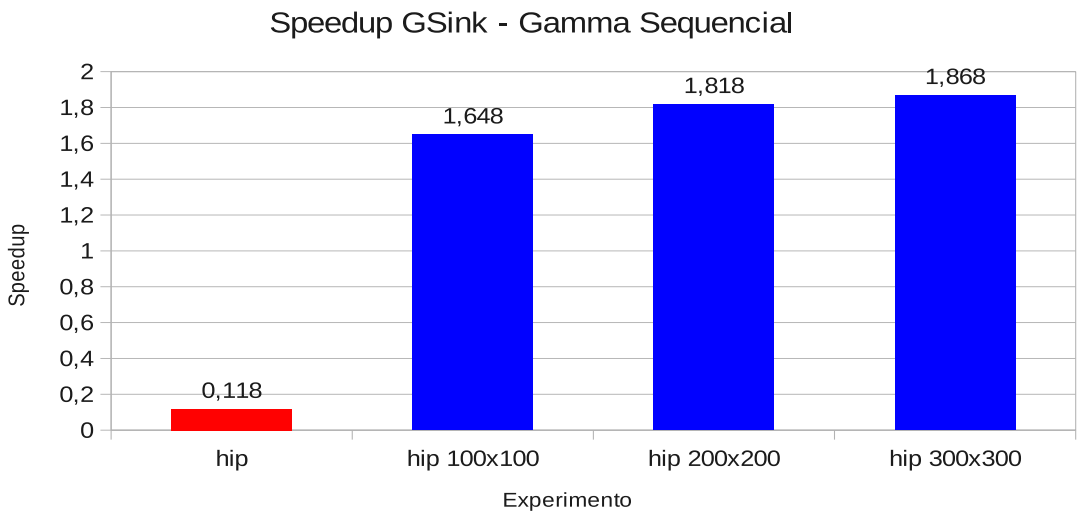


Figura 5.17: Experimento 7 - Speedup do *GSink* em relação ao *Gamma-Sequencial* para os casos de teste *hip*, *hip 100x100*, *hip 200x200* e *hip 300x300*.

Na Figura 5.18 o primeiro *speedup* demonstra o quão custoso é utilizar a versão MPI com granularidade fina da reação, o que faz sentido, ao passo que para compensar o custo envolvido na troca de mensagens desta implementação, se faz necessário o aumento do custo computacional da operação realizada pela reação. Não se demonstrou razoável a utilização de troca de mensagens na rede visando a execução de uma única operação lógica ou aritmética. Entretanto, mesmo com o aumento deste custo computacional, *GSink* apresentou melhor desempenho que *Gamma-MPI*, reforçando o potencial de nosso mecanismo de escalonamento em uma implementação de Gamma que permita granularidades mais grossas de reações.

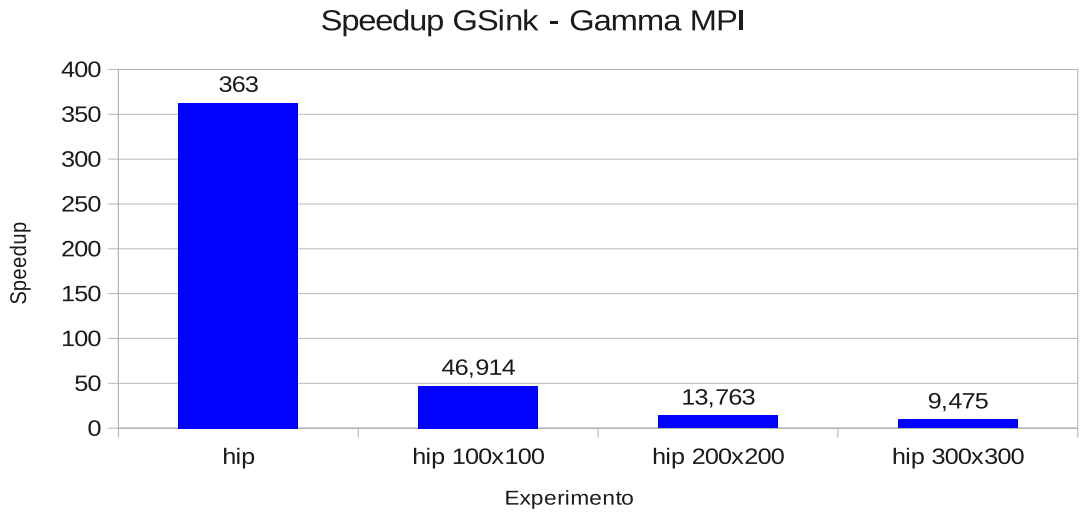


Figura 5.18: Experimento 7 - Speedup do *GSink* em relação ao *Gamma-MPI* para os casos de teste *hip*, *hip 100x100*, *hip 200x200* e *hip 300x300*.

#### 5.2.3.3 Análise de Concorrência

A presente seção apresenta algumas informações sobre os experimentos propostos para o *GSink* (mais especificamente com relação aos experimentos 6 e 7), no que diz respeito à métricas para análise de concorrência. ALVES *et al.* [79] propõem importantes métricas para análise de desempenho, como o Grau Máximo de Concorrência em um DDG (Dynamic Dataflow Graph), *Speed-up* Máximo (Grau Médio de Concorrência) em um DDG entre outras. Desta maneira, esta seção será baseada nos estudos propostos por ALVES *et al.* [79].

***Grau Máximo de Concorrência***

A entrada do algoritmo que calcula o grau máximo de concorrência utiliza um DDG $D = (I, E, R, W)$, onde:

- $I$ - Conjunto de instruções do programa;
- $E$ - Conjunto de arestas;
- $R$ - Conjunto de arestas de retorno; e
- $W$ - Conjunto de pesos das instruções.

Tendo em vista o máximo grau de concorrência, inicialmente se faz necessário identificar um subconjunto de $I$ que possua somente instruções dentro de *loops*. Aplicando técnicas de *loop unrolling*, será fornecido o DAG (Directed Acyclic Graph) correspondente. Assim, neste DAG, pode-se verificar todas as dependências entre instâncias de instruções de diversas iterações distintas. A partir do DAG correspondente, calcula-se o grafo de caminho complementar.

O grafo de caminho complementar de um DAG $D_k = (I, E)$ é definido como um grafo não direcionado $C = (I, E')$ no qual uma aresta $(a, b)$ existe em $E'$ se e somente se $a$ e $b$ pertencerem a $I$ e não existir caminho conectando-os em $D_k$ [79]. Assim, uma aresta $(a, b)$ no grafo de caminho complementar indica que $a$ e $b$ são independentes. Assim, a partir do grafo de caminho complementar, o maior clique dará origem ao grau máximo de concorrência.

Para o nosso exemplo em questão, não existem *loops* no DDG originário, que é formado conforme descrito na Figura 5.19.

Note que os 1000 elementos que compõem o multiconjunto são selecionados dois a dois até que se obtenha somente 1 elemento no multiconjunto. Pela inexistência de *loops*, não realizamos a etapa de *loop unrolling*, passando a calcular o grafo de caminho complementar, onde o maior clique dará origem ao grau máximo de concorrência.

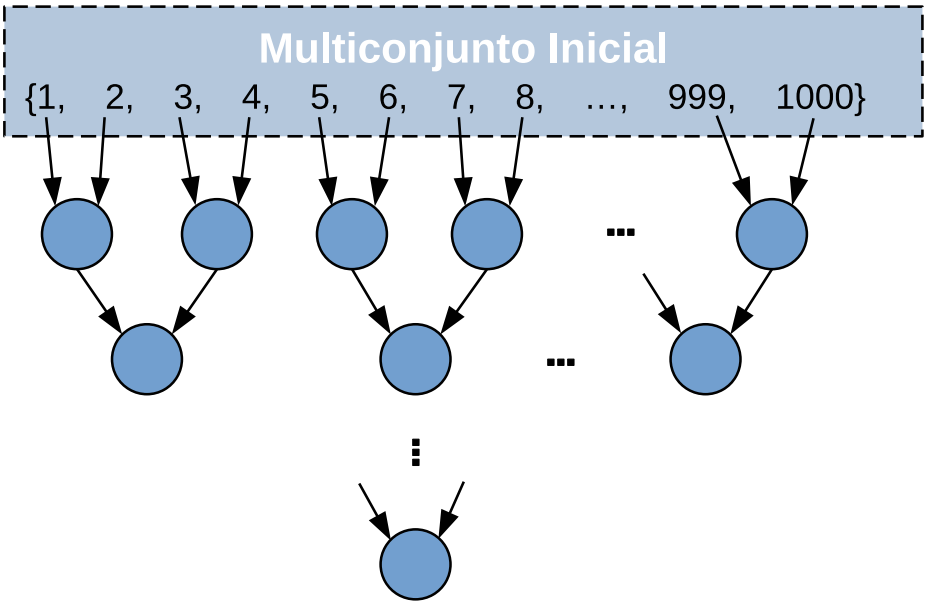


Figura 5.19: Exemplo de DDG - GSink - Experimento 6.

Ora, já na primeira etapa de combinação de elementos do multiconjunto (primeiros vértices a partir da combinação dos elementos do multiconjunto) temos 500 instruções que podem ser executadas em paralelo. Tendo em vista o cálculo do grafo de caminho complementar, veja que estas instruções (primeiros vértices) são independentes, gerando arestas entre si. Como serão geradas tais arestas, teremos um clique de tamanho 500. Note que os demais níveis do grafo sempre possuirão menos instruções (menor quantidade de vértices) uma vez que tal exemplo sempre consome elementos do multiconjunto, diminuindo-o. Assim, este clique com 500 vértices corresponde ao maior clique e, consequentemente, ao grau máximo de concorrência.

***Grau Médio de Concorrência - Máximo Speed-up***

ALVES *et al.* [79] define $T_n$ como sendo o número máximo de passos de computação necessários para executar um DDG com $n$ EPs (Elementos de Processamento). Assim, $T_1$ é a soma dos pesos das instruções, sem que nenhuma instrução seja executada em paralelo, ou seja, quantidade de EP igual a 1. Da mesma forma, $T_\infty$ é o número mínimo de passos de computação necessários para executar um DDG, tendo em vista que neste caso teremos uma quantidade infinita de EPs. Portanto, o máximo de *Speed-up* em um DDG é definido por:

$S_{max} = T_1/T_\infty$

Como o multiconjunto do Experimento 6 é formado por 1000 elementos e estes são combinados dois a dois, teremos a seguinte quantidade de instruções a serem realizadas, em cada etapa de processamento: 500, 250, 125, 62, 32, 16, 8, 4, 2 e 1, totalizando 1000 instruções a serem executadas. Note que aqui consideramos uma "etapa" cada conjunto de vértices (instruções) que poderão ser executados em paralelo. Assim, tomando como base a Figura 5.19, os vértices que estão diretamente interligados por arestas aos elementos do multiconjunto inicial seriam considerados vértices pertencentes à primeira etapa. Os vértices cujas arestas de entrada sejam provenientes de vértices de primeira etapa seria considerados de segunda etapa e assim sucessivamente. Observe ainda que quando descrevemos a quantidade de instruções a serem realizadas em cada etapa, tal quantidade de instruções é sempre reduzida a metade, pela combinação de elementos dois a dois. Isso explica a quantidade de 62 elementos na etapa 4 e de 32 na etapa 5. Na etapa 3, dos 125 elementos, 1 não foi utilizado, devido a quantidade ímpar de elementos. Assim, tal elemento foi utilizado na etapa 5 para compor os 32 elementos.

Ora, levando em consideração uma quantidade de somente um EP disponível, teríamos uma quantidade de 1000 instruções a serem executadas, sem nenhum paralelismo, originando $T_1 = 1000$. Por outro lado, considerando uma quantidade infinita de Elementos de Processamento, poderíamos executar todas as instruções pertencentes a uma etapa de maneira paralela. Assim, como a quantidade de etapas é igual a 10, teríamos $T_\infty = 10$.

Por fim, para o experimento em questão (Experimento 6), o grau médio de concorrência (máximo *Speed-up*) seria dado por:

$$S_{max} = T_1/T_\infty$$
$$S_{max} = 1000/10$$
$$S_{max} = 100$$

As métricas acima fornecidas foram calculadas para o Experimento 6, que consta na Seção 5.2.3. Entretanto, pela quantidade de elementos e natureza das reações envolvidas que combinam elementos dois a dois, os resultados de tais métricas valem para o Experimento 7 desta mesma Seção.

## 5.3 Discussões

O presente Capítulo abordou detalhes envolvidos nos experimentos especificados para o *GFlow* e para o *GSink*, além de apresentar uma discussão sobre os resultados obtidos.

Inicialmente, para o *GFlow*, foram especificados um total de oito experimentos, divididos em duas categorias. A primeira categoria apresentou quatro experimentos, onde as conversões realizadas foram obtidas através de códigos em linguagem de montagem do *TALM* escritos manualmente, ou seja, sem passar pelo processo de compilação do *Couillard*. Dessa forma, apresentamos aspectos inerentes à corretude das conversões realizadas, onde os detalhes de cada conversão de instruções *TALM* em reações Gamma foram abordados. Da mesma maneira, a montagem da listagem inicial de reações e os elementos que compõem o multiconjunto inicial foram apresentados. Já a segunda categoria de experimentos apresentados para o *GFlow* apresentou detalhes de conversão de um código inicialmente escrito em linguagem C, com anotações *THLL*, sua compilação pelo *Couillard*, geração da linguagem de montagem do *TALM* e conversão para um código Gamma a partir do *GFlow*. Nesta ocasião, além dos aspectos de implementação relacionados à conversão propriamente dita, analisamos os resultados obtidos pelo código C inicial e o respectivo código Gamma. Assim, esta segunda categoria de experimentos do *GFlow* foi composta por mais quatro experimentos.

Já a Seção 5.2 foi dedicada a execução de experimentos para o *GSink*. Propusemos um total de sete experimentos, também divididos em duas categorias. A primeira categoria contemplou a execução de cinco experimentos, onde o foco foi apresentar a corretude da execução do *GSink* comparado a duas outras implementações: *Gamma-Sequencial* e *Gamma-MPI*. Além disso, analisamos e apresentamos detalhes dos códigos fonte gerados pelo *Front-End* do *GSink*, visando apresentar a corretude do processo de montagem do código a ser executado pelo *Runtime* do nosso ambiente de execução. A segunda categoria de experimentos apresentou o potencial de ganho de desempenho do *GSink* à medida em que a granularidade das operações realizadas pelas reações aumenta.

Os resultados demonstraram a corretude das conversões realizadas pelo *GFlow*, corroborando o estudo publicado em [24]. Assim, diante das análises realizadas ficou comprovada não somente a corretude nos detalhes específicos de cada conversão, mas também na coerência dos resultados comparado ao Código C inicialmente submetido ao *Couillard*.

Da mesma forma, tendo em vista as comparações com as implementações *Gamma-Sequencial* e *Gamma-MPI* foi demonstrada a corretude de execução do *GSink*, onde os resultados das execuções entre estes três ambientes foram apresentados e discutidos.

Note que, através destes resultados, a intenção de utilização de Gamma como modelo computacional onde a computação permeie os recursos disponíveis ganha maior vulto. A utilização do mecanismo de escalonamento implementado no *GSink* permite a execução paralela de instâncias de reações, o que tem potencial de ga-

nho de desempenho à medida em que a granularidade de tal reação aumente. Um estudo identificando o grau adequado de granularidade para explorar o máximo de paralelismo deverá ser fornecido futuramente, com base na análise de concorrência [79], não fazendo parte do escopo desta Tese.

# Capítulo 6

# Conclusões

O presente Capítulo encerra a apresentação das ideias até aqui expostas e apresenta os resumos das contribuições desta tese, assim como indica caminhos futuros que podem vir a ser seguidos como continuação deste estudo.

## 6.1 Considerações Finais

Nesta tese exploramos a equivalência entre dois modelos computacionais que surgiram como opções para enfrentar desafios encontrados pela computação paralela: Gamma e Dataflow. Ambos consistem em modelos computacionais naturalmente paralelos, onde detalhes de implementação de tal paralelismo são transparentes para o desenvolvedor.

O paradigma computacional Gamma foi proposto em 1986 por BANÂTRE e LE MÉTAYER [4] como um formalismo para especificação de programas, onde o modelo de execução é não determinístico e guarda uma forte relação metafórica com aspectos químicos. Desta forma, os dados encontram-se em um multiconjunto (solução química) onde as operações (reações químicas) sobre este multiconjunto ocorrem de maneira naturalmente paralela e mediante o atendimento de condições específicas (condições de reação). Assim, a computação ocorre de maneira paralela e através de refinamentos do multiconjunto, no que chamamos de modelo caótico de execução [30].

Por outro lado o modelo dataflow caracteriza-se pela execução paralela de instruções a partir da disponibilidade dos operandos necessários a cada tarefa. Neste modelo, um programa pode ser representado por um grafo dirigido, onde os vértices representam as instruções e as arestas caracterizam as dependências de dados. Desta forma, ao contrário do modelo de von Neumman, não existe a necessidade de um Contador de Programa (*Program Counter*), responsável pela busca de instruções, ao passo que cada instrução pode ser executada assim que seus operandos de entrada estejam prontos, possibilitando uma execução computacional naturalmente paralela.

Neste contexto, a equivalência entre os modelos Gamma e Dataflow foi exposta por nós pela primeira vez em nosso trabalho inicial [23], onde a similaridade entre os modelos foi abordada e algoritmos de conversão foram apresentados. Posteriormente, apresentamos um estudo mais completo revisitando o mesmo assunto, agora com enfoque na prova formal de equivalência entre os modelos computacionais descritos [24]. O estudo de equivalência tem potencial para explorar diversos benefícios extras, do ponto de vista de ambos modelos computacionais. Um programa escrito em Gamma pode explorar os benefícios de diversos estudos propostos para dataflow, como por exemplo reutilização de traços de instruções em um grafo dataflow [8] e execução dataflow especulativa e fora de ordem [7]. Por outro lado, um programa expresso através de um grafo dataflow poderá tirar proveito de benefícios de ser executado em um ambiente de execução distribuído e bastante adequado para ser utilizado em conjunto com técnicas de computação aproximativa como o Gamma.

Dessa forma, tendo em vista os conhecimentos obtidos por ocasião de nosso estudo de equivalência entre os modelos computacionais em questão surge a motivação em propor uma ferramenta de conversão Dataflow-Gamma, o *GFlow*. Apresentamos o *GFlow*, seus benefícios, contribuições, considerações sobre sua implementação além de uma série de experimentos visando atestar a corretude das conversões propostas. O *GFlow* permite converter um código C ou dataflow (em linguagem de montagem do *TALM*) em seu equivalente código Gamma. Assim apresentamos detalhes da conversão das diversas categorias de instruções *TALM* para o respectivo código Gamma, além da extração do multiconjunto inicial e formação da listagem inicial de reações. Além disso, apresentamos algumas adequações que foram necessárias, seja por características das implementações de Gamma utilizadas, seja por características do código *TALM* gerado pelo *Couillard*. Abordamos ainda algumas ferramentas *TALM* utilizadas. Os experimentos comprovaram a corretude do *GFlow*. Inicialmente criamos códigos em linguagem de montagem do *TALM* manualmente e, numa segunda categoria de experimentos, utilizamos um código C que foi posteriormente compilado pelo *Couillard* para gerar o *".fl"* necessário à conversão.

Apesar do poder de Gamma em expressar problemas de uma maneira simples e concisa, suas atuais implementações enfrentam problemas, principalmente no que diz respeito a lidar com o não determinismo do modelo. Implementações utilizadas invariavelmente utilizam mecanismos centralizados para gerenciamento do multiconjunto, incorrendo em gargalos computacionais significativos [30]. Assim, propusemos o *GSink*, um ambiente de execução para programas Gamma baseado em [22] que utiliza um mecanismo de escalonamento baseado em reversão de arestas de *sinks* em um grafo orientado acíclico (*SER* [78]). Nossa proposta com o *GSink* não foi prover um ambiente de execução competitivo em termos de desempenho, apesar dos resultados promissores com seu potencial de desempenho, mas sim fornecer a primeira

implementação de Gamma com execução paralela de instâncias de reações. Desta maneira esta pesquisa apresenta o *GSink*, seus benefícios, contribuições, considerações sobre a implementação além de uma série de experimentos visando corretude da execução. Apresentamos detalhes relacionados ao mecanismo de escalonamento utilizado no nosso ambiente de execução, além de informações acerca do *Front-End*, ilustrando a análise léxica e sintática do arquivo Gamma (*".gm"*) de entrada. Os experimentos comprovaram a corretude na execução dos códigos Gamma do GSink, comparados a outras duas implementações do paradigma Gamma.

Dentre as áreas de atuação onde vislumbramos benefícios da utilização de Gamma podemos citar fusão de dados para acompanhamento de contatos no meio militar naval e sistemas de controles e automação, ambos desenvolvidos pela Marinha do Brasil. Em [18] abordamos uma aplicação de fusão de dados em Gamma.

## 6.2 Resumo das Contribuições

O estudo da equivalência entre os modelos computacionais Gamma e Dataflow permitiu explorar benefícios e contribuições para ambos os modelos. A própria proposta de equivalência em si possibilita o compartilhamento de pesquisas já desenvolvidos entre ambos os modelos. Por exemplo, Gamma poderá vir a se beneficiar de estudos relacionados à execução de códigos de maneira especulativa e fora de ordem [7] e reutilização de traços de instruções [25]. Da mesma forma, códigos Dataflow poderão se beneficiar de ambientes de execução Gamma.

Nesse contexto, a proposta do *GFlow* materializa parte destes benefícios, uma vez que proporciona maior versatilidade no desenvolvimento de aplicações paralelas, ao passo que fornece a opção de conversão de um código C ou um grafo dataflow para seu respectivo código Gamma. Além de versatilidade, o *GFlow* traz uma maior expressividade do modelo Gamma, uma vez que incentiva o conhecimento e utilização do modelo.

Entretanto, os ambientes de execução de programas Gamma existentes carecem de investimento em pesquisa, uma vez que ainda são modelos em sua grande parte conceituais e possuem a difícil tarefa de lidar com características de não-determinismo do paradigma Gamma. Dessa maneira, o *GSink* contribui para as implementações do modelo Gamma, ao passo que propõe a implementação de um mecanismo de escalonamento baseado em reversão de arestas de um grafo dirigido acíclico. Trata-se da primeira implementação conceitual que permite que instâncias de diversas reações coexistam. Assim, o *GSink* permite a execução paralela de instâncias de diversas reações. Pelos resultados apresentados tendo em vista o potencial de desempenho do *GSink*, acreditamos que esse modelo de escalonamento seja adequado para utilização em um modelo computacional que permita com que a

computação permeie a rede tendo em vista os recursos computacionais disponíveis (Apêndice C).

Tendo em vista o exposto, podemos elencar o seguinte resumo, relacionado às contribuições fornecidas à comunidade científica, por ocasião da realização deste estudo:

- Apresentação da equivalência entre os modelos computacionais Gamma e Dataflow. Em [23] propomos tal similaridade pela primeira vez;
- Fornecimento da prova formal de equivalência entre os modelos citados e seus algoritmos de transformação [24];
- Fornecimento da primeira implementação de uma ferramenta de conversão entre os modelos computacionais Dataflow e Gamma (*GFlow*);
- Fornecimento da primeira implementação de um ambiente de execução para códigos Gamma que permita a execução paralela de instâncias de reações (*GSink*); e
- Apresentação do potencial de Gamma para utilização de técnicas de computação aproximada, através de alguns experimentos (Apêndice B).

Por fim, a realização deste trabalho possui relação direta com os algoritmos de fusão de dados utilizados e desenvolvidos pela Marinha do Brasil, conforme apresentamos na Seção 1.2. Desta maneira este trabalho contribui para a referida área de atuação, além de indicar relevantes trabalhos futuros que serão elencados na próxima seção.

## 6.3 Trabalhos Futuros

A aplicação das técnicas de redução (apresentadas na Seção 3.1.3) nos algoritmos de conversão (ou em uma nova ferramenta independente) é tema para trabalhos futuros. Neste caso, seria necessário um estudo completo relacionado ao potencial de exploração do paralelismo, que pode ser equalizado com o aumento ou diminuição de elementos computacionais (vértices ou reações).

Conforme apresentado no Apêndice B, Gamma tem potencial para utilização de técnicas de computação aproximativa. Desta forma, será necessário um estudo visando verificar técnicas aplicáveis ao modelo em questão de forma que seja possível adequar desempenho e precisão, seja por término prematuro da computação, seja por parâmetros de precisão utilizados. De toda forma, é interessante e promissor o fornecimento de um estudo onde o modelo sempre execute refinações sucessivas no

multiconjunto, possibilitando o término da computação antes que o estado global de término seja atingido. Da mesma forma, é interessante apresentar um estudo que identifique os domínios de aplicação onde Gamma possa ser utilizado em conjunto com técnicas de computação aproximativa.

Um segmento de pesquisa ainda não explorado refere-se à utilização de multiconjuntos distribuídos em Gamma. Trata-se de estudos visando distribuir e armazenar de maneira eficiente os elementos do multiconjunto pelos nós da rede, respeitando, por exemplo, o conceito de localidade. Desta forma, os custos de comunicação poderiam ser minimizados. Além disso, nosso atual trabalho de pesquisa poderá trazer subsídios para pesquisas relacionadas ao gerenciamento e utilização de membranas no multiconjunto. O conceito permite que existam sub-soluções (subconjuntos) dentro de soluções (multiconjuntos), separadas por uma membrana. Desta forma poderíamos identificar subconjuntos de elementos aos quais determinada reação possui interesse, facilitando a utilização de um protocolo como a *Radnet* e fornecendo um contingenciamento do efeito de uma reação, encapsulando os efeitos da propagação de uma reação sobre todo o multiconjunto.

O fornecimento de um algoritmo e implementação distribuída do mecanismo de reversão de arestas aplicado ao *GSink*, é foco de trabalhos futuros. A intenção é usar um mecanismo onde não seja necessário a utilização de *waves*. Dessa maneira, cada nó da rede poderá se comportar como vértice do grafo de instâncias e realizar um gerenciamento distribuído, onde, instâncias de reações possam vir a ser identificadas e inseridas como vértice do grafo, sem que aja necessidade de término de uma *wave*. Assim sendo, cada vértice implementaria a estrutura de um *ReAgente*, contribuindo para o gerenciamento distribuído do grafo de instâncias. Arestas seriam implementadas como recursos compartilhados, visando garantir a execução de cada *sink*.

Após a proposta da versão distribuída do *GSink*, uma análise de desempenho comparativa utilizando o algoritmo de Par de Plots em Dois Estágios (PPDE), implementado em Gamma [17], e outros algoritmos de fusão de dados recentemente utilizados pela Marinha do Brasil é fortemente recomendada. Note que o *GFlow* fornece funcionalidades que podem ser exploradas aqui, uma vez que algoritmos mais recentes podem ser convertidos para Gamma através de nossa ferramenta. Assim, através de tal estudo, poderia ser fornecido ainda um *benchmark* ou a realização da análise de complexidade algorítmica entre a paralelização utilizando ferramentas tradicionais e a utilização do Gamma.

Uma área não explorada neste estudo, mas que Gamma traz boas perspectivas de trabalho é tolerância a falhas e resiliência. Seria bastante intuitivo partir para uma solução que compreenda estas áreas através da inserção probabilística de subconjuntos redundantes no multiconjunto.

Outra indicação de trabalho futuro consiste em fornecer uma implementação de Gamma que possibilite a alteração da granularidade da reação. Tal fato tende a possibilitar melhores resultados de execução para o *GSink*, além de contribuir para Gamma ser utilizado como plataforma computacional que permeie os recursos computacionais disponíveis. Um estudo identificando o grau adequado de granularidade para explorar o máximo de paralelismo com base na análise de concorrência [79], também é indicado como trabalho futuro.

Por fim, merece uma análise a viabilidade do uso do modelo Gamma como paradigma computacional para a Computação Quântica. Desta forma, estudos iniciais visam provar questões de reversibilidade do modelo, fator este necessário para a utilização em computação quântica.

# Referências Bibliográficas


[1] HENNESSY, J. L., PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach.* 5th ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[2] GOLDMAN, A., UEDA, A. H., MATSUBARA, C. M., et al. "Arquitetura de Computadores: educação, ensino e aprendizado". cap. Modelos para Programação em Arquitetura Multi-Core, Sociedade Brasileira de Computação, 2012. ISBN: 978-85-7669-263-8.

[3] DENNIS, J. B., FOSSEEN, J. B., LINDERMAN, J. P. "Data flow schemas". In: Ershov, A., Nepomniaschy, V. A. (Eds.), *International Symposium on Theoretical Programming*, pp. 187–216, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg. ISBN: 978-3-540-38012-2.

[4] BANÂTRE, J.-P., LE MÉTAYER, D. *A New Computational Model and Its Discipline of Programming.* In: Rapport de Recherche 566, INRIA, France, 1986.

[5] DE MELLO JUNIOR, R. R., DE ARAÚJO, L. S., DUTRA, D. L. C., et al. "Fluid Computing: Interest-Based Communication in Dataflow/Multiset Rewriting Computing". In: *Proceedings of the 10th Euro-American Conference on Telematics and Information Systems*, EATIS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN: 9781450377119. doi: 10.1145/3401895.3401932. Disponível em: <`https://doi.org/10.1145/3401895.3401932`>.

[6] DE ALMEIDA, R. H. P., MELLO, R. R., PAILLARD, G. A. L., et al. "A GPU-Based Implementation for the Gamma Multiset Rewriting Paradigm". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, p. 1225–1230, New York, NY, USA, Association for Computing Machinery, 2016. ISBN: 9781450337397. Disponível em: <`https://doi.org/10.1145/2851613.2851719`>.

[7] MARZULO, L. A. J., FRANCA, F. M. G., COSTA, V. S. "Transactional WaveCache: Towards Speculative and Out-of-Order DataFlow Execution of Memory Operations". In: *2008 20th International Symposium on Computer Architecture and High Performance Computing*, pp. 183–190, 2008. doi: 10.1109/SBAC-PAD.2008.29.

[8] ROUBERTE, L., SENA, A. C., NERY, A. S., et al. "DF-DTM: Dynamic Task Memoization and reuse in dataflow", *Concurrency and Computation: Practice and Experience*, v. 0, n. 0, pp. e4937, 2018. doi: 10.1002/cpe.4937. Disponível em: <`https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4937`>. e4937 cpe.4937.

[9] REINDERS, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. California, USA, O'Reilly Media, Inc., 2007.

[10] DAGUM, L., MENON, R. "OpenMP: an industry standard API for shared-memory programming", *IEEE Computational Science and Engineering*, v. 5, n. 1, pp. 46–55, 1998. doi: 10.1109/99.660313.

[11] ABADI, M. "TensorFlow: Learning Functions at Scale". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, p. 1, New York, NY, USA, 2016. Association for Computing Machinery. ISBN: 9781450342193. doi: 10.1145/2951913.2976746. Disponível em: <`https://doi.org/10.1145/2951913.2976746`>.

[12] ABADI, M., BARHAM, P., CHEN, J., et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, Savannah, GA, nov. 2016. USENIX Association. ISBN: 978-1-931971-33-1. Disponível em: <`https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`>.

[13] ALDINUCCI, M., DANELUTTO, M., KILPATRICK, P., et al. "Fastflow: high-level and efficient streaming on multi-core", *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2017.

[14] MARZULO, L. A., ALVES, T. A., FRANÇA, F. M., et al. "TALM: A Hybrid Execution Model with Distributed Speculation Support". In: *2010 22nd International Symposium on Computer Architecture and High Performance Computing Workshops*, pp. 31–36, 2010. doi: 10.1109/SBAC-PADW.2010.8.

[15] ALVES, T. A. O., GOLDSTEIN, B. F., FRANÇA, F. M. G., et al. "A Minimalistic Dataflow Programming Library for Python". In: *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, pp. 96–101, Oct 2014. doi: 10.1109/SBAC-PADW.2014.20.

[16] BARBOSA, R. P. *Fusão de Alvos Utilizando Grafos em Ambientes de Múltiplos Sensores*. Tese de Doutorado, UFRJ, COPPE, 2012.

[17] JUNIOR, R. R., ALMEIDA, R. H., FRANÇA, F. M., et al. "A Parallel Implementation of Data Fusion Algorithm Using Gamma". In: *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, v. 00, pp. 109–114, Oct. 2016. doi: 10.1109/SBAC-PADW.2015.25. Disponível em: <doi.ieeecomputersociety.org/10.1109/SBAC-PADW.2015.25>.

[18] DE MELLO JUNIOR, R. R. *Fusão de dados em gamma*. Dissertação de Mestrado, PESC/COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 2015.

[19] BARBOSA, R., LIVERNET, F., DE LIMA, B., et al. "Multisensor Data Fusion Using Two-Stage Analysis on Pairs of Plots Graphs". In: *15th International Conference on Informating Fusion, Singapore*, pp. 2073–2078, 2012.

[20] SHI, W., CAO, J., ZHANG, Q., et al. "Edge Computing: Vision and Challenges", *IEEE Internet of Things Journal*, v. 3, n. 5, pp. 637–646, 2016. doi: 10.1109/JIOT.2016.2579198.

[21] BAUER, A. C., ABBASI, H., AHRENS, J., et al. "In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms", *Computer Graphics Forum*, v. 35, n. 3, pp. 577–597, 2016. doi: https://doi.org/10.1111/cgf.12930. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12930>.

[22] PAILLARD, G., FRANCA, F., FILHO, J. "Uma Proposta de um Escalonador para Gamma". In: *II Workshop em Sistemas Computacionais de Alto Desempenho*, pp. 47–54, Pirenópolis, GO, 2001.

[23] MELLO, R. R., ARAÚJO, L. S., ALVES, T. A. O., et al. "Exploring the Equivalence between Dynamic Dataflow Model and Gamma - General Abstract Model for Multiset mAnipulation". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 809–816, 2019. doi: 10.1109/IPDPSW.2019.00133.

[24] MELLO JR., R. R., DE ARAÚJO, L. S., ALVES, T. A. O., et al. "Gamma — General Abstract Model for Multiset mAnipulation and dynamic dataflow model: An equivalence study", *Concurrency and Computation: Practice and Experience*, v. 33, n. 11, pp. e6176, 2021. doi: https://doi.org/10.1002/cpe.6176. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6176>.

[25] ROUBERTE, L., SENA, A. C., NERY, A. S., et al. "DF-DTM: Dynamic Task Memoization and reuse in dataflow", *Concurrency and Computation: Practice and Experience*, v. 31, n. 18, pp. e4937, 2019. doi: https://doi.org/10.1002/cpe.4937. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4937>. e4937 cpe.4937.

[26] BURKS, A. W., GOLDSTINE, H. H., VON NEUMANN, J. "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)". In: *Perspectives on the Computer Revolution*, p. 39–48, USA, Ablex Publishing Corp., 1989. ISBN: 0893913693.

[27] BANÂTRE, J.-P., FRADET, P., MÉTAYER, D. L. "Gamma and the Chemical Reaction Model: Fifteen Years After". In: *Proceedings of the Workshop on Multiset Processing: Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, WMP '00, pp. 17–44, London, UK, UK, 2001. Springer-Verlag. ISBN: 3-540-43063-6. Disponível em: <http://dl.acm.org/citation.cfm?id=647269.721851>.

[28] BANÂTRE, J.-P., LE MÉTAYER, D. "The Gamma Model and Its Discipline of Programming", *Sci. Comput. Program.*, v. 15, n. 1, pp. 55–77, nov. 1990. ISSN: 0167-6423. doi: 10.1016/0167-6423(90)90044-E. Disponível em: <http://dx.doi.org/10.1016/0167-6423(90)90044-E>.

[29] DIJKSTRA, E. W. *A Discipline of Programming.* Upper Saddle River, NJ, USA, Prentice-Hall, 1976. ISBN: 013215871X.

[30] DE ALMEIDA, R. H. P. *Uma Derivação do Paradigma de Reescrita de Multiconjuntos Gamma para a Arquitetura GPU.* Dissertação de Mestrado, PESC/COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 2015.

[31] PAILLARD, G. A. L. *Uma Implementação Paralela e Distribuída de Gamma Estruturada.* Dissertação de Mestrado, PESC/COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 1999.

[32] BANÂTRE, J.-P., LE MÉTAYER, D. "Coordination Programming". Imperial College Press, cap. Gamma and the Chemical Reaction Model: Ten Years After, pp. 3–41, London, UK, UK, 1996. ISBN: 1-86094-023-4. Disponível em: <http://dl.acm.org/citation.cfm?id=270347.270348>.

[33] BANÂTRE, J.-P., LE MÉTAYER, D. "Programming by Multiset Transformation", *Commun. ACM*, v. 36, n. 1, pp. 98–111, jan. 1993. ISSN: 0001-0782. doi: 10.1145/151233.151242. Disponível em: <http://doi.acm.org/10.1145/151233.151242>.

[34] FRADET, P., LE MÉTAYER, D. "Structured Gamma", *Sci. Comput. Program.*, v. 31, n. 2-3, pp. 263–289, jul. 1998. ISSN: 0167-6423. doi: 10.1016/S0167-6423(97)00023-3. Disponível em: <http://dx.doi.org/10.1016/S0167-6423(97)00023-3>.

[35] HANKIN, C., LE MÉTAYER, D., SANDS, D. *A calculus of Gamma programs*. Research Report RR-1758, 1992. Disponível em: <https://hal.inria.fr/inria-00076998>.

[36] SANDS, D. "A Compositional Semantics of Combining Forms for Gamma Programs". In: *International Conference on Formal Methods in Programming and Their Applications*. lncs, 1993.

[37] CIANCARINI, P., GORRIERI, R., ZAVATTARO, G. "An Alternative Semantics for the Calculus of Gamma Programs". In: Andreoli, J., Hankin, C., LeMetayer, D. (Eds.), *Coordination Programming: Mechanisms, Models and Semantics*, Imperial College Press, pp. 232–248, 1996.

[38] MÉTAYER, D. L. "Higher-Order Multiset Programming". In: *in Proc. of the DIMACS workshop on specifications of parallel algorithms, American Mathematical Society, Dimacs series in Discrete Mathematics*. American Mathematical Society, 1994.

[39] BANÂTRE, J. P., FRADET, P., RADENAC, Y. "Higher-Order Chemical Programming Style". In: *Proceedings of the 2004 International Conference on Unconventional Programming Paradigms*, UPP'04, pp. 84–95, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN: 3-540-27884-2, 978-3-540-27884-9. doi: 10.1007/11527800_7. Disponível em: <http://dx.doi.org/10.1007/11527800_7>.

[40] BANÂTRE, J.-P., FRADET, P., RADENAC, Y. "Higher-order Chemical Model of Computation". In: *The Grand Challenge in Non-Classical Computation*, April 2005.

[41] BANÂTRE, J.-P., FRADET, P., RADENAC, Y. "Higher-Order Chemistry". In: *Pre-proceedings of Workshop on Membrane Computing (WMC 2003)*, pp. 53–60, July 2003. Disponível em: <http://www.irisa.fr/paris/Biblio/Papers/Banatre/BanFraRad03WMC.pdf>.

[42] CREVEUIL, C. "Implementation of Gamma on the Connection Machine." In: Banâtre, J.-P., Métayer, D. L. (Eds.), *Research Directions in High-Level Parallel Programming Languages*, v. 574, *Lecture Notes in Computer Science*, pp. 219–230. Springer, 1991. ISBN: 3-540-55160-3. Disponível em: <http://dblp.uni-trier.de/db/conf/hlppp/hlppp1991.html#Creveuil91>.

[43] HUANG, L., TONG, W., KAM, W. N., et al. "Implementation of gamma on a massively parallel computer", *Journal of Computer Science and Technology*, v. 12, n. 1, pp. 29–39, 1997.

[44] BANÂTRE, J.-P., COUTANT, A., METAYER, D. L. "A parallel machine for multiset transformation and its programming style", *Future Generation Computer Systems*, v. 4, n. 2, pp. 133 – 144, 1988. ISSN: 0167-739X. doi: http://dx.doi.org/10.1016/0167-739X(88)90012-X. Disponível em: <http://www.sciencedirect.com/science/article/pii/0167739X8890012X>.

[45] GLADITZ, K., KUCHEN, H. "Parallel Implementation of the Gamma-Operation on Bags". In: *Proceedings of PASCO '94*, pp. 154–163, 1994.

[46] "The Message Passing Interface (MPI) Standard". Disponível em <http://www.mcs.anl.gov/research/projects/mpi/index.htm>.

[47] PAILLARD, G., FRANCA, F., FILHO, J. "A distributed implementation of Structured Gamma". In: *Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings. Eighth International Conference on*, pp. 445–450, 2001. doi: 10.1109/ICPADS.2001.934852.

[48] BERRY, G., BOUDOL, G. "The Chemical Abstract Machine". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT annual symposium on principles of programming languages*, pp. 81–94, 1990.

[49] WANG, C., PRIOL, T. *HOCL Programming Guide.* Technical Report hal-00705283, INRIA, France, 2009.

[50] FERNÁNDEZ, H., TEDESCHI, C., PRIOL, T. "A Chemistry-Inspired Workflow Management System for Decentralizing Workflow Execution", *IEEE*

*Transactions on Services Computing*, v. 9, n. 2, pp. 213–226, 2016. doi: 10.1109/TSC.2013.27.

[51] DENNIS, J., FOSSEEN, J. B. *Introduction to data flow schemas. Technical Memorandum Memo-81-1, Laboratory for Computer Science*. Massachusetts Institute of Technology, 1973.

[52] DENNIS, J. B. "The varieties of data flow computers". In: *Advanced computer architecture*, pp. 51–60, 1986.

[53] SWANSON, S., SCHWERIN, A., MERCALDI, M., et al. "The wavescalar architecture", *ACM Transactions on Computer Systems (TOCS)*, v. 25, n. 2, pp. 1–54, 2007.

[54] DE ARAÚJO, L. S. *Otimizando Laços em Computação por Fluxo de Dados*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2016.

[55] LEE, B., HURSON, A. R. "Issues in dataflow computing". In: *Advances in computers*, v. 37, Elsevier, pp. 285–333, 1993.

[56] DENNIS, J. B., MISUNAS, D. P. "A Preliminary Architecture for a Basic Data-Flow Processor". In: *Proceedings of the 2nd Annual Symposium on Computer Architecture*, ISCA '75, p. 126–132, New York, NY, USA, 1974. Association for Computing Machinery. ISBN: 9781450373661. doi: 10.1145/642089.642111. Disponível em: <`https://doi.org/10.1145/642089.642111`>.

[57] ARVIND, CULLER, D. E. "Tagged token dataflow architecture", 10 1983. Disponível em: <`https://www.osti.gov/biblio/5258727`>.

[58] JOHNSTON, W. M., HANNA, J. P., MILLAR, R. J. "Advances in dataflow programming languages", *ACM computing surveys (CSUR)*, v. 36, n. 1, pp. 1–34, 2004.

[59] STERLING, T. "Studies on optimal task granularity and random mapping", *Advanced Topics in Dataflow Computing and Multithreading, 1995*, pp. 349–365, 1995.

[60] ROUBERTE, L., SENA, A., NERY, A., et al. "DF-DTM: explorando redundância de tarefas em Dataflow". In: *Anais do XVII Simpósio em Sistemas Computacionais de Alto Desempenho*. SBC, 2016.

[61] MARZULO, L. A. J. *Explorando Linhas de Execuçao Paralelas com Programaçao Orientada por Fluxo de Dados*. Tese de Doutorado, Tese de Doutorado, COPPE-UFRJ, 2011.

[62] PAILLARD, G. A. L., FRANÇA, F. M. G., FILHO, J. M. "A Distributed Implementation of Structured Gamma". In: *International Conference on Parallel and Distributed Systems, KiongJu City*, pp. 445–450, 2001.

[63] FRANÇA, F. M. G., FARIA, L. "Optimal mapping of neighbourhood-constrained systems". In: Ferreira, A., Rolim, J. (Eds.), *Parallel Algorithms for Irregularly Structured Problems*, pp. 165–170, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN: 978-3-540-44915-7.

[64] DE OLIVEIRA ALVES, T. A. *EXECUÇÃO ESPECULATIVA EM UMA MÁQUINA VIRTUAL DATAFLOW*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2010.

[65] MARZULO, L. A., ALVES, T. A., FRANÇA, F. M., et al. "Couillard: Parallel programming via coarse-grained Data-flow Compilation", *Parallel Computing*, v. 40, n. 10, pp. 661 – 680, 2014. ISSN: 0167-8191. doi: https://doi.org/10.1016/j.parco.2014.10.002. Disponível em: <`http://www.sciencedirect.com/science/article/pii/S0167819114001252`>.

[66] TOMASULO, R. M. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research and Development*, v. 11, n. 1, pp. 25–33, 1967. doi: 10.1147/rd.111.0025.

[67] BEAZLEY, D. "Ply (python lex-yacc) v3. 10", *See http://www. dabeaz. com/ply*, 2001.

[68] ELLSON, J. "Graphviz-graph visualization software", *http://www. graphviz. org/*, 2008.

[69] CREVEUIL, C. "Implementation of Gamma on the connection machine". In: Banâtre, J. P., Le Métayer, D. (Eds.), *Reasearch Directions in High-Level Parallel Programming Languages*, pp. 219–230, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN: 978-3-540-46762-5.

[70] HUANG, L., TONG, W., KAM, W. N., et al. "Implementation of GAMMA on a massively parallel computer", *Journal of Computer Science and Technology*, v. 12, n. 1, pp. 29–39, Jan 1997. ISSN: 1860-4749. doi: 10.1007/BF02943142. Disponível em: <`https://doi.org/10.1007/BF02943142`>.

[71] KNUTH, D. E. *Art of Computer Programming, Volumes 1-4A Boxed Set*. 3rd ed. , Addison-Wesley Professional, 2011. ISBN: 0321751043, 9780321751041.

[72] DE ALMEIDA, R. H. P., MELLO, JR., R. R., PAILLARD, G. A. L., et al. "A GPU-based Implementation for the Gamma Multiset Rewriting Paradigm". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pp. 1225–1230, New York, NY, USA, 2016. ACM. ISBN: 978-1-4503-3739-7. doi: 10.1145/2851613.2851719. Disponível em: <http://doi.acm.org/10.1145/2851613.2851719>.

[73] BANĂTRE, J.-P., FRADET, P., LE MÉTAYER, D. "Gamma and the Chemical Reaction Model: Fifteen Years After". In: Calude, C. S., PĂun, G., Rozenberg, G., et al. (Eds.), *Multiset Processing*, pp. 17–44, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN: 978-3-540-45523-3.

[74] DE MELLO JR.AND GABRIEL A. L. PAILLARD, R. R., FRANÇA, F. M. G. "Fluid Computing: uma proposta para utilização de uma máquina química abstrata como plataforma paralela distribuida", *Revista Sistemas e Mídias Digitais (RSMD)*, v. 4, n. 1, abril 2019. ISSN: 2525-9555. Disponível em: <http://revistasmd.virtual.ufc.br/arquivos/volume-4/numero-1/rsmd-v4-n1-p9.pdf>.

[75] KHOKHAR, A. A., PRASANNA, V. K., SHAABAN, M. E., et al. "Heterogeneous computing: Challenges and opportunities", *Computer*, v. 26, n. 6, pp. 18–27, 1993.

[76] TOPCUOGLU, H., HARIRI, S., WU, M.-Y. "Performance-effective and low-complexity task scheduling for heterogeneous computing", *IEEE transactions on parallel and distributed systems*, v. 13, n. 3, pp. 260–274, 2002.

[77] BRODTKORB, A. R., DYKEN, C., HAGEN, T. R., et al. "State-of-the-art in heterogeneous computing", *Scientific Programming*, v. 18, n. 1, pp. 1–33, 2010.

[78] BARBOSA, V. C. *Massively Parallel Models of Computation: Distributed Parallel Processing in Artificial Intelligence and Optimisation*. USA, Ellis Horwood, 1993. ISBN: 0135629683.

[79] ALVES, T. A. O., MARZULO, L. A. J., KUNDU, S., et al. "Concurrency Analysis in Dynamic Dataflow Graphs", *IEEE Transactions on Emerging Topics in Computing*, v. 9, n. 1, pp. 44–54, 2021. doi: 10.1109/TETC.2018.2799078.

[80] HAN, J., ORSHANSKY, M. "Approximate computing: An emerging paradigm for energy-efficient design". In: *2013 18th IEEE European Test Symposium (ETS)*, pp. 1–6, May 2013. doi: 10.1109/ETS.2013.6569370.

[81] DUTRA, R. C., MORAES, H. F., AMORIM, C. L. "Interest-centric mobile ad hoc networks". In: *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, pp. 130–138. IEEE, 2012.

[82] ALVES, T. A. O., MARZULO, L. A. J., FRANÇA, F. M. G., et al. "Trebuchet: Exploring TLP with Dataflow Virtualisation", *Int. J. High Perform. Syst. Archit.*, v. 3, n. 2/3, pp. 137–148, maio 2011. ISSN: 1751-6528. doi: 10.1504/IJHPSA.2011.040466. Disponível em: <http://dx.doi.org/10.1504/IJHPSA.2011.040466>.

[83] DUTRA, D. L. C., MORAES, H. F., AMORIM, C. L. "RadFlow: An Interest-Centric Task Based Dataflow Runtime". In: *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, pp. 115–119, Oct 2015. doi: 10.1109/SBAC-PADW.2015.26.

# Apêndice A

# Listagem de Publicações

1. MACEDO, E. L. C., DE OLIVEIRA, E. A. R., SILVA, F. H., et al. **"On the security aspects of Internet of Things: A systematic literature review"**, *Journal of Communications and Networks*, v. 21, n. 5, pp. 444–457, 2019. doi: 10.1109/JCN.2019.000048.
   - A referida publicação fornece uma revisão sistemática da literatura (*Systematic Literature Review* - SLR) visando identificar algumas questões de segurança em Internet das Coisas (*Internet of Things* - IoT). Para tanto, nossa revisão da literatura abordou quatro principais aspectos de segurança: autenticação, controle de acesso, proteção de dados e confiança. O trabalho descreve o protocolo utilizado para a condução da SLR. Trata-se do primeiro trabalho a realizar uma SLR tendo em vista os aspectos de segurança elencados. O referido artigo foi produzido como fruto dos estudos realizados na disciplina de Internet das Coisas.

2. MELLO, R. R., ARAÚJO, L. S., ALVES, T. A. O., et al. **"Exploring the Equivalence between Dynamic Dataflow Model and Gamma — General Abstract Model for Multiset mAnipulation"**. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 809–816, 2019. doi: 10.1109/IPDPSW.2019.00133.
   - A similaridade entre os modelos computacionais Gamma e Dataflow foi apresentada pela primeira vez neste trabalho. Tal equivalência foi proposta através de alguns testes empíricos, apresentação de conceitos básicos e algoritmos de conversão, tendo em vista apresentar a ideia básica por trás da equivalência, onde um vértice de um grafo dataflow guarda correspondência com uma reação Gamma enquanto que as arestas deste grafo correspondem aos dados que compõem o multiconjunto. Além disso, indicaram-se caminhos de pesquisa que visavam alteração da granularidade, tanto das reações Gamma, quanto de vértices dataflow, através da proposta de "Reduções".

3. MELLO JUNIOR, R. R. ; PAILLARD, GABRIEL A.L. ; FRANÇA, FELIPE M. G.. **"Fluid Computing: uma proposta para utilização de uma máquina química abstrata como plataforma paralela distribuída"**, Revista Sistemas e Mídias Digitais (RSMD), v. 4, n. 1, abril 2019. ISSN: 2525-9555. Disponível em: <http://revistasmd.virtual.ufc.br/arquivos/volume-4/numero-1/rsmd-v4-n1-p9.pdf>.

   - Trata-se de um resumo expandido publicado na Revista Sistemas e Mídias Digitais (RSMD) da Universidade Federal do Ceará (UFC). Na ocasião foram publicados os resultados parciais e indicações de trabalhos que visavam fornecer uma plataforma onde a computação permearia a rede pela utilização robusta e segura dos recursos computacionais disponíveis. Desta forma, foram apresentadas a motivação, descrição do estágio atual da pesquisa e sua relevância.

4. DE MELLO JUNIOR, R. R., DE ARAÚJO, L. S., DUTRA, D. L. C., et al. **"Fluid Computing: Interest-Based Communication in Dataflow/Multiset Rewriting Computing"**. In: *Proceedings of the 10th Euro-American Conference on Telematics and Information Systems*, EATIS'20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN: 9781450377119. doi: 10.1145/3401895.3401932. Disponível em: <https://doi.org/10.1145/3401895.3401932>.

   - Tendo em vista as indicações de pesquisa realizadas no trabalho anterior, neste artigo foi sugerido um novo ambiente de execução para programas Gamma. Assim, a proposta foi explorar o modelo computacional Gamma, de forma a sugerir um novo ambiente de execução para programas Gamma, onde a arquitetura de tal modelo de execução seria baseada em um grafo dataflow. Além disso, foi proposta a utilização da Radnet (protocolo de comunicação baseado em interesses) como protocolo de comunicação, uma vez que a identificação de interesses em um grafo dataflow ocorre de maneira mais simples do que em um programa Gamma.

5. MELLO JR., R. R., DE ARAÚJO, L. S., ALVES, T. A. O., et al. **"Gamma — General Abstract Model for Multiset mAnipulation and dynamic dataflow model: An equivalence study"**, *Concurrency and Computation: Practice and Experience*, v. 33, n. 11, pp. e6176, 2021. doi: https://doi.org/10.1002/cpe.6176. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6176>.

- O primeiro trabalho que apresentou a equivalência entre os modelos Gamma e Dataflow foi selecionado para ser estendido para um trabalho maior. Desta forma, esta publicação apresenta a prova formal de equivalência entre os modelos em questão. Inicialmente foi proposto um estudo visando generalizar o tratamento dos dados de entrada e saída (sejam arestas em um grafo dataflow, sejam elementos do multiconjunto em Gamma). Posteriormente, tal generalização foi utilizada para a realização da prova formal. Na ocasião, ainda foram efetuadas algumas considerações sobre o não determinismo de Gamma e informações sobre validação experimental.

# Apêndice B

# Potencial de Gamma para Computação Aproximativa

Muitos sistemas e aplicações podem tolerar alguma perda de qualidade ou precisão nos resultados computacionais. Como exemplo destes domínios de aplicação, podemos citar: processamento de mídia (áudio, vídeo, gráficos e imagens), reconhecimento de padrões, *data mining* e fusão de dados. Relaxando a necessidade de operações totalmente precisas ou completamente determinísticas, a computação aproximativa surge como área potencial para equilibrar precisão e desempenho. Técnicas de computação aproximativa podem ser implementadas em níveis de *software* ou *hardware*. Por exemplo, *Full Adders* aproximativos, multiplicadores aproximativos e Síntese Lógica aproximada são técnicas de computação aproximativa baseadas em *hardware*, enquanto refinamentos incrementais e adaptação dinâmica de largura de *bits* são técnicas implementadas por *software* [80].

O modelo computacional Gamma possui um potencial para utilização de técnicas de computação aproximativa. As reações no paradigma Gamma podem ocorrer livremente, realizando refinamentos sucessivos no multiconjunto. Assim, a computação pode parar a qualquer momento antes do término global do programa, apresentando um resultado aproximado. Portanto, quanto mais próximo do final do programa, mais preciso será o resultado. Por exemplo, considere um simples programa em Gamma para calcular o menor elemento em um multiconjunto. Tal programa em Gamma pode ser expresso pela Equação B.1:

$$\begin{aligned} R = replace(x, y) \\ by\ \ x \\ where\ \ x < y \end{aligned} \tag{B.1}$$

Assim, um par de elementos do multiconjunto é comparado e, no caso do primeiro elemento ser o menor, o segundo será removido. Portanto, cada iteração do algoritmo refinará o multiconjunto removendo os elementos maiores. Assim, podemos modificar o ambiente de execução Gamma permitindo que o programador pare a computação a qualquer momento ou pare após alguma parte dos elementos terem sido processados (por exemplo, pare o cálculo se 96 % dos elementos do multiconjunto forem processados).

## B.1 Resultados Experimentais

Tendo em vista demonstrar o potencial de Gamma para computação aproximativa, propomos um simples experimento de processamento de imagem. Assim, utilizamos uma matriz $10 \times 10$, onde cada elemento irá representar um pixel de uma imagem, cujo valor varia de 0 a 255. Inicialmente, estamos interessados em calcular o valor médio destes pixels. Se o valor do pixel é menor que o valor médio, então o valor do pixel será modificado para 0. Da mesma maneira, se o valor do pixel for maior que a média calculada, este será modificado para 1.

Para implementar uma matriz em Gamma, uma possível solução é representar cada elemento da matriz por uma n-tupla composta por três elementos: linha (r), coluna (c) e valor ($[r, c, value]$). Assim, a tupla [2,8,253] corresponde ao elemento cuja linha é 2, coluna 8 e possui o valor de pixel igual a 253. Portanto, nosso multiconjunto possui 100 tuplas representando os elementos da matriz. Além disso, temos mais um elemento inicial no multiconjunto: uma tupla de somente um elemento, representando o valor inicial de média.

Nossos experimentos foram executados em um Laptop Sony Vaio Intel Core 2 Duo Centrino com 2GB de Memória RAM. O sistema operacional utilizado foi o Linux, distribuição Debian 9 Stretch. Utilizamos uma implementação de Gamma proposta por Juarez Muylaert, que roda em um ambiente distribuído através de troca de mensagens utilizando protocolo MPI [62]. Cada experimento foi executado por 15 vezes. A Tabela B.1 apresenta o valor médio e o desvio padrão dos experimentos descritos abaixo.

Tabela B.1: Resultados Experimentais.

| | Exp 1 | Exp 2 | Exp 3 |
|---|---|---|---|
| Tempo médio (seg) | 30,205 | 25,448 | 24,647 |
| Desvio Padrão | 1,2223 | 1,1347 | 0,9559 |

### B.1.1 Experimento 1

Em nosso primeiro experimento, o programa Gamma foi composto por 4 diferentes reações. A primeira reação, $R1$, executa a soma de todos os 100 valores que representam pixels nas tuplas. A segunda reação, $R2$, calcula o valor médio, dividindo o produto da soma pela quantidade de elementos. $R3$ modifica o valor de todo o pixel cujo valor for menor que a média calculada. Em paralelo com $R3$, $R4$ modifica o valor do pixel para 1 se este valor for maior que a média. Assim, o programa Gamma executa $R1$ e $R2$ de maneira sequencial e, então, calcula $R3$ e $R4$ em paralelo ($R1$; $R2$; $R3 \mid R4$). As reações mencionadas estão descritas a seguir:

```
R1 = replace [r,c,value],[average]
by [r,c,value,1], [(average+value)]
if true

R2 = replace [average]
by [average/100,1]
if true

R3 = replace [r, c, value,tag1], [average,tag2]
by [r,c, 0, 0], [average,tag2]
if (value < average) and (tag1 == 1)

R4 = replace [r, c, value,tag1], [average,tag2]
by [r,c, 1, 0], [average,tag2]
if (value >= average) and (tag1 == 1)
```

### B.1.2 Experimento 2

A diferença entre este segundo experimento e o primeiro que foi apresentado é que agora calculamos a média parcial. Desta forma, a primeira reação $R1$ é responsável por acumular a soma de todos os valores divididos por 100 (quantidade de elementos). Após isso, a segunda e terceira reações ($R2$ e $R3$) modificam o valor do pixel para 0 ou 1 comparando o com o valor médio. Assim, a ordem de execução das reações foi $R1$; $R2 \mid R3$. Repare que temos aqui somente três reações ao invés de quatro, conforme apresentado no primeiro experimento. Observe também que a implementação de Gamma utilizada aqui suporta somente números inteiros. A intenção deste experimento é apresentar que é possível relaxar a precisão (neste caso o cálculo da média) em prol de um aumento de desempenho. Por exemplo, o valor médio calculado para o primeiro experimento foi 122. já no segundo experimento o programa calculou 81 como média. Isso ocorre pois cada cálculo da média parcial perde a parte decimal do número inteiro, ocasionando perda de precisão. Porém,

como podemos ver na Tabela B.1, este segundo experimento obteve tempo médio de execução de 25.448 segundos, contra 30.205 segundos do primeiro experimento.

```
R1 = replace [r,c,value], [average]
by [r,c,value,1], [average + value/100]
if true

R2 = replace [r, c, value,tag1], [average]
by [r,c, 0, 0], [average]
if (value < average) and (tag1 == 1)

R3 = replace [r, c, value,tag1], [average]
by [r,c, 1, 0], [average]
if (value >= average) and (tag1 == 1)
```

### B.1.3 Experimento 3

Neste experimento, apresentamos uma maneira diferente de explorar os resultados aproximados. O valor médio foi calculado da mesma maneira do primeiro experimento (por $R1$ e $R2$). Entretanto, não processamos os elementos que compõem a borda da figura. Em outras palavras, não processamos elementos do multiconjunto cujo valor de linha (r) fosse igual a 0 ou 9 ou valor de coluna (c) igual a 0 ou 9. Em comparação com o tempo de execução do primeiro experimento (30.205 segundos), este experimento alcançou um tempo de execução de 24.647 segundos (Tabela B.1).

```
R1 = replace [r,c,value],[average]
by [r,c,value,1], [(average+value)]
if true

R2 = replace [average]
by [average/100,1]
if true

R3 = replace [r, c, value,tag1], [average,tag2]
by [r,c, 0, 0], [average,tag2]
if (value < average) and (tag1 == 1) and (r != 0)
and (r != 9) and (c != 0) and (c != 9)

R4 = replace [r, c, value,tag1], [average,tag2]
by [r,c, 1, 0], [average,tag2]
if (value >= average) and (tag1 == 1) and (r != 0)
and (r != 9) and (c != 0) and (c != 9)
```

# Apêndice C

# Fluid Computing

O presente apêndice visa apresentar nossa proposta de arquitetura para a *Fluid Computing*. Dessa forma, iremos abordar um novo ambiente para execução de códigos Gamma, baseado no modelo de execução dataflow e que utiliza a *Radnet* como protocolo de comunicação. Aqui, a proposta é utilizar a conversão de um código Gamma para seu equivalente programa em dataflow e submeter tal aplicação a um ambiente de execução baseado em um grafo de fluxo de dados, onde a comunicação entre os elementos de processamento seja realizada através de um protocolo baseado em interesse, a Radnet.

## C.1 Radnet

A Radnet é um protocolo de rede oportunista, baseado em interesses originalmente proposto para MANETs [81] com pouca ou nenhuma infraestrutura. Ao invés de endereçamento ponto a ponto, aplicações enviam mensagens cujos cabeçalhos utilizam prefixos ativos (*AP - Active Prefixes*). Um *AP* possui dois componentes: um prefixo e um nome de interesse, construído pelo nó e pela aplicação local, respectivamente. O prefixo permite roteamento de mensagens probabilísticas, identificação do nó e endereça uma aplicação a um determinado nó. A Figura C.1 apresenta os componentes de um PA.

A Figura C.2 apresenta uma possível comunicação entre dois dispositivos em uma rede de quatro nós utilizando o protocolo Radnet, onde o raio de transmissão *wireless* é delimitado pela circunferência tracejada, cada *AP* com dois campos numéricos e um único interesse registrado na camada de rede. A comunicação inicia pelo envio da mensagem do nó $A$ com prefixo {1;5} e interesse {Football}, ou seja, PA: {1;5;Football}. O nó $B$, dentro da faixa de transmissão de $A$, recebe o pacote de $A$ e encaminha a mensagem de $A$ porque há um prefixo correspondente de $A$ com $B$. O nó $A$ recebe o pacote de retorno encaminhado por $B$, mas detecta que o pacote foi processado anteriormente e o descarta. O nó $C$ recebe e encaminha a mensagem

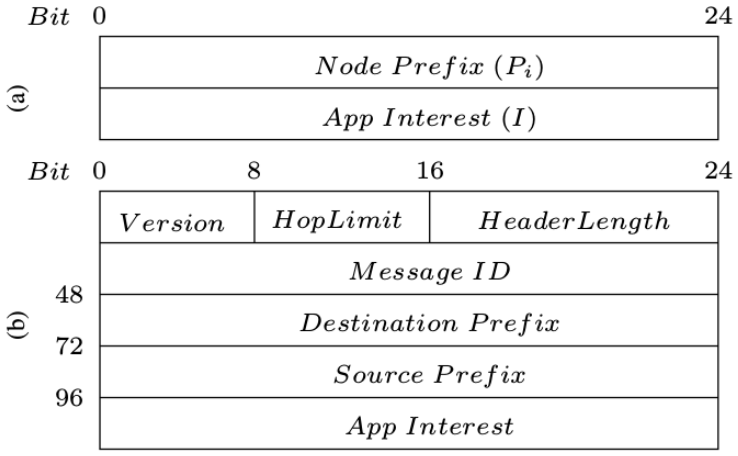


Figura C.1: Mensagem Radnet: (a) Prefixo Ativo e (b) Cabeçalho da Mensagem [5].

de $A$ porque há correspondência de prefixo no primeiro campo (= 1). O nó $D$, ao receber o pacote, detecta que ele tem o mesmo interesse (Futebol) e o repassa para a aplicação e, sem correspondência de prefixo, descarta a mensagem.

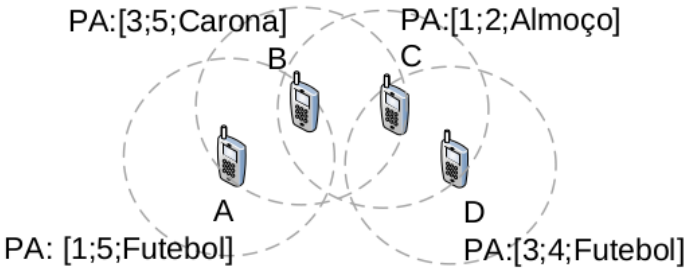


Figura C.2: Exemplo de comunicação Radnet com quatro nós [5].

## C.2 Proposta de Arquitetura

Este estudo, publicado em 2020 [5, 74], propõe um novo ambiente para execução de programas Gamma. A ideia é implementar um ambiente conforme descrito na Figura C.3, onde temos como entrada um código escrito de acordo com a sintaxe Gamma e, ao final do processo, o resultado desta computação será obtido, de maneira compilada. A sintaxe de Gamma utilizada é baseada na implementação de Gamma proposta por Juarez Muylaert [62]. O código Gamma é traduzido pelo *Front End* do ambiente de compilação, gerando sua representação intermediária (IR) que representa um grafo dataflow escrito de acordo com o código *assembly* do *TALM* [82]. Conforme vimos, o *TALM* é um modelo de execução que fornece um

conjunto de instruções necessário para o modelo dataflow, permitindo execução dataflow em máquinas von Neumann. Tal conversão de um código Gamma para seu equivalente grafo dataflow será realizada conforme apresentado em nosso trabalho inicial [23], abordado no Capítulo 3. Com a representação intermediária expressa em *TALM*, o interpretador produz o resultado final da computação.

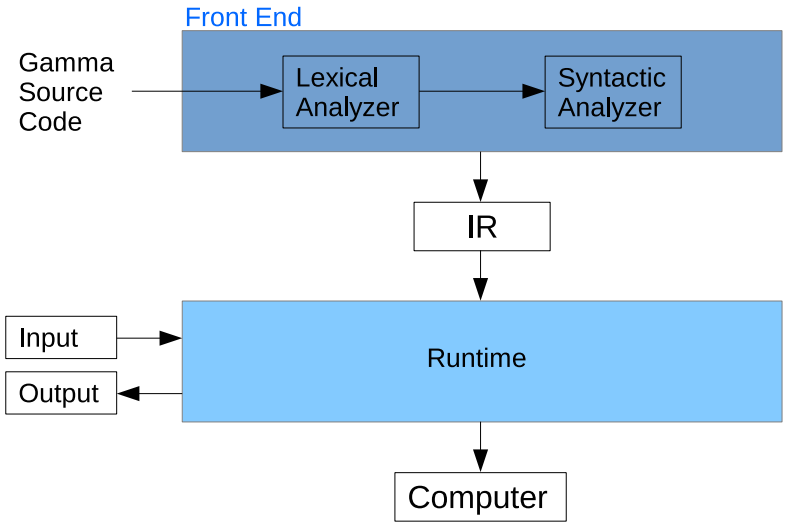


Figura C.3: Procedimento da etapa de interpretação [5].

Para as análises léxica e sintática, que correspondem às tarefas do *Front End*, utilizamos *Lex*, uma ferramenta para construção de analisadores léxicos (*scanners*) e *Yacc* (parser), a partir de uma descrição de uma gramática livre de contexto.

O ambiente de execução proposto é baseado na *SUCURI* [15], uma biblioteca escrita em linguagem Phyton que permite programação dataflow de alto nível. A Figura C.4 apresenta uma breve descrição desta arquitetura. Repare que quando executado em um ambiente de *cluster*, esta estrutura de nós é replicada entre os nós deste *cluster*. Entretanto, somente o nó zero irá possuir a *Ready Queue*, a *Waiting Queue* e a *Matching Unit*. Desta forma, outros nós do *cluster* terão uma versão mais simples do escalonador, utilizada somente para redistribuir tarefas e operandos a seus *Workers*, de acordo com a demanda do escalonador principal. Este escalonador principal é responsável por entregar todos os operandos da *Operand List* a seu nó de destino no grafo. Quando uma correspondência (*match*) ocorre, uma tarefa é criada e colocada na *Ready Queue*. Quando um *worker* solicita uma tarefa ao escalonador, ela é retirada da *Ready Queue* e entregue ao *worker* que a solicitou.

Cada nó desta arquitetura irá executar uma função específica, que irá corresponder à computação expressa por um nó de um grafo dataflow. Observe que, ao converter Gamma para dataflow, cada reação em Gamma irá produzir um ou mais nós (subgrafos) do grafo final equivalente. Assim, podemos afirmar que em nossa representação intermediária (de acordo com o *TALM*), uma reação será responsá-

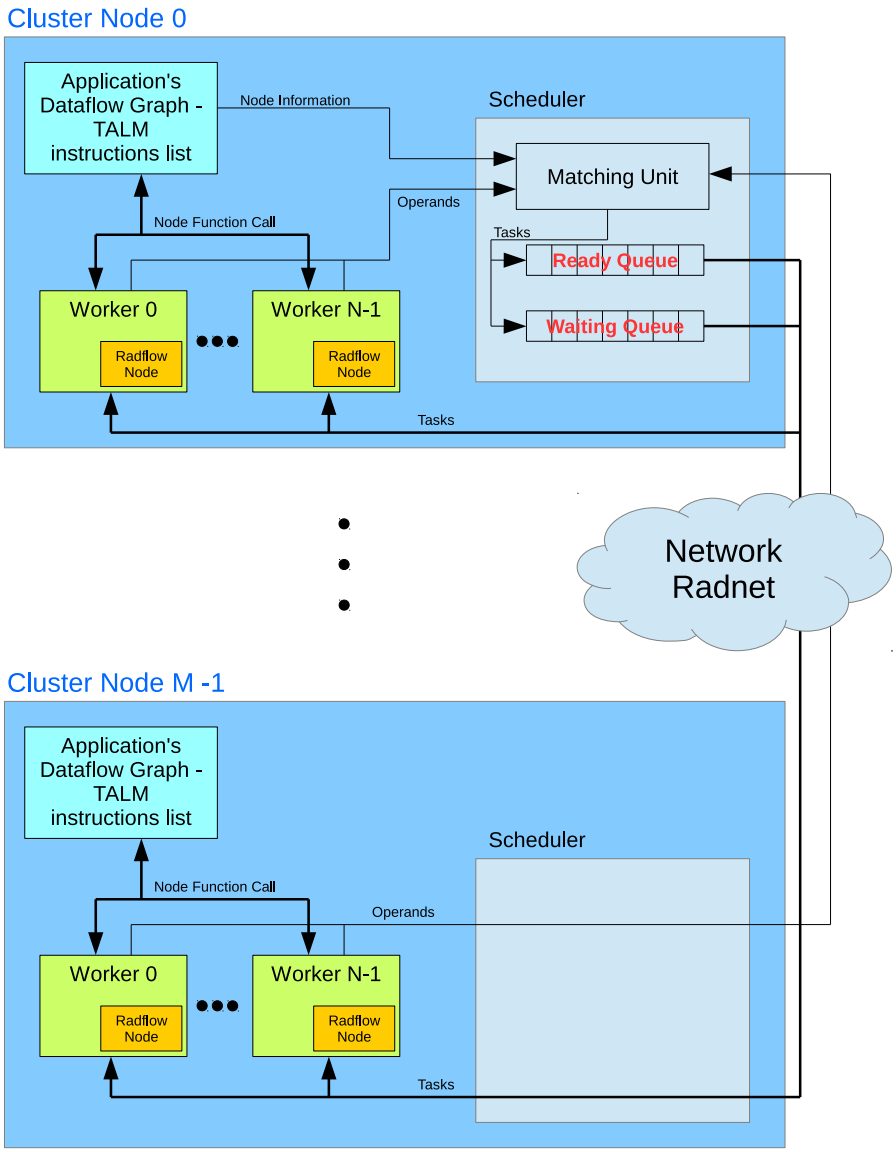


Figura C.4: Arquitetura Proposta [5].

vel por criar um subconjunto de instruções *TALM*. O conjunto de instruções que compõe nossa representação intermediária representa um grafo dataflow. Então, é importante mencionar que cada instrução *TALM* representa um nó e as dependências de dados (arestas do grafo) são dadas através dos operandos de entrada e saída de cada instrução.

Originalmente, a *SUCURI* propõe uma comunicação entre os nós através da utilização do protocolo *MPI*. Porém, nossa proposta visa utilizar o protocolo de comunicação *Radnet*, inserindo recursos de um nó *Radflow* [83]. A utilização da *Radnet* no contexto de execução dataflow tende a tornar mais simples a identificação de interesses, uma vez que cada nó possui interesse nos dados expressos pelas suas arestas de entrada e saída.