UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Fluxo de Dados em *Workflows* Científicos: O Sistema Kepler

São Paulo - SP

Dezembro 2007

UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Fluxo de Dados em *Workflows* Científicos: O Sistema Kepler

Monografia apresentada à Universidade de São Paulo
como parte dos requisitos necessários para a conclusão do
curso de graduação em Bacharelado em Ciência da Computação

Aluno: William Tankiti Yamamoto
Orientador: Prof. Dr. João Eduardo Ferreira

São Paulo - SP
Dezembro - 2007

Organização do texto

O texto está dividido em duas partes principais: técnica e subjetiva. Iniciando a parte técnica, uma breve introdução sobre o assunto central e os fatores motivacionais envolvidos no desenvolvimento do trabalho são oferecidos ao leitor. Descrevem-se também as metas e os objetivos estabelecidos, conjuntamente, pelo aluno e seu orientador. Ainda na área técnica, o software utilizado é explicado e detalhado tanto na teoria quanto na prática, facilitando a compreensão dos exemplos implementados e os resultados obtidos. Para finalizar a primeira parte da monografia, faz-se uma conclusão do trabalho realizado.

Em seguida, descreve-se a parte subjetiva do texto. Nela é feita uma breve análise sobre a experiência do aluno neste trabalho e no curso de Bacharelado em Ciência da Computação.

Sumário

Parte técnica

Capítulo 1

Introdução

Este trabalho tem como tema central *workflows* científicos. Mas o que é *workflow* científico? Para responder a esta pergunta, apresentar-se-ão alguns fundamentos e conceitos básicos a seguir.

1.1 Definição de *Workflow*

De acordo com [6], *workflow* é "a automação do processo de negócio onde documentos, informações ou tarefas são passadas de um participante para outro de acordo com um conjunto de regras de procedimentos para se atingir um objetivo de negócio".

1.2 O que é *Workflow* Científico?

É a aplicação do conceito de *workflow* em ambientes científicos. Diferentemente dos ambientes de negócio, os ambientes científicos visam à análise e manipulação de grande volume de dados.

1.3 *Workflow* Científico vs. *Workflow* de Negócio

Diretrizes de *Workflow* de Negócio (*Business Workflow*)

- Geralmente envolve documentos e atividades;

- Não há transação e processamentos intensivos de dados;
- Vários padrões e ferramentas disponíveis: *Workflow Management Coalition* (WfMC), *Web Services Flow Language* (WSFL), *Business Process Execution Language for Web Services* (BPEL4WS), *XML Process Definition Language* (XPDL) etc.;
- Controle de fluxo complexo.

Já em *workflows* científicos:

- Análise de grande volume de dados;
- Fluxo de dados.

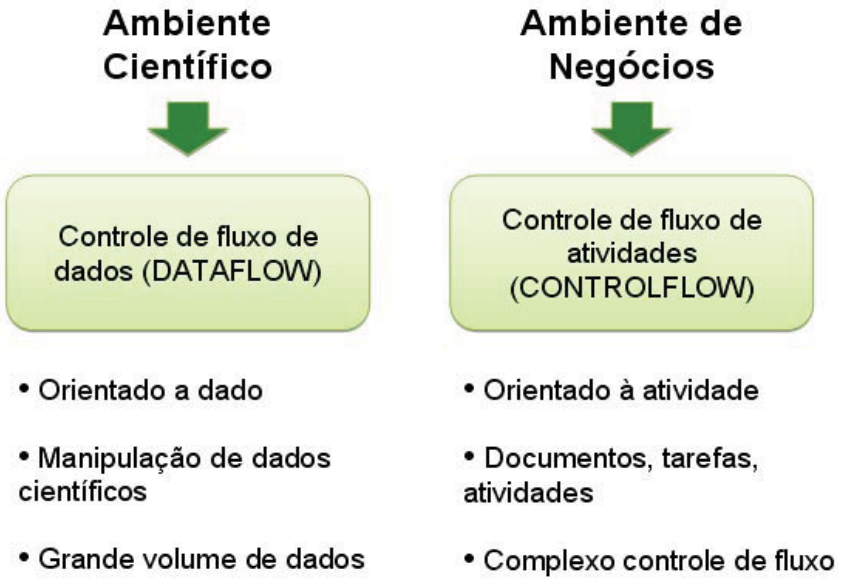

Figura 1.1: Workflow Cientifico vs. Workflow de Negócio

1.4 Motivação

Um dos possíveis desafios atrelados à área de pesquisa de gerenciamento de processos de negócios (relacionados ao *business workflow*) e das aplicações de *workflows* científicos é combinar o controle de fluxo de dados com o fluxo de atividades em um único sistema, independentemente do ambiente (negócio ou científico).

Atualmente, os sistemas de gerenciamento de *workflows* existentes têm uma abordagem prioritária, ou seja, ou priorizam o controle de fluxo de atividades ou o controle de fluxo de atividades. Contudo, possivelmente poderá haver situações em que será necessário adotar as duas abordagens simultaneamente. Desta maneira, entender o funcionamento do fluxo de dados no *workflow* científico é fundamental para uma futura integração.

1.5 Objetivo

Este trabalho tem como objetivo estudar o fluxo de dados no **Kepler**, uma ferramenta específica para *workflows* científicos. Tal estudo baseia-se na leitura de artigos e dissertações de mestrado e doutorado relacionados direta e indiretamente aos *workflows* científicos e ao Kepler. Além disso, foi de fundamental importância a leitura da documentação disponível sobre a ferramenta para entender seu funcionamento tanto teórico quanto prático.

A fim de obter resultados mais precisos e conclusivos, foi escolhido um caso prático para ser implementado na ferramenta Kepler.

Capítulo 2

Fundamentos

2.1 Sobre o Kepler

É um ambiente de desenvolvimento específico para análise, manipulação, modelagem e simulação de *workflows* científicos. O sistema Kepler fornece aos usuários uma ferramenta de fácil uso e que simplifica a criação e a execução de *workflows* científicos ou experimentos computacionais complexos.

Teoricamente, desenvolver um sistema de *workflow* científico no Kepler[1] é bastante simples. Basta o usuário arrastar e colocar os componentes na área de desenvolvimento, conectando-os de forma a obter o fluxo de dados desejado.

Atualmente, o Kepler é utilizado em áreas como a bioinformática, ecoinformática e geoinformática. Modelado para utilização em sistemas distribuídos complexos, Kepler oferece uma solução na qual um cientista pode compartilhar e utilizar dados (ou *workflows*) com outros cientistas localizados em outras partes mundo.

O Kepler foi desenvolvido com base nos princípios da orientação a atores. Sustentado pelo modelo proposto por Carl Hewitt [4] e posteriormente estendido e formalizado por Gul Agha [5], Kepler utiliza a metáfora diretor/ator para representar os componentes do *workflow* e a comunicação entre eles. Assim como em um filme, o diretor do Kepler comanda o seu elenco (atores), especificando quando cada um deve agir e como estão conectados entre si. Já o papel dos atores é processar os dados disponíveis nas suas

[1] para conhecer mais sobre o Kepler visite o sítio *http://www.kepler-project.org*. A versão utilizada neste trabalho foi *beta3*.

portas de entrada e disponibilizar os resultados na porta de saída.

Os principais componentes do Kepler são:

- Diretor: componente que controla a execução do *workflow*, gerenciando outros componentes (atores, portas etc.). Define o modelo computacional a ser utilizado (síncrono, paralelo, distribuído etc.), sendo obrigatória a sua presença.
- Ator: componente do *workflow* que representa um dado ou serviço. Podem ser conectados com outros atores por meio de portas e ter parâmetros configuráveis.
- Porta: cada ator pode conter uma ou mais portas, utilizadas no consumo e na produção de dados assim como na comunicação com outros atores envolvidos no *workflow*. Atores são conectados entre si por meio de portas. A conexão que representa o fluxo de dados entre um ator e outro é chamada de canal. As portas podem ser divididas em três tipos diferentes:

  1. Entrada: usada para consumo de dados;
  2. Saída: destino dos dados produzidos pelo ator;
  3. Entrada/Saída: para consumo e saída de dados.

  Além disso, as portas podem ser configuradas como simples ou múltiplas (multiportas). Uma porta de entrada simples pode estar conectada a um único canal. Já uma porta de entrada múltipla pode estar conectada a vários canais simultaneamente.
- Relação: permitem replicar fluxos de dados. Assim, o mesmo dado pode ser mandado para vários lugares do *workflow*.
- Parâmetro: são valores configuráveis que podem fazer parte de um *workflow*, diretor ou ator.

2.2 Diretores

Entidade que controla a execução do *workflow*, gerenciando atores, portas, parâmetros etc.. Uma de suas principais tarefas é determinar quando os atores devem ser executados.

Além disso, fica sob responsabilidade do diretor saber como os atores estão conectados e determinar o modelo de transporte de dados de um ator para outro.

O Kepler disponibiliza quatro tipos de diferentes de diretor. São eles:

- *Synchronous Dataflow* (SDF): modelo computacional que determina uma ordem de acionamento dos atores, garantindo uma seqüência pré-determinada de operações. O controle de fluxo de dados é completamente previsível em tempo de compilação. Este diretor será detalhado na Seção 2.2.1 uma vez que o mesmo será utilizado na implementação do exemplo (Capítulo 4).

- *Process Network* (PN): modelo usado para representar processamento paralelo em sistemas distribuídos. Diferentemente do SDF, o diretor PN não determina uma ordem de execução dos atores. Os atores passam a agir de forma independente, sendo acionados apenas quando há dados suficientes em suas portas de entrada. Este diretor será detalhado na Seção 2.2.2 uma vez que o mesmo será utilizado na implementação do exemplo (Capítulo 4).

- *Continuous Time* (CT): o modelo proposto pelo diretor CT tem como objetivo simular sistemas que podem ser definidos por meio de equações diferenciais ordinárias. É usado, geralmente, em modelos de circuitos analógicos, sistemas dinâmicos e físicos.

- *Discrete Event* (DE): este diretor define um modelo baseado no tempo de simulação. O fluxo de dados é desconhecido em tempo de compilação. É utilizado em sistemas digitais e redes de comunicação.

2.2.1 *Synchronous Dataflow* (SDF)

A característica fundamental que difere o diretor SDF dos demais diretores é a organização. Sob direção desse diretor, os atores são organizados em fila de execução. Na fase de pré-inicialização dos atores, o diretor SDF delega à classe *BaseSDFScheduler* a responsabilidade de formar a referente fila.

Isso pode ser observado no trecho de código abaixo, onde é instanciado um objeto da classe *BaseSDFScheduler* e executado o método *getSchedule()* para seqüenciar os disparos de atores.

```
public void preinitialize() throws IllegalActionException {
    super.preinitialize();

    BaseSDFScheduler scheduler = (BaseSDFScheduler) getScheduler();

    if (scheduler == null) {
        throw new IllegalActionException("Attempted to initialize "
                + "SDF system with no scheduler");
    }

    // force the schedule to be computed.
    if (_debugging) {
        _debug("Computing schedule");
    }

    try {
        scheduler.getSchedule();
    } catch (Exception ex) {
        throw new IllegalActionException(this, ex,
                "Failed to compute schedule:");
    }

    // Declare the dependencies of rate parameters of external
    // ports.  Note that this must occur after scheduling, since
    // rate parameters are assumed to exist.
    scheduler.declareRateDependency();
}
```

Figura 2.1: Método *preinitialize()* do diretor SDF

O escalonador[2] ordena os atores de maneira que cada um deles é acionado apenas uma única vez quando suas portas de entrada tiverem dados suficientes para iniciar o processamento. Para exemplificar, considere o seguinte *workflow* (figura 2.2).

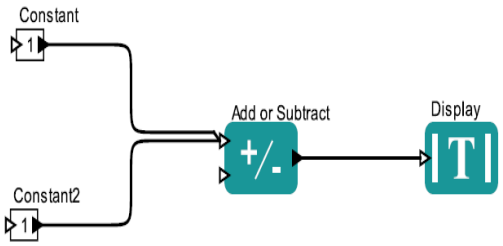

Figura 2.2: Exemplo de adição de dois números

Composto pelos atores `Constant`, `Constant2`, `Add or Subtract` e `Display`, o exemplo acima soma os valores dados em `Constant` e `Constant2` e exibe o resultado em uma janela. Nesse caso, o escalonador ordena os atores na seguinte seqüência de disparo:

1. `Constant2`

[2]tradução dada para a palavra *scheduler*

2. `Constant`

3. `Add or Subtract`

4. `Display`

```
..SDF Director Preinitializing ...
Invoking preinitialize():  ..Add or Subtract
Invoking preinitialize():  ..Constant
Invoking preinitialize():  ..Constant2
Invoking preinitialize():  ..Display
..SDF Director Finished preinitialize().
Computing schedule
Normalized Firing Counts:
{ptolemy.actor.lib.gui.Display {..Display}=1, ptolemy.actor.lib.Const
{..Constant}=1, ptolemy.actor.lib.AddSubtract {..Add or Subtract}=1,
ptolemy.actor.lib.Const {..Constant2}=1}
Schedule is:
Execute Schedule{
Fire Actor ptolemy.actor.lib.Const {..Constant2}
Fire Actor ptolemy.actor.lib.Const {..Constant}
Fire Actor ptolemy.actor.lib.AddSubtract {..Add or Subtract}
Fire Actor ptolemy.actor.lib.gui.Display {..Display}
}
Adding firingsPerIteration parameter to Display with value 1
Adding firingsPerIteration parameter to Constant with value 1
Adding firingsPerIteration parameter to Add or Subtract with value 1
Adding firingsPerIteration parameter to Constant2 with value 1
Called initialize().
Invoking initialize():  ..Add or Subtract
Initializing actor: ..Add or Subtract.
Invoking initialize():  ..Constant
Initializing actor: ..Constant.
Invoking initialize():  ..Constant2
Initializing actor: ..Constant2.
Invoking initialize():  ..Display
Initializing actor: ..Display.
Director: Called prefire().
Director prefire returns true.
The actor ..Constant2 will be iterated.
The actor ..Constant2 was iterated.
The actor ..Constant will be iterated.
The actor ..Constant was iterated.
The actor ..Add or Subtract will be iterated.
The actor ..Add or Subtract was iterated.
The actor ..Display will be iterated.
The actor ..Display was iterated.
Director: Called wrapup().
```

Figura 2.3: Algumas ações tomadas pelo diretor SDF

Os diretores SDF são eficientes, porém estáticos. Isso significa que em cada iteração os números de *tokens* produzidos e consumidos serão iguais. O modelo de computação que este diretor oferece é bastante estável e previsível.

2.2.2 *Process Network* (PN)

Sua principal característica é o acionamento dos atores através da disponibilidade de dados nas suas portas de entrada.

Basicamente, cada ator do sistema é um processo independente. O diretor PN "cria" uma *thread* para cada ator, permitindo que vários processos sejam executados concorrentemente. O modelo de computação proporcionado pelo diretor PN é uma ótima solução para processamento paralelo em sistemas distribuídos.

Entretanto, este modelo computacional pode causar resultados inesperados como *overflow* (ocorre quando um ator processa *tokens* a uma taxa muito superior em relação a outro).

2.3 Atores

Basicamente, atores são definidos como entidades que processam dados presentes nas suas portas de entrada ou que criam e enviam dados para outras entidades por meio de suas portas de saída.

Entretanto, atores somente têm conhecimento da disponibilidade de dados nas suas portas de entrada e como processar estes dados e mandá-los para suas portas de saída. A tarefa de transportar dados é responsabilidade das portas, já que um ator não sabe com quais atores ele está conectado.

Os atores podem ser divididos em dois tipos: *AtomicActor* e *CompositeActor*. Atores que pertencem à classe *AtomicActor* representam entidades simples de processamento de dados enquanto que atores do tipo *CompositeActor* são entidades compostas por vários outros atores.

A seguir, detalhar-se-ão alguns atores que foram utilizados no trabalho desenvolvido.

- *Add or Subtract*: ator que soma ou subtrai números que chegam às suas portas de entrada;

Figura 2.4: Ator *Add or Subtract*

- *Boolean Switch*: produz *tokens* a partir dos dados obtidos pela porta de entrada *input*, disponibilizando-os nas portas de saída *trueOutput* ou *falseOutput* de acordo com a porta de entrada *control*;

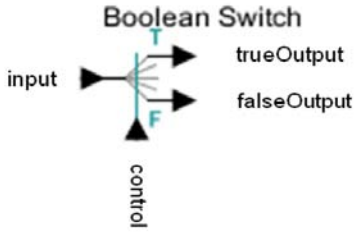

Figura 2.5: Ator *Boolean Switch*

- *CompositeActor*: é uma entidade composta por vários atores. Objetos desta classe são extensões de *CompositeEntity*. Por meio da utilização deste ator o Kepler possibilita combinar modelos computacionais diferentes em um mesmo *workflow*. Isso porque *CompositeActor* pode ou não ter um diretor local responsável pelos atores que o compõem. O modelo computacional implementado pelo diretor local não precisa ser, necessariamente, o mesmo modelo de computacional oferecido pelo diretor global;

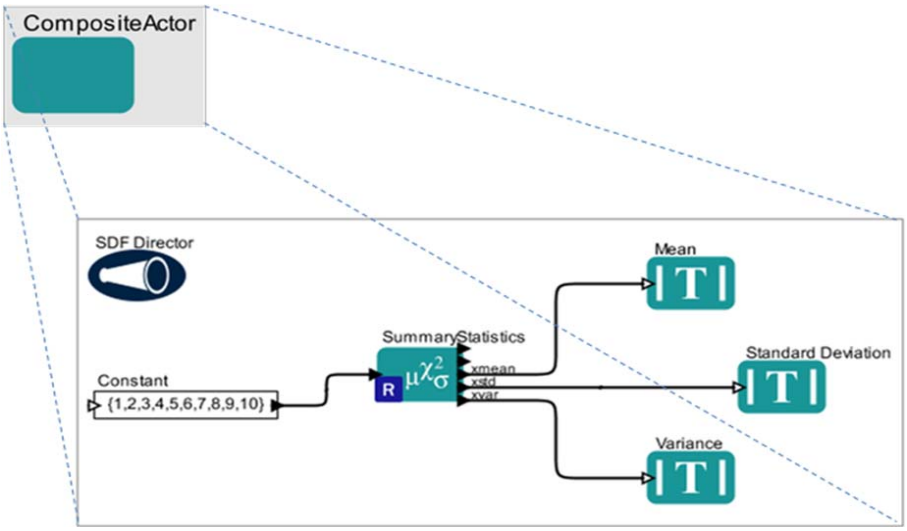

Figura 2.6: Ator *CompositeActor*

- *Constant*: o *token* de saída deste ator é determinado pelo valor do parâmetro. A porta de entrada funciona como disparador. Assim que algum dado chegar por esta porta o ator envia o valor do parâmetro para sua porta de saída;

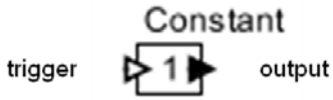

Figura 2.7: Ator *Constant*

- *Display*: mostra em uma tela os valores dos *tokens* que chegam à porta de entrada *input*;

Figura 2.8: Ator *Display*

- *Variable Setter*: sua função é alterar o valor de uma variável ou parâmetro. O valor de atribuição é equivalente ao dado obtido pela porta de entrada *input*.

Figura 2.9: Ator *Variable Setter*

2.4 Dados

Por padrão, o sistema aceita tipos de dados mais comumente utilizados como:

- Números: *int*, *float*, *double*, números complexos;
- *Strings*;
- Estruturas de dados: *arrays*, matrizes, tabelas de bancos de dados.

Além desses tipos comuns, Kepler permite que tipos de dados sejam criados pelo usuário. De fato, isto é possível graças ao tratamento no qual os dados recebem. Todo dado é encapsulado por um objeto da classe *Token*, possibilitando que sejam manipulados de maneira uniforme independentemente do seu tipo. Há um contêiner básico de dado: *token*.

Os *tokens* são transportados de um ator para outro por meio dos conectores.

Capítulo 3

Fluxo de dados no Kepler

Basicamente, a transmissão de dados de um ator a outro pode ser representada pela figura abaixo:

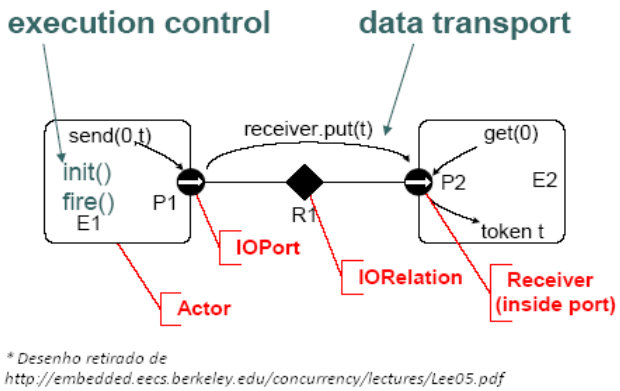

Figura 3.1: Transmissão de dados de um ator a outro

No caso, *t* é o dado a ser transmitido do ator *E1* (à esquerda) ao ator *E2* (à direita). Em certo instante, o diretor executa o método *fire()* implementado pelo ator *E1* que, por sua vez, processa o dado e disponibiliza o resultado obtido na porta *P1* (por meio da chamada do método *send()* da porta *P1*).

O método *send()* da classe *TypedIOPort* verifica se o *token* a ser enviado é compatível com os tipos de dados aceitos pela porta em questão (e convertendo os dados, se necessário).

```
public void send(int channelIndex, Token token)
        throws IllegalActionException, NoRoomException {
    _checkType(token);
    super.send(channelIndex, token);
}
```

Note pelo trecho de código acima que o método *send()* de *TypedIOPort* chama o método *send()* de sua superclasse. No caso, a superclasse de *TypedIOPort* é *IOPort* (ver o diagrama hierárquico representado pela figura 3.2).

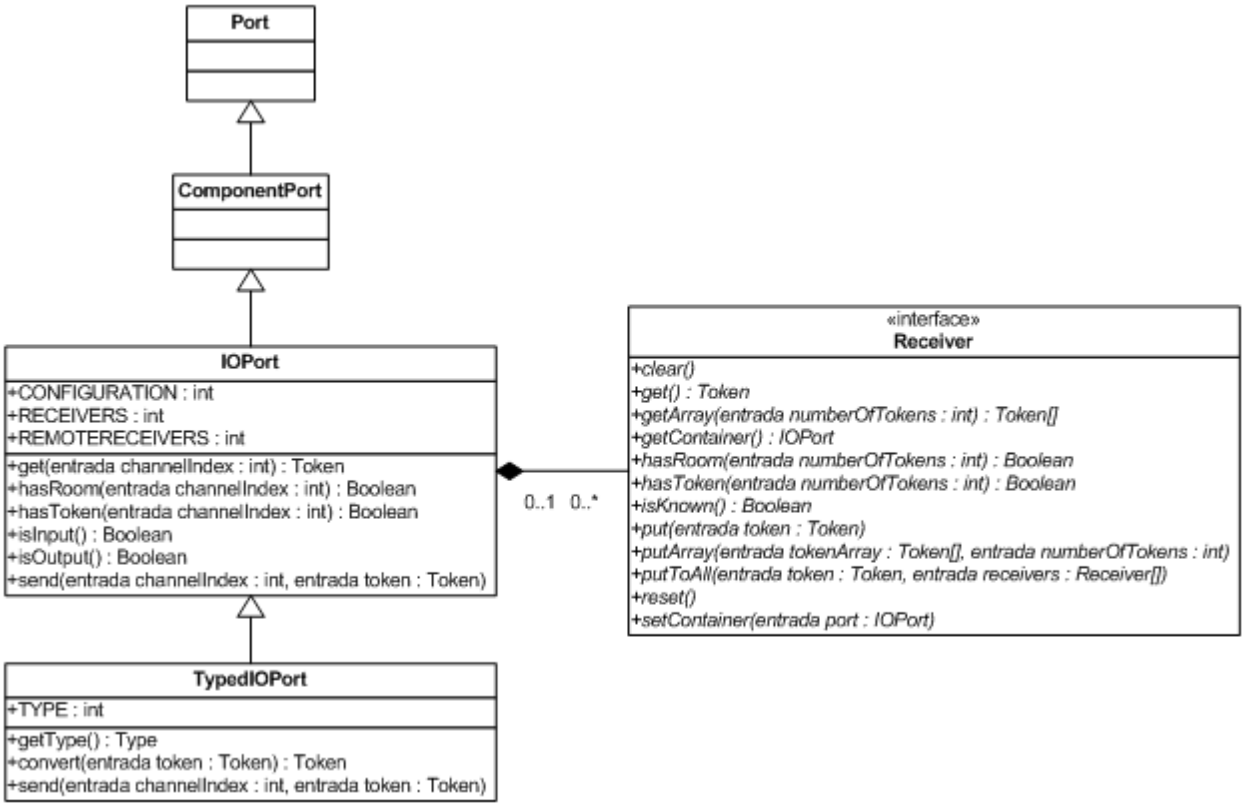

Figura 3.2: Hierarquia de classes do componente porta

O método *send()* de *IOPort* (trecho de código abaixo) chama o método *getRemoteReceivers()* da classe *Receiver* para obter a referência de todos os receptores que estão conectados à porta (detalhes da classe *Receiver* serão dados mais adiante). Assim, a instância da porta executa o método *putToAll()* da classe *Receiver*, delegando para esta última a ação de transportar o *token t* para os devidos receptores (obtidos por *getRemoteReceivers()* que pela figura pertencem à porta *P2*). O ator *E2* chama o método *get()*

da porta *P2* implementado na classe *IOPort* para que o *token t* possa ser processado.

```
public void send(int channelIndex, Token token)
        throws IllegalActionException, NoRoomException {
    ...
    ...
    Receiver[][] farReceivers;
    ...

    try {
        ...
        farReceivers = getRemoteReceivers();

        if ((farReceivers == null) || (farReceivers.length <= channelIndex)
                || (farReceivers[channelIndex] == null)) {
            return;
        }
    } finally {
        _workspace.doneReading();
    }

    if (farReceivers[channelIndex].length > 0) {
        // Delegate to the receiver to handle putting to all
        // receivers, since domain-specific techniques might be relevant.
        farReceivers[channelIndex][0].putToAll(token,
                farReceivers[channelIndex]);
    }
}
```

Figura 3.3: Método *send()* da classe *IOPort*

Observe na figura 3.1 que é colocado, além do *token t*, zero como argumento do método *send()*. Esse número representa o índice do canal pelo qual o *token t* será transmitido. O mesmo acontece no método *get()* executado pelo ator *E2*, que recebe zero como argumento, indicando que o *token* estará disponível no canal zero. A figura a seguir exemplifica a porta *P1* transmitindo o *token t* por dois canais diferentes (0 e 1). Portas que transmitem ou recebem dados em mais de um canal são chamadas de multiportas.

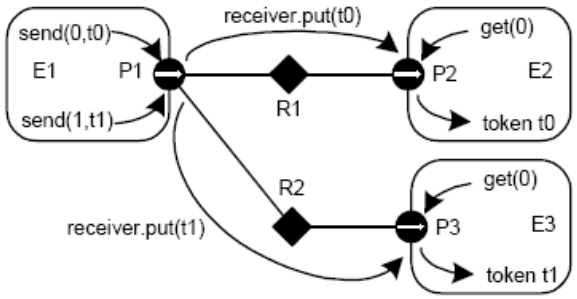

Figura 3.4: Transporte de dados por vários canais

No Kepler, as multiportas são representadas por triângulos brancos (veja a figura 3.5).

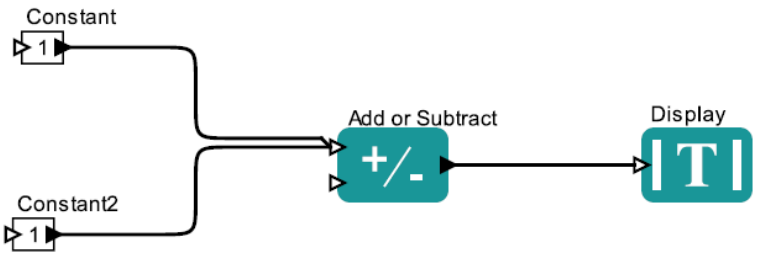

Figura 3.5: Os triângulos brancos representam as multiportas

Observe pelo diagrama 3.2 que objetos da classe *IOPort* podem conter um ou mais objetos da classe *Receiver*. Estes últimos são chamados de receptores[1]. Os receptores são os componentes do sistema responsáveis pelo armazenamento e distribuição de *tokens*. Assim, métodos como *get()*, *put()* e *getRemoteReceivers()* estão implementados na classe *Receiver* para que sejam realizadas tais funções. Portanto, o transporte de dados de um ator para o outro é feita por meio dos receptores, enquanto que as portas são responsáveis pelo encapsulamento dos diversos receptores nelas contidas e pela comunicação entre estes últimos e os atores.

Além disso, dado o papel fundamental dos receptores no fluxo de dados do Kepler, é importante dizer como são instanciados os objetos da classe *Receiver*. Pode-se perceber que o modelo de computação do *workflow* especificado pelo diretor está diretamente relacionado ao comportamento e ao modo de transmissão de dados dos receptores. Desta maneira, o diretor é responsável pela criação e definição do tipo dos receptores.

Veja na figura 3.6 uma representação da arquitetura que envolve parte do processo de fluxo de dados. Inicialmente, o diretor do *workflow* cria receptores compatíveis com seu modelo de computação. Estes receptores são criados dentro das portas que, por sua vez, são componentes agregados aos atores.

[1]tradução de *receivers*

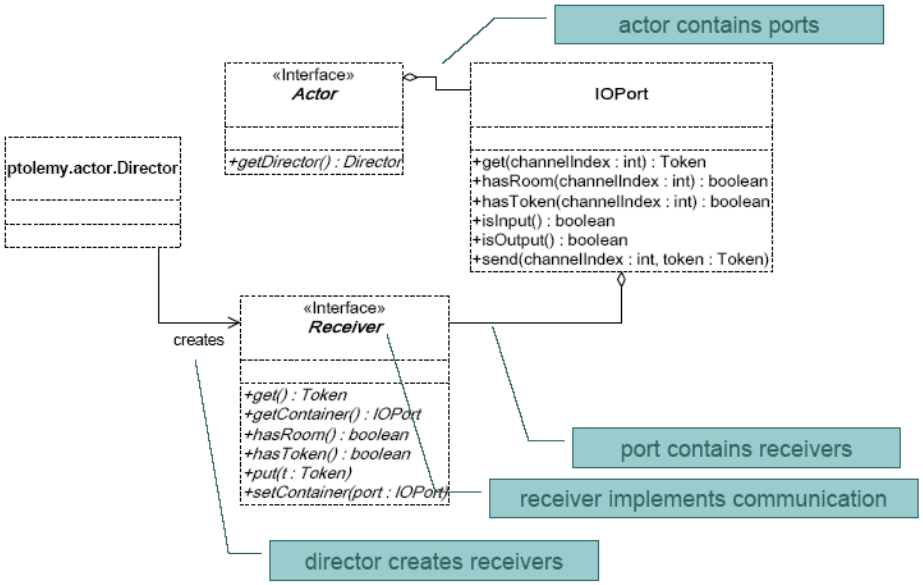

Figura 3.6: Esquema geral dos principais componentes

Capítulo 4

Caso prático implementado

4.1 Sobre a escolha do caso

O caso prático implementado foi extraído de [7]. Sua escolha não foi feita por acaso. Apesar de este trabalho priorizar o fluxo de dados, percebeu-se a necessidade de um caso que apresentasse também características de controle de fluxo de atividades.

O caso escolhido será descrito na próxima seção. Trata-se de um exemplo de *workflow* de negócio com controle de atividades e de fluxo e dados e em [7] foi modelado em *Redes de Petri Coloridas* [3].

Por meio da implementação deste caso prático, pretende-se analisar o comportamento do Kepler no controle de fluxo de dados e verificar os limites e restrições da ferramenta no tratamento do fluxo de atividades.

4.2 Controle de fluxo de empresas

Breve descrição do problema: duas empresas, A e B, estão localizadas em cidades diferentes. A empresa A produz e envia enormes engradados de um dado produto à empresa B, um a um. Esses engradados enviados pela empresa A são armazenados em um galpão (estoque) pertencente à empresa B até serem retirados para processamento (por exemplo, distribuição, venda ou consumo interno dos engradados).

Entretanto, a empresa B tem um problema: o galpão é capaz de armazenar apenas

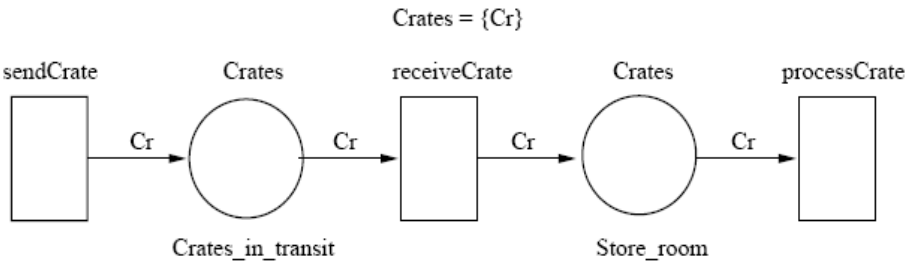

Figura 4.1: Produção e processamento de engradados

até certo número máximo (MAX) de engradados. Assim, para evitar o aluguel de outro galpão para estoque ou que engradados sejam deixados na rua, B estabeleceu um acordo de protocolo de controle de fluxo com A.

Para implementar tal protocolo, a empresa A mantém um registro do número de engradados a enviar enquanto B concentra-se no número de vagas disponíveis para armazenar novos engradados no galpão. Caso existam vagas disponíveis no estoque, a empresa B pode comunicar tal número de vagas à empresa A enviando-lhe uma carta contendo a referida informação. O número de vagas é, então, atualizado para zero. Ao receber uma carta, a empresa A atualiza o número de engradados a enviar somando-o com o número contido no item recebido. Ao enviar um engradado para a empresa B, A diminui o número de engradados a enviar em 1. Quando um engradado é processado em B, o número de vagas disponíveis no galpão é acrescido de 1.

Inicialmente, tem-se a seguinte situação: nenhum engradado ou carta está em trânsito; o galpão de armazenamento da empresa B está vazio, ou seja, o número de vagas vazias é equivalente à capacidade máxima de estocagem (MAX); o número de engradados a enviar pela empresa A é igual a zero.

A figura 4.2 representa uma modelagem do sistema descrito anteriormente.

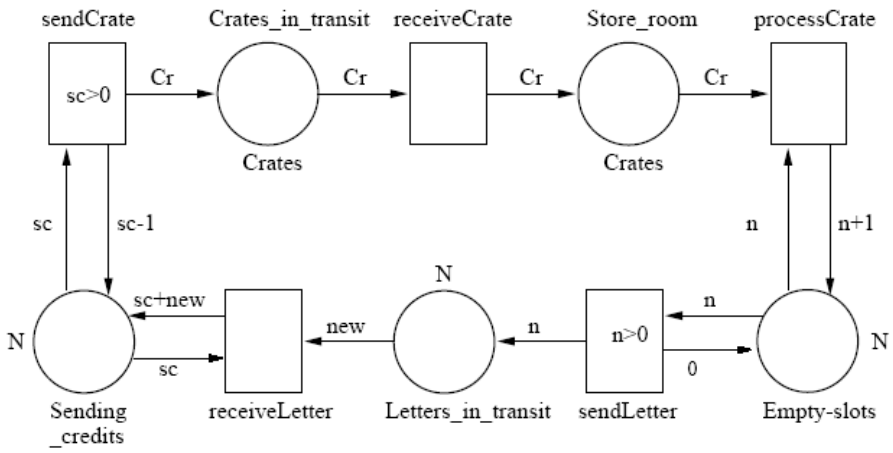

Figura 4.2: Representação do protocolo de controle de fluxo

4.3 Implementação no Kepler

Para implementar o exemplo, foram utilizados 4 atores compostos (*EnviaCartas*, *ProcessaEngradado*, *ProduzEngradado* e *VagasVazias*) que representam as atividades mais importantes do processo. Cada um desses *subworkflows* possui outro ator composto (do tipo *CompositeActor*), que são coordenados por diretores SDF. A escolha dos diretores locais SDF se deu por conta da necessidade de sincronização nesses trechos do modelo.

Foram utilizados 4 *subworkflows* (*AcionaEnvioDeCartas*, *AcionaProcessamento*, *AcionaProdução* e *AcionaWorkflow*) representando os eventos externos que influenciam no fluxo entre as empresas. Nesses casos o diretor utilizado é o diretor Global PN, uma vez que este permite autonomia de execução dos diversos módulos (*subworkflows*).

A solução adotada optou por usar variáveis e parâmetros globais para compartilhar informações entre os atores principais do modelo em vez do uso de relações entre portas desses atores. Isso se deve a um comportamento característico do modelo de atores: o ator só executa sua função se houver dados na porta de entrada.

O impacto dessa premissa no modelo aconteceria da seguinte forma: suponhamos que exista uma relação entre uma porta de saída do ator *EnviaCartas* e uma porta de entrada do ator *ProduzEngradado*.

A informação *Cartas* gerada pelo ator *EnviaCartas* chegaria a entrada do *ProduzEngradado* através de um *token*. A existência de 1 *token* na entrada de um ator possibilita

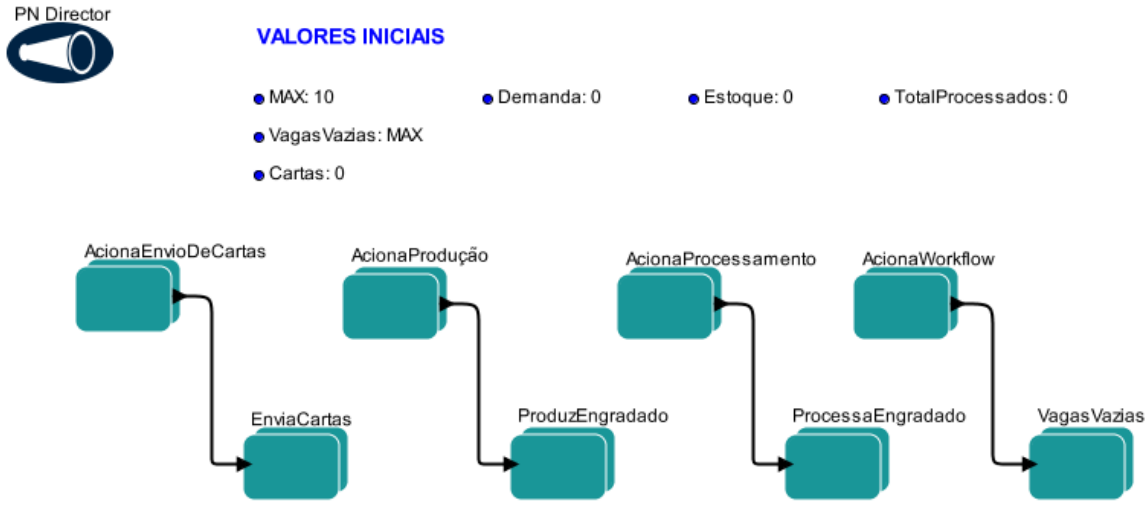

Figura 4.3: Implementação do caso dos engradados no Kepler

sua execução uma única vez, consumindo esse *token*.

Porém, são necessárias várias execuções de *ProduzEngradado* para efetivamente se produzir e enviar os engradados para a Empresa B. Como um ator poderia realizar n execuções (*fire()*) se recebeu apenas 1 *token* em sua porta de entrada?

Percebendo essa característica da ferramenta, optou-se pelo uso de variáveis e parâmetros globais para compartilhar dados e tornar a execução de cada módulo mais independente e a representação mais próxima do problema apresentado inicialmente.

A seguir, descreve-se cada um dos componentes do *workflow* implementado.

Parâmetros e seus respectivos valores iniciais:

- *MAX*: capacidade máxima de estocagem do galpão de armazenamento de engradados;
- *VagasVazias*: inicialmente seu valor é equivalente ao valor da variável *MAX*;
- *Cartas*: nenhuma carta em trânsito no momento inicial;
- *Demanda*: o número de engradados a enviar pela empresa A é, inicialmente, igual a zero;
- *Estoque*: registro corrente do número de engradados no galpão;
- *TotalProcessados*: computa o número total de engradados processados por B;

Atores principais do *workflow*:

- *VagasVazias*

  Representação abstrata do galpão de armazenamento dos engradados. O ator *AcionaWorkflow* determina quando o ator *VagasVazias* deve executar. Já o ator *Constant3* (ver figura abaixo) verifica se o número de vagas disponíveis no estoque é diferente de zero. Considerando o comportamento do ator *Boolean Switch*, caso haja dados disponíveis na porta de entrada *port* e se o valor de *VagasVazias* for diferente de zero, a porta de saída mais acima de *Boolean Switch* disponibilizará o dado booleano *true* para o ator composto *VagasVaziasCartas*. Com isso, este último é acionado e passa a atualizar as variáveis *Cartas* e *VagasVazias* de modo sincronizado (já que estão sob a direção do diretor SDF).

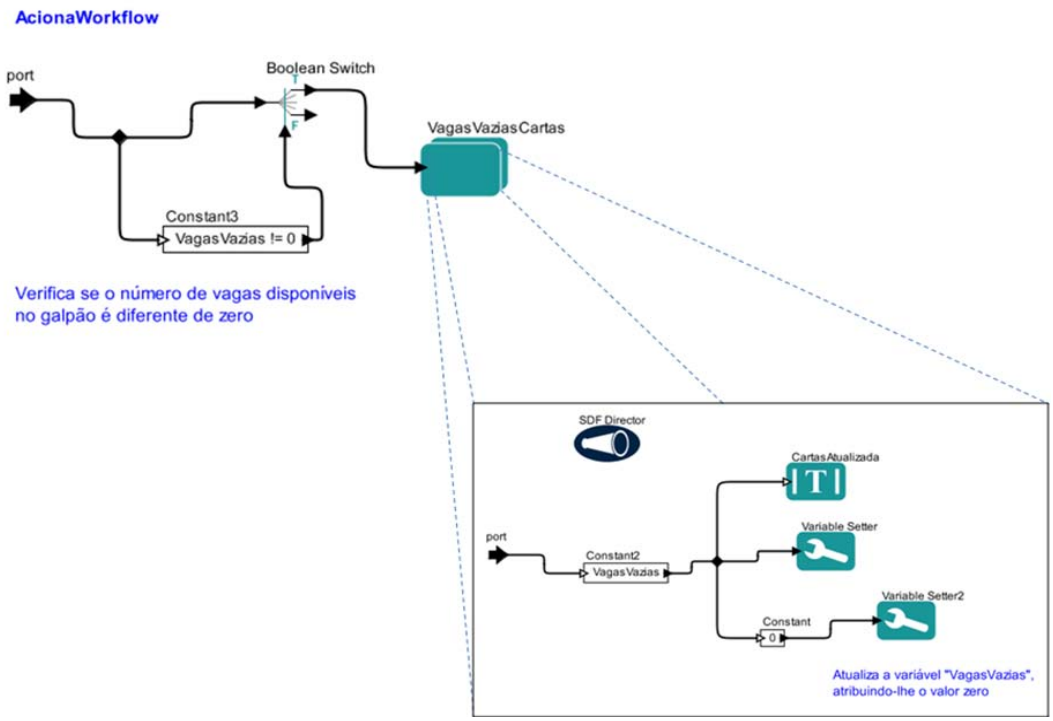

Figura 4.4: Ator *VagasVazias* no Kepler

- *EnviaCartas*

  Ator composto (do tipo *CompositeActor*) que representa o envio de cartas da empresa B para A. Tem uma porta de entrada responsável pelo acionamento do ator em questão. Quando *EnviaCartas* é acionado o ator *Constant3* entra em ação, ana-

lisando a variável *Cartas*. Caso a expressão `Cartas > 0 && Cartas <= MAX` seja verdadeira, então o ator composto *CartasDemanda* entra em execução. Este último é responsável por atualizar as variáveis *Cartas* (atribuindo o valor zero) e *Demanda* (incrementando em uma unidade) de forma sincronizada, ou seja, garantindo que ambas variáveis sejam modificadas conjuntamente e em apenas uma iteração do diretor global PN.

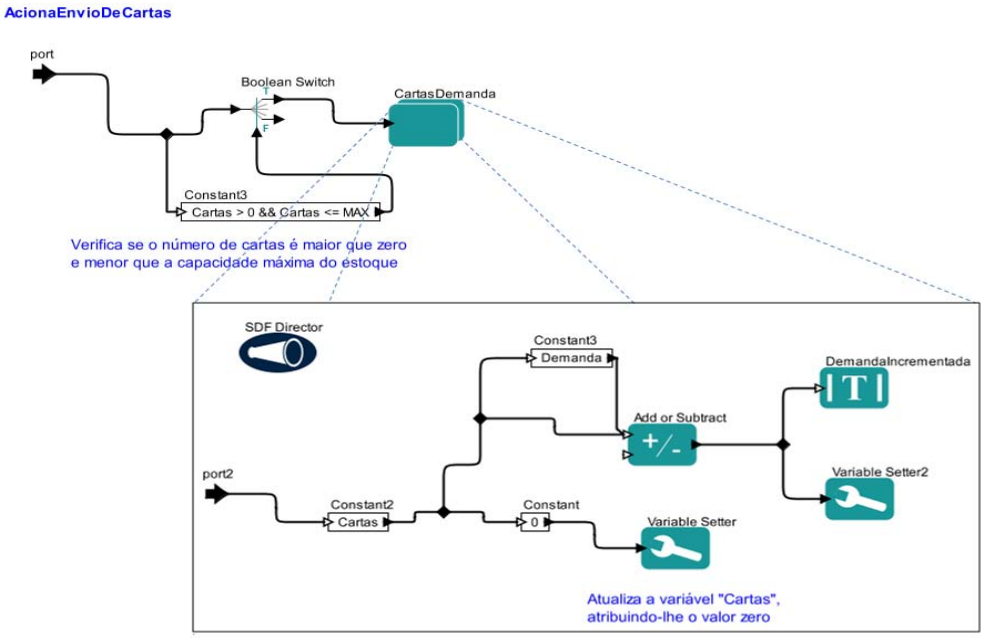

Figura 4.5: Ator *EnviaCartas* implementado no Kepler

- *ProduzEngradado*

  Representação da produção e envio de engradados da empresa *A*. A execução deste *subworkflow* está vinculada aos disparos feitos pelo *trigger AcionaProdução*. Os engradados são produzidos somente quando *AcionaProdução* dispara algum sinal para a porta de entrada *port* de *ProduzEngradado*. Quando o *subworkflow* é acionado, verifica se há demanda. Caso exista, o engradado é produzido. É importante notar que somente 1 engradado é produzido em cada disparo do *trigger*. A representação da produção do engradado consiste apenas no decréscimo do valor da variável *Demanda* e sua imediata atualização (operação realizada no ator composto *DemandaEstoque*).

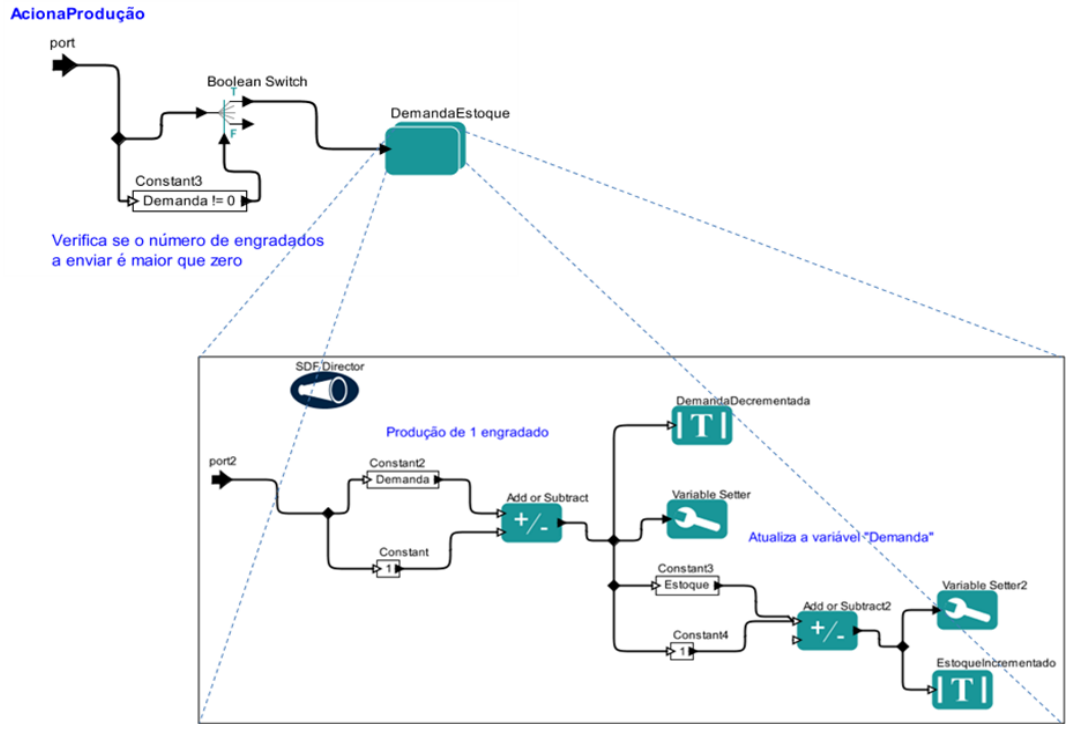

Figura 4.6: Ator *ProduzEngradado*

- *ProcessaEngradado*

  Módulo responsável pelo processamento dos engradados feito pela empresa *B*. Como nos atores descritos anteriormente, o ator *ProcessaEngradado* é acionado por um evento externo, representado por *AcionaProcessamento*. Assim que entra em execução, *ProcessaEngradado* verifica a existência de engradados estocados galpão. Caso exista, o ator *Boolean Switch* manda um *token* para *EstoqueVagasVazias* para que este seja acionado. O ator *EstoqueVagasVazias*, por sua vez, decrementa a variável *Estoque* em 1 e incrementa com mesmo valor a variável *VagasVazias*.

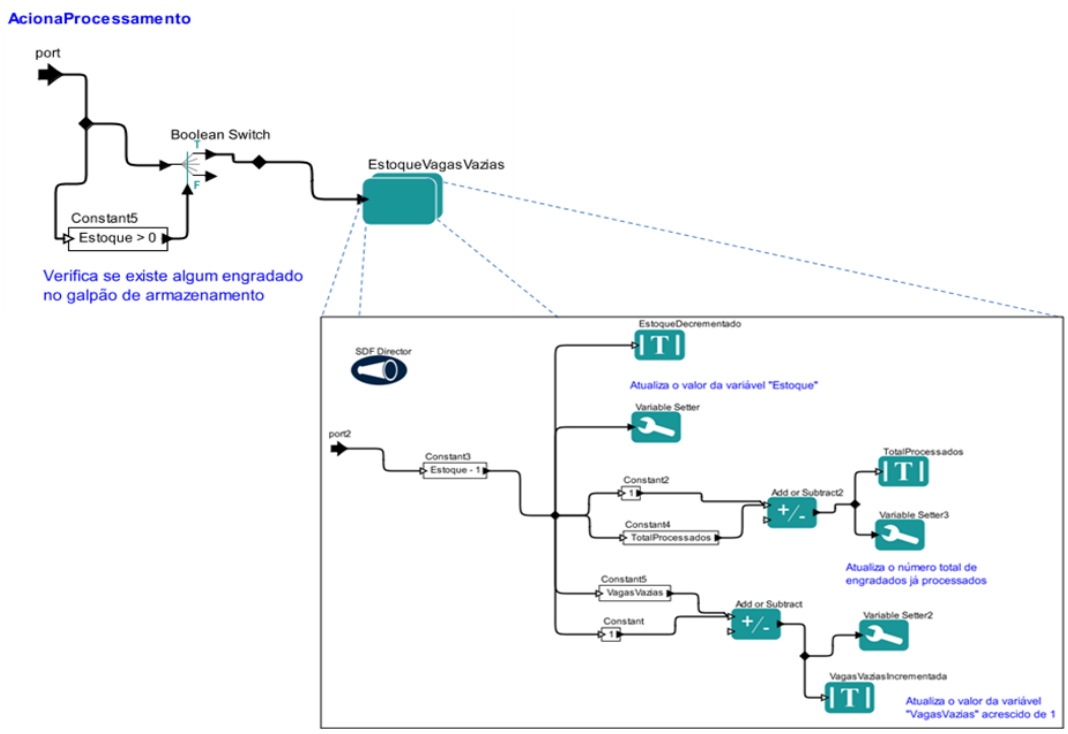

Figura 4.7: *ProcessaEngradado*

Triggers[1]: os *triggers* são atores compostos responsáveis por acionar os atores principais do *workflow*. O funcionamento de cada um dos quatro *triggers* é idêntico. Basicamente, mantém-se um contador até atingir-se certo valor x. Quando o valor x é atingido, dispara-se o *token true* pela porta de saída. A seguir são listados os *triggers* que compõem o *workflow*.

- *AcionaWorkflow*: aciona o ator *VagasVazias*;
- *AcionaEnvioDeCartas*: dispara *EnvioDeCartas*;
- *AcionaProdução*: *trigger* de *ProduzEngradado*;
- *AcionaProcessamento*: aciona o *ProcessaEngradado*.

[1]traduzido no texto como *acionador*

Capítulo 5

Conclusões

Este trabalho tem como objetivo estudar em detalhes como o sistema Kepler implementa o controle de fluxo de dados. Para atingir tal objetivo, além da leitura da documentação disponível do Kepler para entender seu funcionamento, tanto teórico quanto prático, foi necessário estudar os conceitos de Redes de Petri conforme descrito em [3].

A fim de obter resultados mais detalhados e conclusivos, foi escolhido um caso prático para ser implementado na ferramenta Kepler. Conforme descrito ao longo do texto pode-se observar que o controle de fluxo de dados entre dois atores se dá apenas quando estes estão conectados, ou seja, quando há ligação entre a porta de saída de um com a porta de entrada de outro. Embora isso seja uma alternativa para comunicação entre processos que compartilham dados, tal solução não leva em consideração o controle de fluxo de atividades. Essa foi uma limitação muito bem caracterizada na implementação do estudo de caso.

Uma outra importante limitação identificada no sistema Kepler é que não há fluxo de dados entre os atores quando estes se comunicam por meio de atualização de variáveis.

Essas duas limitações dificultam a implementação de importantes e recentes funcionalidades em *workflows* científicos: comunicação entre processos e a orquestração do fluxo de atividades.

Por outro lado, a forma de controlar fluxo de dados disponibilizada pelo Sistema Kepler explica o sucesso do uso dessa ferramenta pelos usuários de bioinformática. De fato, isso é um grande diferencial do sistema Kepler.

Os próximos passos deste trabalho incluem um estudo para criação de novos diretores e atores de modo que o sistema Kepler possa controlar fluxo de atividades e comunicação entre processos.

Parte subjetiva

Capítulo 6

Trabalho de Formatura e BCC

A seguir, relatar-se-ão algumas experiências obtidas com o desenvolvimento deste trabalho e com o curso de Bacharelado de Ciência da Computação IME-USP.

6.1 Desafios e frustações

Inicialmente, o primeiro desafio encontrado foi o de escolher o tema do trabalho de conclusão de curso. Até os primeiros meses deste ano, não tinha a menor idéia de qual área seguir. Comecei a pesquisar as áreas de interesses de diversos professores do IME, buscando uma que se relacionasse com minhas preferências e com meu perfil. A princípio fiquei um pouco desanimado com os resultados desta pesquisa. Mas, felizmente, encontrei a área de *workflows*, uma das áreas de pesquisa do professor Jef. Particularmente, tenho forte interesse nesse tópico, já que tive experiências muito boas e vejo um grande potencial de crescimento e desenvolvimento. Confesso que fiquei surpreso quando "descobri"que algum professor do IME trabalhava com *workflow*. Até então não sabia que este assunto era abordado no mundo acadêmico, apesar de ter forte presença no mundo corporativo.

Outro desafio que enfrentei foi com relação à definição da proposta do trabalho. *Workflows* científicos são muito utilizados em ambientes científicos e acadêmicos, porém ainda há muito o que explorar. Assim, foi preciso escolher cuidadosamente as palavras e frases que compuseram a proposta final para que não ficasse algo genérico, com muitas coisas a se fazer e sem objetivo claro.

Entretanto, o maior desafio que enfrentei no desenvolvimento deste trabalho foi lidar

com o tempo. Neste ano tive um horário muito apertado e com muitos compromissos, estudando no período da manhã, trabalhando a tarde e novamente estudando no turno da noite. A escassez de tempo fez com que eu perdesse várias noites de sono e deixasse a diversão de lado em inúmeras oportunidades. Mas nada como o incentivo de colegas (tanto de trabalho quanto de faculdade) e, principalmente, o carinho e compreensão da família e amigos para superar este obstáculo.

A falta de tempo realmente foi um problema que incomodou bastante. Apesar de ter conseguido gastar boa parte do meu tempo nos estudos, gostaria de ter me dedicado muito mais no desenvolvimento e aprimoramento deste trabalho. Uma das minhas maiores frustrações foi a de não ter tido tempo para criar novos diretores e atores no Kepler. Acho que isso influenciaria positivamente nos resultados finais.

6.2 Disciplinas mais relevantes para o trabalho

Dentre as disciplinas do BCC que considero terem sido mais relevantes e essenciais para o desenvolvimento deste trabalho estão:

- MAC0110 - Introdução à Computação

  Em minha opinião, é uma das matérias mais importantes do BCC. Esta disciplina introdutória coloca o aluno dentro do mundo da computação e inicia alguns conceitos como lógica de programação e linguagens algorítmicas. É por meio dela que se tem o primeiro contato direto com programação. No meu caso, a adoção de Java como linguagem base desta displina foi fundamental para que tivesse uma simpatia maior em relação às outras linguagens, já que nunca tinha escrito uma linha de código sequer antes de entrar na faculdade. Essa "simpatia"adquirida alavancou a minha curiosidade e vontade de aprofundar cada vez mais meus conhecimentos no mundo Java. O conhecimento adquirido nessa linguagem foi muito importante para este trabalho.

- MAC0122 - Princípios de Desenvolvimento de Algoritmos

  Disciplina chave do curso. Independentemente da área que o aluno pretenda seguir, MAC0122 é uma matéria essencial para a formação acadêmica.

- MAC0441 - Programação Orientada a Objetos

  Certamente uma das disciplinas mais importantes para o desenvolvimento do trabalho. A ferramenta Kepler é modelado nos preceitos da orientação a objetos, com estruturas hierárquicas complexas e intensa modularização do sistema em diversos componentes. A leitura da documentação disponível sobre o Kepler exigiu um bom conhecimento na área de orientação de objetos, repleta de diagramas de UML, de classes e de casos de uso.

- MAC0323 - Estruturas de Dados

  Falando especificamente sobre o trabalho, a disciplina foi fundamental na parte de *Dados* do sistema Kepler.

- MAC0328 - Algorítmos em Grafos

  Embora a disciplina de Algorítmos em Grafos tenha abordado diversos algorítmos importantes e complexos, apenas a idéia básica de grafos foi utilizada no trabalho. A ferramenta Kepler baseia-se na construção visual de grafos para desenvolvimento de *workflows*.

- MAC0438 - Programação Concorrente

  A noção de execução concorrente de processos foi bastante importante em boa da implementação do caso prático.

- MAC0439 - Laboratório de Banco de Dados

  A discplina Laboratório de Banco de Dados foi uma espécie de revisão de alguns conceitos já vistos anteriormente. Toda parte de padrões de workflows e processos de negócios foram estudadas previamente a pedido do professor Jef.

6.3 Experiências obtidas no trabalho

No início do segundo semestre deste ano, o professor Jef pediu que eu e sua aluna de mestrado Grace Borges trabalhássemos juntos no estudo do sistema Kepler. A partir de então houve um grande avanço no desenvolvimento deste trabalho. Acredito que sem essa

interação com Grace os resultados obtidos não teriam sido satisfatórios e os objetivos não teriam sido atingidos a tempo.

Outra experiência bastante marcante durante esse período foi uma apresentação que eu e Grace fizemos ao Calton Pu durante o período que esteve no Brasil. Foi um fato importante do qual o professor Jef, Grace e eu pudemos tirar proveito e esclarecer nossas visões e dúvidas sobre o Kepler.

6.4 Trabalhos futuros

Os próximos passos incluem um estudo para criação de novos diretores e atores de modo que o sistema Kepler possa controlar fluxo de atividades e comunicação entre processos.

Continua sendo imprescindível a leitura cada vez mais aprofundada de artigos e documentação disponível sobre a ferramenta.

Referências Bibliográficas

[1] S. Bowers and B. Ludaescher. Actor-oriented design of scientific workflows. *In International Conference on Conceptual Modeling (ER´2005)*, 2005

[2] Edward A. Lee and Stephen Neuendorffer, "Concurrent Models of Computation for Embedded Software", *Technical Memorandum UCB/ERL M04/26*, University of California, Berkeley, CA 94720, July 22, 2004.

[3] Grace A. P. Borges, "Controle de dados em Processos de Negócio e Workflows Científicos", *Exame de Qualificação de Mestrado*, 2007

[4] Carl Hewitt. et al. Actor Induction and Meta-evaluation, *Conference Record of ACM Symposium on Principles of Programming Languages*, January 1974

[5] Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems, *Doctoral Dissertation*, 1986

[6] Workflow Management Coalition, 1995

[7] Committee Draft ISO/IEC 15909. High-level petri nets - concepts, definitions and graphical notation. Oct 1997.Version 3.4.

[8] B. Ludäscher et al. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience*, Dec. 2005

[9] Shawn Bowers, Bertram Ludascher, Anne H.H. Ngu and Terence Critchlow. Enabling ScientificWorkflow Reuse through Structured Composition of Dataflow and Control-Flow. *22nd International Conference on Data Engineering Workshops (ICDEW'06)*. 2006

[10] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng (eds.), "Heterogeneous Concurrent Modeling and Design in Java,(Volume 1:Introduction to Ptolemy II)", *Technical Memorandum UCB/ERL M03/27*, University of California, Berkeley, CA USA 94720, July 16, 2003.

[11] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng, (eds.), "Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture)", *TechnicalMemorandum UCB/ERL M03/28*, University of California, Berkeley, CA USA 94720, July 16, 2003.

[12] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng (eds.), "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)", *TechnicalMemorandum UCB/ERL M03/29*, University of California, Berkeley, CA USA 94720, July 16, 2003.